

Node.js & HTML5

開發思維與入門學習

Thinking and Getting Started

Jollen Chen

Node.js & HTML5 開發思惟與入門學習

Thinking and Getting Started

Jollen Chen

This book is for sale at

<http://leanpub.com/html5-javascript-thinking>

This version was published on 2014-03-13



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 - 2014 Jollen Chen

Contents

JavaScript 設計模式	2
Object	2
宣告 Class	3
使用 Factory Pattern	5
Constructor Pattern	7
Design Pattern for Front-End	8
Module Pattern	8
jQuery Pattern	11
選擇器模式	13
jQuery Pattern 實作 (jQuery 插件開發)	14
Prototype Pattern	16
其它模式	17
 HTML5 軟體開發的概念	 18
HTML5 的 Runtime 是瀏覽器	19
從 Chrome 瀏覽器談起	21
理解平臺的概念	22
從 Web Page 到 Web App	23
HTML5 時代來臨	23
JavaScript 也能開發雲端服務	25
為 HTML 加入應用程式特性	26
Over HTTP	27
Web-Oriented Architect	28
Data Push 設計模式	30
Device API 的革命時代	32

CONTENTS

重要的資訊交換格式: JSON	35
結論	37
Node.js 入門 - URL Routing 篇	38
觀看 Nodejs 線上課程	38
第一個 Node.js 程式	39
製作 Node.js 模組	43
URL Routing	45
設計 HTTP API	51
解析 Query String	53
結論	55
Node.js 入門 - WebSocket 與 JSON 篇	57
建立 WebSocket Server	58
儲存用戶端 WebSocket 連線	60
學習 JSON 格式	63
JSON Stringify	66
製作 WebSocket 用戶端	69
結論	82
軟體思惟 - Lambda 篇	84
Lambda	84
Callback Function	86
使用 TypeScript	89
軟體思惟 - Web Service 篇	91
再探 HTTP API	91
REST	93
CRUD 與 HTTP Method	94
結論	96
軟體思惟 - Non-blocking IO 篇	97
Node.js File System	98
讀取多個檔案	100
結論	105

CONTENTS

Node.js 應用 - Express.js 入門	106
Express.js 初體驗	106
Express.js 的學習建議	112
MVC 與 HTML Template Engine	113
學習 Jade 程式語言	113
解析 app.js	117
Express URL Routing	119
Middleware 的觀念	122
Express.js 應用 - Middleware	124
使用 <i>app.get()</i> 撰寫 Middleware	124
Middleware 與流程控制	128
使用 <i>app.use()</i> 撰寫 Middleware	131
常用的 Express.js Middleware	134
結論	136
REST API 架構 - 使用 Express.js	137
關於 SOA 與 3-Tier 架構	138
Presenetation 在 Client 端	140
Express.js 與 REST API	142
RESTful 架構實務 - 即時聊天室	144
結論	154
REST API 應用 - 使用 jQuery	155
呼叫 REST API - 使用 jQuery	155
認識 Key-Value Pairs 觀念	168
結論	170
MVC 架構實作 - Backbone.js 入門	171
初學 Backbone Way	171
認識 <i>View.sel</i>	181
認識 Backbone.Model	181
認識 Backbone.Model.fetch	189
認識 Backbone.Model.save	191
結論	193

作者

Jollen (陳俊宏), Moko365 與 Mokoversity 創辦人。

熟悉 Android、Embedded Linux、Frontend 與 Node.js 軟體技術。在多家全台前一百大科技公司擔任顧問，過去曾參與過全球第一個開放手機平台計畫 Openmoko，負責大中華區事務。目前兼任手機廠的架構設計工程師。針對手機軟體開發所創辦的教育訓練與顧問公司：仕橙 3G 教室 (Moko365)，在兩岸業界極負盛名。目前也和 MagicLEGO.org 協同推廣基於三星方案的多核心平臺。並擁有二項與嵌入式系統有關的美國專利。

- Email: jollen@jollen.org
- Blog: <http://www.jollen.org/blog>

JavaScript 設計模式

JavaScript 或許不是物件導向式語言（OOP），因為目前流行的 JavaScript 語法並沒有很明顯的 OO 特徵。但是，JavaScript 編程時，到處充滿物件導向的觀念。本章針對 JavaScript 的入門觀念，進行簡要的整理。

對初學者而言，程式語法是重要的課程，但是要入門 JavaScript 編程，學習語法還不夠。幾個重要的入門觀念，不但能幫助初學者寫好程式，更可以說是 JavaScript 真正的入門課程：

- Object & Function
- Instantiable Function
- Callback & Lambda
- Inheritance & Prototype
- MVP 設計模式
- MVVM 架構模式

Object

JavaScript 第一個觀念，就是「物件生成」。利用 `function` 關鍵字來宣告類別，並且利用 `new` 關鍵字來實例化，是生成物件最簡單的寫法。另外一個做法是利用 `Object.create()` 方法。

在 JavaScript 裡，也可以直接宣告物件。利用一對大括號所宣告出來的變數，都是物件。

JavaScript 不是強型別（Strong data type）的程式語言，任何的變數宣告，只要使用 `var` 關鍵字即可。要生成（Create）物件時也一樣，以下是一個範例：

```
1  var person = {  
2      name: "Jollen",  
3      job: "Software Developer",  
4  
5      queryJob: function() {  
6          alert(this.job);  
7      }  
8  };
```

上述的表示式執行後，可以得到 `person` 物件。如同典型的物件導向觀念，在物件裡會有 attribute 與 method。

在 `person` 物件裡，有二個 attribute 與一個 method。例如，要呼叫 `person` 物件的 `queryJob()` method:

```
1  var person = {  
2      name: "Jollen",  
3      job: "Software Developer",  
4  
5      queryJob: function() {  
6          alert(this.job);  
7      }  
8  };  
9  
10 person.queryJob();
```

直接宣告物件是一種寫法，另外一種寫法是稱為 `Instantiable Function`，說明如下。

宣告 Class

同樣地，JavaScript 沒有類似 `Class` 這樣的語法，所以要宣告 `Class` 的話，以 `function` 關鍵字來實作即可，等價於函數宣告：


```
1  function Person(name, job) {
2      this.name = name;
3      this.job = job;
4      this.queryJob = function() {
5          alert(this.job);
6      };
7  }
```

將 Function 關鍵字做為 Class 的宣告，自然就要討論是否能以 new 關鍵字將 Class 實例化成物件。在 JavaScript 裡，可以支援這樣的寫法。以下是一個實例化（Instantiate）的例子：

```
1  <!doctype html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>MokoCrush</title>
6  </head>
7
8  <body>
9  <script>
10 function Person(name, job) {
11     this.name = name;
12     this.job = job;
13     this.queryJob = function() {
14         alert(name + "'s job is " + job);
15     };
16 }
17
18 var person = new Person("Jollen", "Software Developer");
19
20 person.queryJob();
21 </script>
22 </body>
23 </html>
```

在這個例子裡，`person` 是 `Person class` 的實例化。所以，調用 `person.queryJob()` 方法時，所看到的畫面如下：

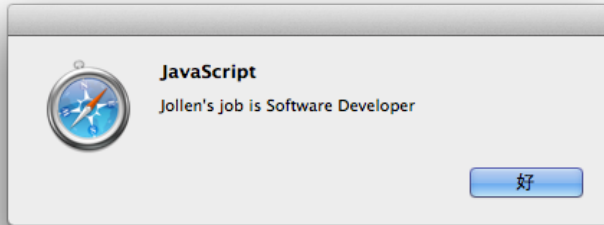


圖 1.1 example-1-1.html 執行結果

JavaScript 裡生成物件的做法：

- 使用 `var` 關鍵字宣告物件
- Instantiable Function

使用 **Factory Pattern**

Instantiable Function 可以利用 `new` 關鍵字生成它的物件，我們可以進一步將生成物件的過程封裝起來。這就是 **Factory Pattern** 的用途。

Factory Pattern 是很常用的一種物件生成設計模式，在軟體工程的領域裡，它用來將建立物件的過程抽象化。例如，將上述的例子重構，改以 `factory pattern` 來實作：

```
1  // 不可做為 Constructor Pattern
2  function personFactory(name, job) {
3      var o = new Object();
4
5      o.name = name;
6      o.job = job;
7      o.queryJob = function() {
8          alert(name + "'s job is " + job);
9      };
10
11     return o;
12 }
13
14 var person1 = personFactory("Jollen", "Software Develop\
15 er");
16 var person2 = personFactory("Paul", "Product Manager");
17
18 person2.queryJob();
```

範例中，person1 以及 person2 物件的生成被抽象化了，也就是被封裝了起來。在主程式裡，我們看不到 new 關鍵字的使用，利用了 personFactory() 函數將真正的物件生成封裝起來。不過，使用 factory pattern 的話，就沒有辦法判斷物件原本的 class type 了。例如：

```
1  alert(person1 instanceof Object); // true
```

在 personFactory() 裡生成的物件，實際上是 Object 的實例化。實作時，如果需要明確地生成 Person 類別的物件，還是要以先前的做法為主：

```
1  // 可做為 Constructor Pattern
2  function Person(name, job) {
3      this.name = name;
4      this.job = job;
5      this.queryJob = function() {
6          alert(this.job);
7      };
8  }
9
10 // 將 Person() 視為 constructor
11 var person = new Person("Jollen", "Software Developer");
12
13 alert(person instanceof Person); // true
```

在這個例子裡，物件 `person` 確實是 `Person` 類別的實例化。將 `Person()` 視為 `constructor`，而不是函數，觀念上就能達成上述的目的（判斷物件原本的 `class type`）。

Constructor Pattern

以上的觀念，可以用一個觀念來總結：函數（`functions`）就是物件（`objects`）。這是 JavaScript 極為重要的觀念，實務上，大量被應用在 `Callback function` 的實作上。上述的例子，將 `Person()` 視為 `constructor` 來使用；這個觀念亦稱為 `Constructor Pattern`。

上述二個例子，只有第二個寫法，才能做為 `Constructor Pattern` 的觀念。其差異如下：

- 不需要明確地產生物件
- 直接使用 `this` 物件來指定屬性（`properties`）與方法（`method`）
- 不需要 `return` 敘述

另外，根據物件導向的觀念，Constructor 函數的命名，要以大寫字元開頭。

Design Pattern for Front-End

用白話文來解釋設計模式（Design Pattern）：撰寫代碼與解決特定問題的「固定作法」。由於 HTML5+JavaScript 的廣泛流行，有一些專門針對 HTML+JavaScript 的設計模式被提出，其中最重要的二個設計模式為：Module Pattern 與 jQuery Plugin Pattern。

要學好 HTML5+JavaScript 應用程式開發，上述二個設計模式不能不學。此外，大家要有一個認知，就是：只懂 JavaScript 語法是做不出好高級品的，意思是撰寫不出良好品質的程式碼。在 HTML5+JavaScript 的世界裡，設計模式才是重點。要強調一點，並不是不需要學習 JavaScript 語法，而是只懂得語法並不足以應付未來的研發工作。

Module Pattern

如同此設計模式的名字所述，module pattern 的目的是把程式碼「模組化」；將 JavaScript 的程式碼模組化，有其固定的做法。要將程式碼模組化，前提是將程式碼 Closure，即封閉性。接下來，以一個連續的範例，來說明 Module Pattern。在繼續進行下去前，請務必熟讀並了解 JavaScript 物件的觀念。

使用 Private/Public 觀念

請不要再使用 Local variable 與 Global variable 的寫法了，這在 HTML5+JavaScript 的世界裡行不通。以下是一個標準的錯誤示範：

```
1  var count = 0;                                //GLOBAL
2
3  function incrementCounter() {
4      count++;                                    //GLOBAL
5      return count;
6  }
7
8  function resetCounter() {
9      var orig;                                    //LOCAL
10
11      orig = count;
12      count = 0;
13  }
```

這個例子只是讓大家瞧瞧 Local variable（區域變數）與 Global variable（全域變數）的寫法。現在，我們更上一層樓，把 Local variable 重構為 Private attribute，並把 Global variable 重構為 Public attribute。如下：

```
1  var testModule = (function () {
2
3      var counter = 0;                                // Private
4
5      return {
6          incrementCounter: function() {                // Global
7              return ++counter;
8          },
9
10         resetCounter: function() {                    // Global
11             counter = 0;
12         }
13     };
14
15 }) ();
```

原本只是利用 `function` 關鍵字來定義一個類別 (Class)，現在更進一步以 `Module` 的方式將類別「封閉」。結果是，原本的區域變數 `counter` 變成了 `Private attribute`；原本的二個函數，現在成為了 `Global method`。將原本的程式碼，製作成 `JavaScript module`，這就是 `Module pattern` 的基本精神。

`Closure` 的目的，在避免全域變數的污染。變數污染，指的是自己的全域變數，被外部的程式碼做修改。以上述例子來看，如果 `counter` 沒有在 `Closure` 裡的話，其它地方的 `JavaScript` 就可以任意修改其值：因為 `counter` 是全域變數。為了避免這個問題，將程式碼 `Closure` 起來：只有 `Closure` 裡的程式碼，能修改 `counter` 變數。

外部程式碼，無法修改「封閉」程式碼裡的變數。概念上，以 `module` 將 `attribute` 與 `method` 進行封裝，這個觀念就是「`closure`」。簡單說，將封閉的程式碼，放進 `testModule` 變數：`testModule` 成為一個模組。模組可被使用。

`JavaScript` 沒有明顯的物件導向語法，所以上面的一切都是觀念問題，而不是語法問題。軟體工程領域，很多時候都是在處理這樣的哲學思想；技術面只是整個軟體工程的一小部份。

所以，我們要把軟體開發當做一個創作過程，而不是寫程式 (Coding) 的過程。

Import Modules

`jQuery` 是很好用的程式庫，它也被製作成 `module`。不過，由於 `jQuery` 強的擴充性，讓 `jQuery` 擁有為數可觀的「`plugins`」。所以，`jQuery` 本身就是一個平臺 (Platform)，或是一個開發框架 (Framework)，再加上有如空氣般，`jQuery` 在 `Web` 相關領域真是無所不在，所以就有 `jQuery pattern` 的出現，後續將會介紹 `jQuery pattern`。

`jQuery` 在 `JavaScript` 領域已經自成一格。

`jQuery` 本身是一個 `module`，`module` 可以匯入 (Import) 使用。以下是一個範例：

```
1  var testModule = (function (jq) {
2
3      var counter = 0;
4
5      // Private
6      function showHTML() {
7          jq(".header").html("<h1>" + counter + "</h1>");
8      };
9
10     return {
11         incrementCounter: function() {
12             return ++counter;
13         },
14
15         resetCounter: function() {
16             counter = 0;
17         },
18
19         setCount: function(val) {
20             counter = val;
21         },
22
23         showCount: function() {
24             showHTML();
25         }
26     };
27
28 }) ($);
```

jQuery Pattern

jQuery pattern 就是開發 jQuery 插件 (Plugin) 的方式，所以技術上倒也沒有什麼學問。不過，jQuery pattern 有很高深的哲學道理，意思是說，在軟體工程領域裡，它創造了一個獨

特的觀念。這個觀念就是 jQuery 知名的 “\$” (Dollar sign)，也就是「Selector」。

以下的例子，就是 jQuery pattern:

```
1 $( "div#news" ).html( "<h2>News Today</h2>" );
```

從 jQuery 設計模式的角度思考，上述的寫法似乎不太好。從 jQuery 設計模式的角度思考，如果今天我們想要透過 WebSocket 與伺服器溝通，並且在一個 “div” 裡來顯示結果，應該怎麼設計呢？想法如下：

- 將 WebSocket 的功能寫成一個 function
- 將 JavaScript function 封裝成 module
- 在 jQuery 裡擴充新的函數，簡單說，就是製作一個 jQuery 插件 (Plugin)

以下是一段程式碼樣板：

```
1
```

上述的寫法，採用匿名模組來實作。接者，再將程式碼儲存為 jquery.websocket.js。使用方法如下：

```
1 <!DOCTYPE html>
2 <head>
3 <script type='text/javascript' src='./jquery.min.js'>
4 <script type='text/javascript' src='./jquery.websocket.\
5 js">
6 </head>
7 <body>
8 <div id="message"></div>
9
```

```
10 <script type="text/javascript">
11 $("#message").createWebSocket();
12 </script>
13 </body>
14 </html>
```

這是採用 jQuery pattern 的寫法。這種做法可以良好地組織 HTML5 與 JavaScript 程式碼。此外，JavaScript 的 module 具備「Closure」的特性，即封閉性，可以避免一些衍生問題。

由於 HTML5+JavaScript 的設計思想，和 Native App 的作法有很大的不同，所以了解 HTML5+JavaScript 的應用程式「如何設計」，會是重要的一門課。了解設計模式，除了能有效組織 HTML5+JavaScript 程式碼外，也能做出正確的設計。

選擇器模式

我們在實作 Web Socket 連線生成時，利用了 jQuery pattern，這是一種選擇器模式。為什麼要使用選擇器模式，除了程式碼的組織較好外，另一個原因就是效能。事實上，讓程式碼組織更良好是次要的理由，真正的、主要的、最重要的原因是：使用選擇器方式可以讓 JavaScript 程式碼效能更好。

根據不同瀏覽器的實作，選擇器模式可以達到超過十倍以上效能。再回顧一次我們的寫法：

```
1 <div id="message"></div>
2 <script type="text/javascript">
3 $("#message").createWebSocket();
4 </script>
```

總計利用了三個模式：

- 以 Closure 模式將類別封閉，這與 static class 有關係，在這裡先不做討論

- 使用選擇器模式，範例採用目前最流行的 jQuery selector “\$”
- Read/Write Div Pattern

選擇器模式的效率取決於瀏覽器本身的實作，不過，以選擇器模式來代替直接存取 DOM，一般相信是最好的做法。因此，現代的 JavaScript 程式庫，幾乎都利用選擇器模式來實作 (jQuery 一直都是最佳例子)；當我們實作自己的 JavaScript 程式庫時，也該善用選擇器模式。

典型的選擇器模式，是直接呼叫 DOM 的 API:

```
1 document.querySelector( "#header" );
```

不過，使用 jQuery 的選擇器「\$」是目前的主流做法。

jQuery Pattern 實作 (jQuery 插件開發)

簡單來說，jQuery pattern 就是撰寫 jQuery Plugins。要開發 jQuery 的插件，是相當輕鬆愉快的工作，這都歸功於 jQuery 的優良架構。

Step 1: 加入新增函數

在 \$.fn 物件裡，加入新的函數屬性。範例：

```
1 $.fn.hello = function() {  
2     // your code here  
3 };
```

Step 2: 將程式碼 Closure

```
1  (function($) {  
2      $.fn.hello = function() {  
3          // your code here  
4      };  
5  })($);
```

為什麼要將程式碼 Closure（封閉）的原因，前文已做過說明。

Step 3: 儲存為獨立檔案使用

將上述程式碼，儲存為獨立檔案，例如：jquery.foo.js，並在 HTML5 裡使用。例如：

```
1  <!doctype html>  
2  <html lang="en">  
3  <head>  
4      <meta charset="UTF-8">  
5      <title></title>  
6  
7      <script type='text/javascript' src='./jquery.min.js\  
8  "></script>  
9      <script type='text/javascript' src='./jquery.foo.js\  
10  "></script>  
11  </head>  
12  
13  <body>  
14      <div class="content">  
15      </div>  
16  <script>  
17  $(".content").hello();  
18  </script>  
19  </body>  
20  </html>
```

上述的做法，是將自己的實作，設計成 jQuery Plugin 的形式。制作 jQuery 插件是非常簡單的，只要以上三個步驟即可完成。

Prototype Pattern

每一個 JavaScript 的函數，都包含一個 *prototype* 的屬性。*prototype* 是一個物件，並且該函數的實例化，都能共用 *prototype*。所謂的 Prototype Pattern，就是擴充 *prototype* 物件。

Prototype Pattern 是非常重要的 JavaScript 設計模式，在開發程式庫時，被大量使用。以下是一個實例：

```
1  // 可做為 Constructor Pattern
2  function Person() {
3  }
4
5  Person.prototype.name = 'Jollen';
6  Person.prototype.sayName = function() {
7      alert(this.name);
8  };
9
10 // 將 Person() 視為 constructor
11 var person = new Person();
12 person.sayName();           // 'Jollen'
```

Prototype Pattern 的優點：

- 可以在實例化前，就定義好物件的 property 與 method; Constructor Pattern 的做法則是在實例化時，透過 Constructor 來定義
- 實例化之間共用 *prototype*

實例化之間是共用同一份 *prototype*，這是 Prototype Pattern 與 Constructor Pattern 最大的不同。

其它模式

本章整理了 JavaScript 實務上的入門觀念，當然還有許多主題等著我們去探索。例如：MVC。MVC (Model-View-Controller) 是一個年代久遠的設計模式。在軟體工程領域，也是一個使用率非常高的模式。在開發 HTML5+JavaScript 應用程式時，可以使用 MVC 模式將程式碼組織得更完善。做法如下：

- model/ 目錄
- controller/ 目錄
- view/ 目錄
- media/ 目錄

上面的做法，其實就是 Django 組織一個專案的方式。Django 是一個知名的 Web 開發框架。關於其它更進階的主題，將在後文繼續介紹。

HTML5 軟體開發的概念

HTML5 不只是技術，它是 Mobile + Cloud 的全新時代。HTML5 重新發明了 Web，現在的 Web 已不同以往。對完全不懂 HTML5 的創業人，本章內容將對您有很大的幫助。

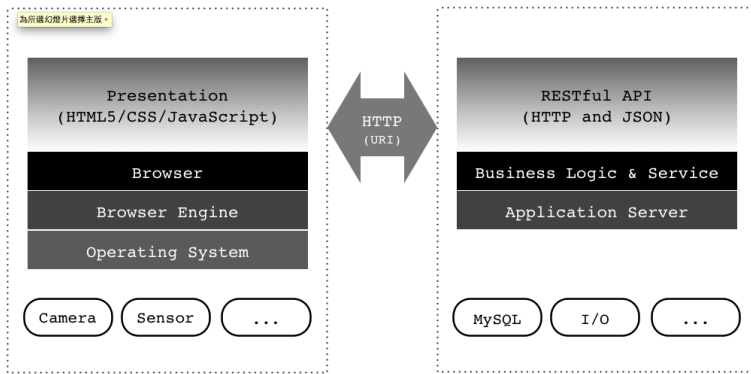


圖 2.1: Web Service 導向架構

圖 2.1 把 HTML5 與 Cloud 的關係，做了很簡單的說明。圖 2.1 同時也是 HTML5 軟體開發的起點，這是一張「系統」架構，而不是「應用程式」架構。因為廣意的 HTML5 包含用戶端與伺服器端；目前的用戶端，大多是行動裝置。所以 Mobile + Cloud 正好表達了 HTML5 中心哲學。

HTML5 系統架構屬於 Web Application 的做法，技術上來說，這是 Web 導向架構（WOA, Web-Oriented Architect）。圖 2.1 的重點整理如下：

- Presentation 就是 View，也就是 App UI 的部份。用戶端使用 HTML5/CSS/JS 來打造 UI。

- UI 可以透過瀏覽器，來呼叫底層的 Device API。目前這個部份，需要在瀏覽器引擎裡，整合額外的軟體。目前最知名的 Device API 就是：PhoneGap。
- PhoneGap 技術，讓 JavaScript 可以呼叫到 Device API，HTML5 App 使用 JavaScript 就可以控制硬體。可以想像成：透過網頁來控制硬體。
- 最重要的軟體是瀏覽器，它是應用程式的 Runtime。
- 伺服器端以 NodeJS 打造專用的 Application Server。
- NodeJS Application Server 提供 API：即 Web Service。這些 API 採用 URL 的形式呈現，也稱為 Open API、HTTP API 或 Platform API。技術上，正式的名稱為 RESTful API。
- 用戶端呼叫 Server 的 RESTful API 來存取服務。
- NodeJS 使用 JavaScript，但能存取 I/O，例如：資料庫、檔案等。這是 NodeJS 的主要特點之一。
- 用戶端使用 JavaScript，伺服器也用 JavaScript，所以開發者，都用 JavaScript 程式語言：這是一個 All in JavaScript 的時代。
- 以 JSON 格式來交換資料，降低頻寬的使用。

HTML5 的 Runtime 是瀏覽器

廣義上的 HTML5 必須搭配許多技術來綜合應用，所以 HTML5 到底怎麼跟這些技術搭配、HTML5 的軟體開發思想是什麼，以及 HTML5 應用程式的設計原則為何？

HTML5 來了。什麼最重要：Browser 最重要。因為所有 HTML5 App 都在 Browser 環境裡執行，所以，HTML5 App 的 Runtime 就是瀏覽器。

Runtime 是執行環境。過去應用程式都是在作業系統平臺上執行，並且採用 C/C++ 程式語言編寫，我們便將這類型的應用程式稱為 Native Application，即主動式應用程式，代表應用程式被編譯後（Compile）就是一個可主動執行的執行檔。

接下來 Android 出現後，帶動一個以 Java 程式語言來開發 App 的風潮。Java 程式語言早在 Android 問世前就已經被發明出來，雖後在 Android App 的年代人氣達到頂峰。利用 Java 程式語言編寫的應用程式，都需要 Java 虛擬機 (Virtual Machine) 才能執行；Android 的 Java 虛擬機稱為 Dalvik VM。所有以 Java 撰寫的 Android App 都在 Dalvik VM 上執行，它是由虛擬器來管理並直譯，不能主動執行。

可以這樣想，第一代的 App 使用 OS 做為 Runtime。第二代的 App 使用 Java Virtual Machine 做為 Runtime，例如：Android。第三代的 App 將使用 Browser 做為 Runtime。所以，Runtime 就是個關鍵技術。以前沒有掌握好 OS 沒關係，過去沒掌握好 VM 沒關係，現在還是沒掌握好 Browser 技術，就很有關係了。

同時，過去桌面電腦 (PC) 的瀏覽器，正快速往手機裝置移動。各家軟體大廠無不加碼研發人才，努力打造一個能完全相容 HTML5 的瀏覽器，在 2012 年 10 月份現身的 Windows 8 Mobile Phone，也在 HTML5 做了很大的改進。2012 年，更傳出 [Facebook 有意購併老字號的瀏覽器開發商 Opera¹](#)，這個傳言最後沒有成真，但顯示了瀏覽器在行動裝置產業的重要性。

HTML5 成為移動網路的標準已經是既定的事實了，唯一的關鍵是它的普及性。而行動瀏覽器技術，直接決定了 HTML5 的普及速度。這會是科技業的大盛宴，當行動瀏覽器成熟，並且安裝到每一台行動裝置上時，HTML5 將開始展現它的強大影響力。這個的影響力的層面相當廣，它會普及到各行各業。

所有的行業，都需要雲端做為媒體，將資訊與服務 Web 化，Web 化後都能整合到行動裝置上。在行動裝置上，可以使用瀏覽器來使用這些資訊與服務。很明顯地，HTML5 的影響力將遠超過當年的 Android。

當年的 Android 作業系統，還不算是各行各業的事情。但是，HTML5 將影響到科技業、銀行業、出版業、遊戲業等等，數都數不清的行業別都將因 HTML5 起變化。將透過雲端將資

¹<http://www.techbang.com/posts/9559-facebook-to-buy-opera-browser>

訊與服務，傳遞到手機上的行業，都需要 HTML5。

所以，從產業的角度來看，「HTML5 = Mobile + Cloud」，這很簡單地說明了一切。從技術上來看，瀏覽器也是非常重要的「通路」，因為掌握了瀏覽器，等於控制了使用者的手機。瀏覽器開發廠，可以在瀏覽器「內置」一些資訊與服務。在瀏覽器內置服務，這一想就知道是多麼可怕的一件事情，這比起當初微軟在其作業系統裡內建 IE 瀏覽器還可怕。

接下來的幾年，科技業的主流發展趨勢，幾乎正式抵定了，就是「手機、行動瀏覽器與雲端應用」。各大瀏覽器與 HTML5 的相容性，成為相當重要的指標。因運而生的網站 html5test.com 可以幫助我們了解瀏覽器的 HTML5 相容性。

根據 html5test.com 的測試報告，桌面瀏覽器 (Desktop) 平均分數仍然領先行動 (Mobile) 瀏覽器，可見 HTML5 行動瀏覽器，仍有很大的發展空間。

目前在桌面瀏覽器的部份，來自北京的 [Maxthon 瀏覽器](#) 以 476 分取得第一²，領先第二名的 Chrome 29。在行動瀏覽器方面，Opera 目前是市佔率的龍頭，在 [html5test](#) 上的分數也佔於領先地位。不過，尚未正式發表的 Tizen 目前也突破了 400 分大關，是值得留意的潛力股。

從 Chrome 瀏覽器談起

Android 版的 Chrome 瀏覽器出現了，Chrome 是 HTML5 發展史的一個重要指標。預計今年 HTML5 (Chrome) 將全面走入各式產品，這宣告純硬體時代正式結束了。未來的產品，少了軟體與雲端的加持，將顯得平庸而無奇。純硬體能走進市場的機會，只會越來越少。手機肯定首當其衝。特別是 Android 版的 Chrome 瀏覽器現身了，它將引發新軟體革命嗎？

我們知道，雲端應用目前有二大龍頭：Google 與 Facebook。Google 老大哥的 Gmail、相簿、位置服務等，儼然成為一項民

²<http://html5test.com/results/desktop.html>

生必須品。Facebook 則是在社交網路 (Social Networking) 領域獨佔鰲頭。他們都是網路巨擎，也都是以平臺 (Platform) 的概念經營。平臺的概念是什麼？簡單說，就是提供開發者 API 服務。

理解平臺的概念

平臺是一個很容易理解的概念，就像大家手上的手機，裡頭安裝了許多使用到 Google 以及 Facebook API 的 App；這些 App 都會透過「雲端」，存取其服務。iPhone 與 Android 手機裡的這些「雲端 App」，所使用到的核心技術，就是 HTML5。這代表著，只要 HTML5 的規格能開始推出草案³ (Draft)，並且手機上的 HTML5 瀏覽器技術更加成熟，手機行業將會展開一場 HTML5 大戰。

HTML5 大戰就是雲端運算的戰爭，這肯定是新一波的軟體革命。以 HTML5 技術，結合網路服務、開發 App，並整合至手機，將成為顯學。

所以，筆者認為，Android 版 Chrome 的到來，從產業的角度來看，肯定是一個重要指標，具有特別的意義。它將帶領 HTML5 往前衝刺。其實，Chrome 很早就是 HTML5 的領頭羊了，例如：早在 2010 年，Google 就宣佈以 HTML5 取代 Google Gear 技術⁴，從這裡可見一斑。

我們可以這樣假設：有了 Chrome，雲 (Cloud Computing) 就更容易放進裝置裡。正因為如此，所以 Chrome 的出現，有了 HTML5 大戰的煙硝味。Chrome 將加速雲端應用走進手機 App，所以手機不能只有硬體功能，硬體廠將面臨新的一波挑戰。

³<http://www.jollen.org/blog/2012/02/html5-evolution.html>

⁴<http://www.ithome.com.tw/itadm/article.php?c=66509>

從 Web Page 到 Web App

現在，我們可以使用 HTML5+JavaScript 來編寫應用程式，並且以瀏覽器來瀏覽。因此，HTML5 應用程式，說白了就是一種 Web 應用程式，透過瀏覽器來「瀏覽」它。這種使用案例已經存在已久，例如：用瀏覽器打開 Gmail 應用程式，收發郵件。所以，HTML5 應用程式並不是新鮮技術，就是我們所熟悉的 Web 應用程式。

Web 應用程式的執行環境就是瀏覽器。以 Web 技術來製作應用程式的概念、價值與優點就不必多說了，例如：跨平臺、跨裝置等等，就是大家熟悉的優點。試想，不管我是用個人電腦、手機或平板電腦，甚致是電視，只要有瀏覽器、有網路，就可以隨時隨地收發 Gmail。正因為這樣的優點，Web 應用程式有相當大的優點，以及潛在的商業價值。而 HTML5 就是 Web 應用程式的標準，這就是為什麼 HTML5 將成為主流技術、為什麼 HTML5 會走紅，以及為什麼各行各業的每個人，都要學習 HTML5 的原因。

HTML5 時代來臨

HTML5 的前身，其實是一份稱為 [Web Applications 1.0⁵](http://www.whatwg.org/specs/web-apps/2005-09-01/) 的網路標準草案，HTML5 的用途，不言而喻。HTML5 的身世說來可憐，在 HTML5 標準制定將近七年的時間，不太被網路界重視。還好，蘋果已故執行長 Steve Jobs 的高瞻遠矚，讓 HTML5 在 2010 年有了強出頭的機會。這是大家知道的知名事件，Apple 官方宣佈不支援 Adobe Flash。

HTML5 希望未來網路世界，有更加豐富的 Web Applications，這個願景即將成真。有二個原因。第一、HTML5 第一個正式的候選草案版本，將在今年，也就是 2012 年推出，這代表所有的瀏覽器 (Browser) 軟體製造商，將會有一份可共同依循的開發標準。這就是業界的鐵則，任何技術標準化後，都能成為「產業標準」。科技業不用懷疑，HTML5 時代來了。

⁵<http://www.whatwg.org/specs/web-apps/2005-09-01/>

第二個原因、手機。HTML5 是標準的乞丐命、皇帝運，當初提出 HTML5 建議的 Opera Software 萬萬沒想到，2010 年 Steve Jobs 公開支持 HTML5，2011 年智慧型手機又大爆發，這些 iOS & Android 手機對於 Web 的熱切需求，促使 HTML5 有了大躍進的發展。我們不用再懷擬，這一切來的很突然，不過都是真的。

另一波的手機與網路革命展開了，產業生態巨變，一波未平、一波又起。前一波，被來勢兇兇的 Android 作業系統敲得頭昏腦漲嗎？還來不及反應過來的話，繫好安全帶，因為 HTML5 來了。

All in JavaScript 的時代

歡迎來到「All in JavaScript」的程式樂園。JavaScript 過去曾經在動態網頁製作上紅極一時，後來有一段時間，因為開發人員重視 Server-side 技術的程度，遠勝於 Client-side，因此 JavaScript 又迅速沈寂。

2003 到 2007 年這段時間，應該是 JavaScript 最谷底的時候。而後在 2007 到 2009 年，因為 Web 2.0 風格網頁，以及 jQuery 的盛行，再度得到開發者的重視。2010 到 2011 年因為 Mobile Native App 的大量流行，使得眾多開發者不再以 JavaScript 做為首選，再度走入低潮。

學好 JavaScript 程式語言

但是，HTML5 來了。jQuery 被大量使用在網頁設計上，不管是視覺效果、特效或使用者介面，因為 jQuery 相當便利的開發模式，讓 JavaScript 再度被重視了起來。再加上 HTML5 在後推了一把，JavaScript 儼然成為今年最受矚目的程式語言。所以，JavaScript 是「王者再臨」的最佳代言人。

現在，JavaScript 的主要用途，已經由過去的動態網頁 (Dynamic Webpages)，轉為開發 HTML5 App 角色；也就是 HTML5 的應用。我們不僅僅使用 JavaScript 製作有動態效果

的網頁，還藉助它來開發大量的 UI interactive、使用者體驗的設計，以及最重要的雲端服務整合。

還有一個很重要的應用，就是「JavaScript in Browser」，也就是利用 JavaScript 來增強瀏覽器的功能，最為大家所熟悉的例子，就是 Google Chrome。Google Chrome 為了增強對 JavaScript 的支援與效能，開發了新的 JavaScript 引擎；在日記「HTML5 在手持裝置將開始爆發式成長」就提到了，「JavaScript 引擎的成熟度是關鍵」。

所以，測試 JavaScript 的使用案例（Use Cases）在各大瀏覽器的效能，更為一項重要的工程工作。更進一步地，由於 JavaScript 現在搭配 HTML5 來開發「軟體」，而不只是用來製作動態網頁，所以研究 JavaScript 的軟體設計模式，當然也就變成一門顯學；目前被廣為推薦的就是「Essential JavaScript Design Patterns」一書。

JavaScript 也能開發雲端服務

時間到了 2012 年，在 HTML5 時代正式啟動的今天，JavaScript 成為軟體工程師的必修語言，也是程式設計初學者的最佳選擇。從去年大約 1.5% 的使用率，飆升到這個月的 3.3% 左右的使用率。再者，被熱烈炒作的「雲端運算」概念，現在也可以使用 JavaScript 來開發 Server-side 的雲端服務；這就是近二年爆起的新技術「NodeJS」。

所以，不管是 Client-side 或 Server-side，無論是網頁或雲端服務，都可以使用 JavaScript 一統天下了，接下來「All in JavaScript」將成為軟體開發的主流。雖然 JavaScript 還不是當今最受歡迎的程式語言，但是在「Browser & Cloud」的領域，頗有王者的感覺。JavaScript 無疑是王者再臨的代表。

HTML5 + CSS3 + JavaScript

JavaScript 之所以在網頁設計上佔有重要地位，很重要一個原因是 jQuery 的流行。jQuery 是一個以 JavaScript 技術開發的

框架 (JS Framework)，並且簡單易學、又易用。使用 jQuery 讓網頁變得活潑、美觀、又具備高度互動性，是簡單不過的事情了。jQuery 是一個框架，網路上有眾多 JavaScript 程式設計師，為它編寫了數以萬計的插件 (Plugings)，這是 jQuery 能成功的重要原因。

例如，我想要設計一個有「淡入淺出」效果的「投影片撥放」網頁，利用 jQuery 以及相關插件，就可以在幾分鐘內完成。現在，瀏覽網頁就好像在使用應用程式一樣，就像我們瀏覽 Gmail 時，使用經驗跟應用程式已經不向上下了。所以，網頁不再只是網頁 (Web pages)，網頁是應用程式了；這就是 Web 應用程式。

為 HTML 加入應用程式特性

有哪些技術是 Web 應用程式的主要元素？首先，當然是 HTML 標籤。「以前」，最新版本的 HTML 標準是 HTML 4.0，但是它沒有「Web 應用程式」的特性，意思是說，HTML 4.0 其實不包含應用程式方面的標籤。然而，HTML 5.0 把這部份加入了，所以，HTML 5.0 是真正能支援應用程式特性的 HTML 標籤，HTML 5.0 是支援 Web 應用程式的第一個 HTML 版本。事實上，HTML5 完全就是朝 Web 應用程式的方向去制定。

HTML5 本身是標籤。標籤的核心精神是描述資料 (Data)，例如：文字內容、圖形、顏色等等，要有互動的 UI、要有動態效果，或是進行計算等「程式語言」的工作，就要在網頁裡加上程式碼，這個程式碼就是 JavaScript。如果覺得寫 JavaScript 很麻煩，jQuery 提供更簡便的方式，讓我們在網頁裡加上這些功能。

所以，要製作 HTML5 應用程式，除了 HTML5 標籤外，也要使用 jQuery，並且也要學習 JavaScript 程式設計。另外，我們也知道，HTML 將外觀樣式 (Style) 分開了，外觀樣式以 CSS 描述；目前 CSS 的最新標準是 3.0 版。

總結來說，HTML5+CSS3+JavaScript 就是 HTML5 應用程式的靈魂。初學者，就是要先掌握這三大技術元素。

Over HTTP

用戶端與服務器是透過 HTTP 協定溝通，所以 Open API 的形式就是 URL。例如：

```
http://www.moko365.com/enterprise/search?q=android
```

伺服器以 API 形式提供服務，供用戶端呼叫使用。API 也可以附加參數，稱為 Query String，如圖 2.2。

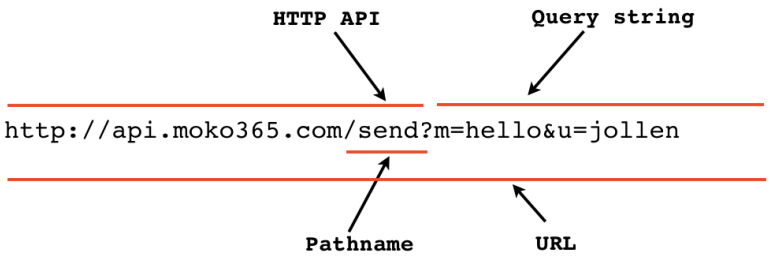


圖 2.2: API 與 Query String

沒錯，這就是過去在學習 CGI 時的觀念。前面的部份是 API，問號後接的就是 API 的參數，稱為 Query string。一串 Query string 裡包含多個參數，以這個例子來說，用戶端在呼叫此 API 時，傳遞給服務器的參數如表 2-1。

表 2-1 API 的參數

參數	值	用途說明
m	‘hello’	指定要傳送的訊息 (message)
u	‘jollen’	指定 Username

NodeJS 的重要技術之一，就是解析並且處理 API 與 Query String。NodeJS 使用了非常巧妙的機制來處理 API 與 Query String，後續會進行詳細的介紹。

另外，上述定義 API 的方式，是典型的 CGI 做法。這樣的 API

是基於 HTTP 協定，因此也稱為 HTTP API。目前定義 Web Service API 的做法，已經有標準了，稱為 REST。這部份後續再做說明。

Web-Oriented Architect

Web 導向架構（WOA, Web-Oriented Architect）的精神一張圖就能講明白了。Web 導向架構著重幾個觀念：

- Device-Server 設計模式
- Device 端使用 Browser，以 Browser 做為執行環境（Runtime）
- Server 端提供 APIs，即 PaaS 概念
- Device-Server 採用非同步通訊（Asynchronous communication）

事實上，非同步通訊大家都使用過，就是 AJAX；AJAX 的第一個 A 就是 Asynchronous。但是考量 Server 端的負載（Loading），以及百萬連線（Millions requests）等級的處理能力需求，應該儘量少用 AJAX 機制。這就是 Device-Server 與 Client-Server 的差別，大家可能還不太明白。所以，將二者的差別簡單整理如下：

- Client-Server 做法：在瀏覽器裡（Client）主動向 Server 請求內容，Client 定時（如：每隔 5 秒鐘）發出請求，持續更新內容。
- Device-Server 做法：在裝置端（Device）和 Server 建立連線，Server 主動將更新內容推送（Push）給裝置端，更明確地說，裝置裡的瀏覽器，瀏覽器再將新內容刷新。

這樣就很清楚了，傳統的 Client-Server 做法是「Data Pull」，即主動去拉資料；Device-Server 的做法是「Data Push」，即推送

資料，由 Server 在必要時才將資料推送給 Device。Data Push 的經典代表作就是 BlackBerry（黑莓機）的郵件服務。

為什麼 AJAX 不好用？因為 Server 要冒著「不知道有多少 Client、不知道同時有多少 Requests」的風險。畢竟，主動權在 Client 端，突然一個時間，幾十萬個 Client 來請求資料，每個人又是發瘋似地，每隔一秒來要一次，Server 豈不掛點了。

要達成 Data Push 的目的，有解決二個技術問題：

- Device 端要與 Server 建立永續性（Persistent）連線，也就是 Socket Connection
- Server 推送出去的資料，格式要有統一標準，且輕量化

要解決這二個問題，就是要使用我們在前一章所介紹到的二項技術：WebSocket 與 JSON。所以，總結目前的說明：HTML5 不只是 HTML5，廣義的 HTML5 應用程式，由眾多技術交互形成。

並且，Web 導向架構是 Client-Server 或 Device-Server。哪一種架構會是主流？這並沒有肯定的答案，不過大略可以區分如下：

- 從 Web Page 角度來看，以 Client-Server 為主，這像是傳統 PC 時代的使用案例從 Web App 角度來看，將 Device-Server 為主
- Web App 的開發思惟，與 Web Page 有很大的不同。目前 Web App 的 UI 製作，採是強調跨螢幕與裝置的特性，這種設計稱為 Responsive Design。並且，Responsive Design 進向以行動裝置為預設值的做法，也就是「Mobie First」

再次總結這二種模式的重點，Client-Server 是 Data Pull 的概念，而 Device-Server 是 Data Push。Data Push 也稱為 Client Pull，表示用戶端主動更新（Refresh）資料；Data Push 則是 Server Push，由伺服器主動推送資料。

Data Push 設計模式

Data Push 的優點之一，就是：處理百萬連線請求。原因很簡單。因為採用的是 Data Push 機制，所以伺服器可以控制用戶數；如果要處理用戶端超過 10 萬個（舉例），就可以將工作分散到其它伺服器。反之，Data Pull 的做法，難處在於，我們很難了解有多少使用者、在什麼時間，同時進行 Refresh。

Server 透過 WebSocket 找到裝置，並以 JSON 格式將資料推送給裝置，這就是 Data Push 的觀念。此外，Server 端應該以什麼技術來實作呢？看來看去，現在最實際的技術就是 NodeJS。以 NodeJS 技術開發一個專用的 Web Server，透過這個 Web Server 將資料包裝成 JSON 後，經由 WebSocket 送出。

Data Push 的設計模式如圖 2.3。

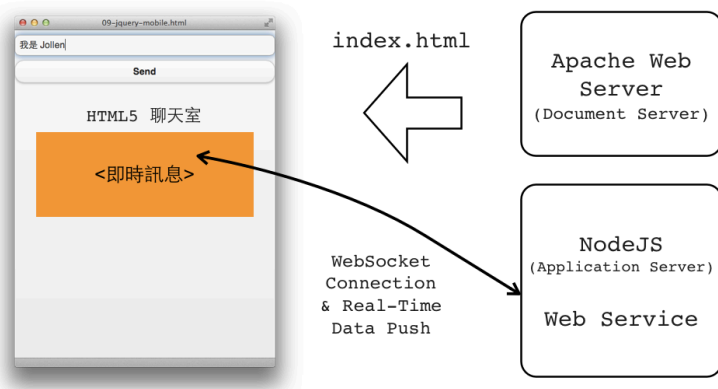


圖 2.3: Data Push 架構圖

步驟如下：

- Client 建立與 Server 的 Persistent Connection (WebSocket)
- Server 保存此連線
- Server 有新的資料時，將 Data Push 給所有的 Client（經由 WebSocket）

- Client 收到資料，並更新 HTML 內容

上述的步驟並不是什麼學問，因為是很典型的機制。不過，HTML5 已經制定標準做法了，就是 WebSocket。

AJAX Refresh 重要性下降

AJAX 本質上是一種 Data Pull 模式，也就是由用戶端（Client）來拉取資料（因此也稱為 Data pull）。過去，開發者經常使用 AJAX 來實作 Refresh 功能。但是 AJAX Refresh 以下的技術缺點：

- 可能造成 Server 的負載
- 不夠即時

即便 AJAX 可以不斷更新（Refresh）資料，但還是不夠即時。要打造「Real-time Web Application」不使用 Data Push 模式是辦不到。因為用戶端，也就是瀏覽器，是以「Refresh」方式向 Web Server 要求資料。例如：每隔 1 秒鐘請求一次資料。

使用 AJAX 與 Refresh 的模式，會讓伺服器端很難預期同時會有多少請求進來。可能在一些熱門時間，忽然有巨量請求，就會讓伺服器負載量提高，甚致有服務中斷的風險。

AJAX 也不夠即時，當伺服器有新內容時，必須依賴用戶端來主動要求，因此會造成一段「Latency」時間。傳統的「聊天室」網站就是這樣設計：必須更新網頁，才能看到新的聊天訊息。

從這個角度來看，NodeJS 的 Data Push 方式，才是好的做法。不過，AJAX 也不是全然沒有用處了。一些「Non Refreshing」的設計需求，AJAX 仍然非常有用，例如：設計「註冊表單」時，當使用者輸入帳號後，以使用 AJAX 向伺服器查詢「使用者帳號是否可用」。

差異是：使用 AJAX 的方式來呼叫 Open API，而不是不斷的 Refreshing 頁面。

另一個重點是，使用 NodeJS 與 Data Push 方式，才能實作真正的 Real-time Web Application。AJAX 搭配 Refresh 方式，顯然已經不合適了。

Device API 的革命時代

HTML5 應用程式大致可分為：Web App 以及 Mobile App。HTML5 本來就是「Web Application」的概念，這是它的核心思想。當然，在手機上使用瀏覽器也能執行（瀏覽）Web 應用程式；開發者可以將瀏覽器做適度的封裝，例如：去除網址列，這會讓使用者「好像在使用 Native 應用程式」一樣。沒錯，HTML5 在 Mobile 應用程式的價值，就是提供與 Native 應用程式「相同」的體驗。

不過，Web App 與 Mobile App 的概念已經糊模化了。事實上，Web App 可以很容易封裝成 Mobile App：只要有好的 Responsive Design 與 Web Service 設計。

要讓 HTML5 Mobile 應用程式，在手機上表現出 Native 應用程式的體驗，有一個很重要的條件，就是：HTML5 能存取 Device API。更簡單一點，要想辦法使 JavaScript 呼叫底層的手機 API。例如，在 Android 手機上，透過 JavaScript 去呼叫作業系統的 Java API，即 Device API。這是頭一遭，能使用「做網頁的方式」來做出「手機應用程式」。使用 JavaScript 呼叫 Device API 的技術，幾年前出現了，它叫做 PhoneGap。

PhoneGap 技術的概念，就是填補瀏覽器與手機間的隔閡 (Gap)。

這一定是一場革命，如果能用做網頁的方式來做出手機應用程式，代表不久的未來，設計師也能輕易開發手機應用程式。做技術的人來做應用程式，和設計師來做應用程式，不需要多加思考，也能知道設計師比技術人員更適合做出絕佳的應用程式，因為這是 UI 與 UX 當道的時代。

HTML5+JavaScript+PhoneGap 讓這場革命成形了。Adobe Systems 公司看出了 PhoneGap 的未來價值，所以它在 2011 年

底，收購了 PhoneGap 的開發商，並且將 PhoneGap 整合至 DreamWaver 軟體裡。從 DreamWaver 5.5 開始，用 HTML5 製作手機應用程式，變得簡單不過了。

Native Mobile App 設計模式

使用 HTML5 來製作 Native Mobile App 的重點在於：JavaScript 如何呼叫 Native API。Native API 也稱為 Device API，這是由裝置本身的作業系統所提供的 API。

從技術的角度來看，採取了一種 DOM 與 Device API 連結 (Binding) 的方式，來達到上述目的。

接下來以 Android 作業系統為例，說明 PhoneGap 的實作原理。Android 作業系統內建的瀏覽器，採用的 HTML 引擎稱為 Webkit，這是由蘋果公司 (Apple Inc) 所貢獻的開源項目。Android 收錄了 Webkit 做為主要的 HTML 與 JavaScript 引擎，並且將 Webkit 封裝成 WebView 元件，整合至 Android 框架內部。目前，Google 已經開發出自己的 JavaScript 引擎，稱為 V8。

WebView 元件提供一個稱之為 `addJavaScriptInterface()` 的 API，這個 API 就是 PhoneGap 的核心

DOM (Document-Object Model) 是 JavaScript 存取 HTML 文件的物件模型，簡單來說，透過 DOM，JavaScript 便能存取網頁裡的物件。`addJavaScriptInterface()` 的原理，就是在 HTML 的 DOM 裡加入一個新物件，並且讓這個物件「Binding」到 Java 程式碼裡。

意思是說，這裡有一個連接 DOM 與 Java 層的物件，JavaScript 透過這個物件，就可以存取到 Java 層。`addJavaScriptInterface()` 是 DOM-to-Java 的 binding 實作。以下是 `addJavaScriptInterface()` 的實作範例：

同樣的原理也可以應用在其它作業系統，例如：iOS、BlackBerry、Bada、Symbian、Window Mobile 等。所以，使用 PhoneGap 來開發手機應用程式，可以很輕易達到「Write once,

run anywhere」的目標。「寫一次、到處都能執行」是 PhoneGap 的主要價值，因為應用程式是使用 HTML5+JavaScript 來製作，在任何有搭載瀏覽器與 PhoneGap 的裝置上都能執行，並不受制於作業系統。

Android 作業系統採取 Callback object 的方式來實作「DOM-to-Java」，也就是，Java 應用程式裡的 Method，都是被 HTML5 裡的 JavaScript 程式碼所呼叫。所以在實作 Callback method 時，需要注意幾個重點：

- 避免 Long operation
- 避免 Blocking operation
- 避免使用 Java thread 來操作 Callback object

Android 應用程式是單線程（Single thread）模式，這個單一的線程稱為 Main thread（主線程），並且是由 Android DalvikVM 自動產生的。由應用程式自行產生的線程，則稱為 Worker thread。所以，如果在 Main thread 與 Worker thread 裡同時去操作相同的的實例（Instance），例如：Method call，這個實例就必須有能力處理重覆進入（Re-entrancy）的問題。

能處理重覆進入的 Method，也被稱為 Thread-safe method。簡單來說，不是 Thread-safe 的 Method，就只能在 Main thread 裡呼叫使用，不能由 Worker thread 來呼叫。在 Android 作業系統裡，與 UI 有關的 API，都不是 Thread-safe method；意思是說，透過 Worker thread 去操作 UI，並不是正確的做法。

將這個觀念呼應到上述的 Callback object。當 JavaScript 呼叫 Callback method 時，這個 method 如果不是 Thread-safe 的實作，就不能有其它的 Worker thread 來呼叫它。概念上，Main thread 與 Worker thread 是併行（Concurrent）的關係，所以這個現象是可能發生的。

Open Web Devices 計畫

PhoneGap 提供許多「API」給 JavaScript 開發者使用，透過這些 API，JavaScript 就可以呼叫 Device API。所以，這些 JavaScript

API 就要有標準，這是很容易了解的概念。PhoneGap 就自己定義了 API 標準。

還有另外一個開放源碼的作業系統，也擁有與 PhoneGap 一樣的概念與技術，這個作業系統就是 B2G(Boot-to-Gecko)。B2G 也要一套這種 API 標準。然而，B2G 並不採用 PhoneGap 的 API 標準，而是結盟電信營運商，成立一個開放標準計畫，來定義 API 標準。這個計畫就是 Open Web Devices(OWD)。

使用 PhoneGap 或追隨 OWD 計畫，從商業面來看，並不是一個好主意。不過是 PhoneGap 或 OWD，多少都隱含了一些商業企圖，以及競爭策略。身為設備製造商，「Build My Own」也是我的聰明選擇。

以 Android 作業系統為例，要做出 PhoneGap 技術，並不是很難，事實上，對熟悉 Android 技術的開發者來說，這是沒有技術難度的工作。現在，為大家解密 PhoneGap 的技術原理。

重要的資訊交換格式：JSON

筆者借用了 PaaS 行銷名詞，來介紹一個技術：JSON。PaaS 是一個很棒的概念，這就是一種雲端技術。然而，傳統做法，Client 端得到的是一個 HTML5 文件，而不是格式化後的資料(Formatted)，這是一個缺點。

如果 Server 回傳的是格式化後的資料，Client 端就可以更有效率地利用這些資料。試想，如果 Google 傳回的搜尋結果是一堆 HTML5，那我們不就還要再去 Parse 這份文件，才能取出真正的結果，然後才能再次利用這些資料（例如：再儲存為 CVS 格式）。

為了解決這個問題，必須要有一個標準，不能大家都用自己的 HTML5 文件，或是自定的格式。軟體工程師設計了一些標準。一開始提出的做法，是制定標準的 XML 標籤，這樣大家就可以統一文件格式了。但是還有一個問題，就是「資料量太大」。

試想，Server 要回傳二筆資料，這二筆資料都是電話號碼：

- 0911-111-111
- 0900-000-000

然後用 XML 來表示，就變成：

```
1 <Telephone>
2   <Item>0911-111-111</Item>
3   <Item>0900-000-000</Item>
4 </Telephone>
```

這種把資料打腫了才回傳的做法，大大浪費網路頻寬。上面只是一個簡單的例子，現實環境，要回傳的資料可能是一個 10MB 的 XML 文件，結果原始資料可能只有 1MB。這就像，一個好端端的人，硬是被塞進了 9 公斤的肥肉。

要解決這個問題，就要有一個輕量級（Light-weight）的資料交換格式，這項技術就是 JSON。所以，JSON 是 Client/Server 交換資料的一種格式，一種 Light-weight data exchange 技術。

還有一些不可不知的 HTML5 技術：

- PhoneGap: Device API 的標準，使用 JavaScript 呼叫 Device API 的好技術，Nitobi 公司是 PhoneGap 的開發商，這家公司現已被 Adobe Systems 收購
- WebSocket: HTML5 標準裡的一個技術
- NodeJS: 開發專用 Web Service 的技術，採用 JavaScript 語言。專用 Web 服務器是 Cloud 的重要領域。Apache Web Server 是通用型 Web Server，Cloud 需要專用的 Web Service

現在開發專用的 Web Service 非常重要，這是 PaaS 的靈魂。例如，開發股票報價專用 Web Server。過去常聽到的 Web Server，例如：Apache，都是一般用途的 Web Server，用來「host webpages」。

現在 Client 端的網頁是用 JavaScript，Server 端的開發也可以用 JavaScript，Client/Server 通通都用 JavaScript，這是一個「All in JavaScript」的時代。

結論

根據截至目前為止的介紹，可以歸納出二個 HTML5 的重要設計模式如下：

- Device-Server 設計模式 (Data push)
- Native Mobile App 設計模式 (DOM-to-Java bindings)

Node.js 入門 - URL Routing 篇

HTML5 的學習，可分為二個層面：

- Frontend: 即 Client (Device) 端的開發
- Backend: 即 Server (Cloud) 的開發

這一章將從 Backend 開始整理重要的入門技術。通常 HTML5 的學習是由 Frontend 開始，而且是從最根本的 HTML5 標籤或 CSS 語法開始介紹。不過，從 Backend 的角度去看 Frontend，可以讓我們重新認識許多常見的 Frontend 技術。

筆者並不建議初學者，從 HTML5 標籤與 CSS 語法的角度來學習。現在的時空背景，與過去開發 Web 所使用的軟體技術已經有很大的差異了；因此，過去學習 Web 的觀念與方式，不太能適用於現在的 HTML5 軟體開發。

觀看 Nodejs 線上課程

這個章節將介紹 Node.js 技術，這是一個在 HTML5 領域裡非常重要的技術。Node.js 並不是使用於 Client-side，如同先前所做的介紹，它使用於 Server-side。

接下來將會大家介紹的是 Node.js 的精神與技術哲學。本章節的寫作原則是著重「軟體思考」，關於技術上的細節不多加著墨。有關 Node.js 函數的說明，可參考 Node.js 官方網站。在繼續進行範例說明前，請先備妥這份文件：

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

另外，也請參考 Node.js 官方網站的說明，來安裝 Node.js 環境。關於環境安裝，以及 Node.js 的入門觀念，可參考由 MokoVersity 所提供的免費線上課程：

<http://www.mokoversity.com/course/html5/nodejs-overview>

取得上課範例

本章所撰寫的 Node.js 程式碼，皆可在 Github 上取得：

<https://github.com/jollen/html5-websocket-nodejs>

接下來，讓我們用一個連貫性的實例：即時通訊軟體，來為大家介紹 Node.js 技術。

第一個 Node.js 程式

我們提過了，Node.js 就是 Web Server。所以，不免俗地先了解“Hello, World”的寫法：

01-create-server/hello.js

```
1 1 var http = require('http');
2 2
3 3 var httpServer = http.createServer(function (req, re\
4 s) {
5 4   res.writeHead(200, {'Content-Type': 'text/html'});
6 5   res.end('<h1>Hello World</h1>\n');
7 6 });
8 7
9 8 httpServer.listen(8080);
10 9
11 10 console.log('Server running at http://127.0.0.1:8080\
12 /');
```

程式碼第 3 行的地方，呼叫 http 模組的 `createServer()` 函數來建立一個 Web Server 物件。建立 Web Server 物件，並且啟動一個 Web Server，是 Node.js 技術的第一個步驟。

`createServer()` 函數的說明如下：

```
1 http.createServer([requestListener])
```

`createServer()` 執行成功後傳回 Web Server 物件，參數 `requestListener` 是一個 Request Handler Function，用來處理 `request` 事件。關於 Node.js 的事件處理技術，後續再做說明。當 `request` 事件發生時，Request Handler Function 將被 Callback，並帶有二個參數：

- `req`: `http.ServerRequest` 的實例化 (instance)
- `res`: `http.ServerResponse` 的實例化

將上述的範例，儲存為 `hello.js`，並且利用 `node` 指令執行：

```
$ node hello.js
```

安裝 Node.js 後，就可以取得 `node` 命令。

這是執行 Node.js 程式碼的陽春版做法，後續將導入 `forever` 工具，以進階的方式來執行 Node.js 程式。

Node.js 採用 Google 所開發的 V8 JavaScript 引擎，原本 V8 引擎是設計給瀏覽器使用的 JavaScript 引擎，現在有開發者把它抽離出來，變成一個獨立的直譯器，讓 JavaScript 程式碼升格為 Server-Side Script。

我們利用瀏覽器連到 `http://127.0.0.1:1234/`，結果如圖 3.1。

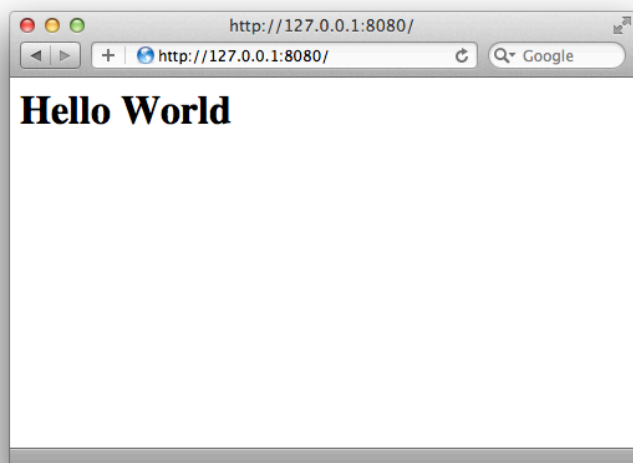


圖 3.1: `hello.js` 執行結果

V8 JavaScript 引擎介紹

JavaScript 引擎將成為手持裝置的重要技術。早期的 Android 系統，使用的 JavaScript 引擎稱為 JavaScriptCore (JSC)，這是由 Apple 所開發的 JavaScript 引擎，並且包含在 Webkit 中。因為一些原因，Google 也決定開發自己的 JavaScript 引擎，稱之為 V8。

技術上，JSC 與 V8 的設計理念不同，一般相信，新一代的 V8 引擎效能比 JSC 引擎更好。Android 2.3 加入了 V8 引擎，若想使用最近的 V8 引擎，就要使用 Android 2.3 以上的版本。

為什麼要使用 Node.js？

到這裡，大家可能會有一個疑問。為什麼不使用現有的 Web Server 來開發 Web Service 就好，例如使用 Apache。非要使用

Node.js 技術不可嗎？這個問題的答案，要從 Thread Model 說起。

典型的 Web Server 以 Multi-thread 架構來實作「Concurrency」，也就是以建立 Thread 的方式，來處理處理事件（Events）。但是過多的 Thread 會造成伺服器的負擔：

- 假設一個 Thread 可以處理 10 個事件
- 同時處理 1000 個事件，就必須建立 100 個 Thread
- 大量的 Thread 在分時作業系統裡，會造成 Context-Switch Overhead，讓每一個 Thread 處理事件的時間拉長，形成效能低落的現象

由此可知：

- 若是將 Multi-thread 架構，應用在「處理巨量的同時連線請求」上，伺服器的負擔就會很大
- Multi-thread 架構的軟體，可能在系統產生過多的 Thread，過多的 Thread 除造成伺服器的負載增加外，也需要大量的記憶體

是否有替代方案呢？將上述的 Multi-thread 架構，改為 Event Loop 架構即可。Node.js 的訴求之一就是：採用 Event Loop 架構。此外，Node.js 採用 JavaScript 程式語言，JavaScript 本身也有一些很好的語言特性：

- 具備 Lambda 運算子，大量使用匿名函數（anonymous function）與 Closure（封閉性）觀念
- 使用 Callback Object 做為函數的參數（Lambda），易於處理 Non-blocking Operation 與 Event Handling

Node.js 本身的 I/O 操作，也大多是（幾乎）Non-blocking 的機制。這點與 PHP 有很大的不同。這樣的機制，對於消化巨量的連線請求，有非常大的幫助。

製作 Node.js 模組

學習 Node.js 的第一件事情，就是了解如何將程式碼模組化，簡單來說，就是製作一個程式庫 Node.js 的模組隱含著 Closure 的特性，這與第一章介紹的觀念相同（但做法不同）。

JavaScript 比較講求模組化，所以我們繼續重構 hello.js。先將 Web Server 的部份獨立成一個模組，程式碼規劃如下：

- index.js: 主程式
- server.js: 啟動 Web server 的模組

index.js 的完整程式碼如下：

02-modules/hello.js

```
1 var server = require("./server");
2
3 server.start();
```

主程式的部份，以 require() 函數將 server 模組（即 server.js 檔案）引入，接著呼叫模組裡的 start() 函數。server.js 完整程式碼如下：

02-modules/server.js

```
1 1 var http = require("http");
2 2
3 3 function start() {
4 4   function onRequest(request, response) {
5 5     console.log("Request for " + pathname + " received.");
6 6   }
7 6
8 7   response.writeHead(200, {"Content-Type": "text/plain"});
9 7 }
```

```
10 8    response.write("Hello World");
11 9    response.end();
12 10   }
13 11
14 12   http.createServer(onRequest).listen(8080);
15 13   console.log("Server has started.");
16 14   }
17 15
18 16   // Export functions
19 17   exports.start = start;
```

程式碼第 3 行到第 16 行的地方，我們實作了一個函數，並且將它匯出。請特別留意，沒有匯出的函數，將不是 Public，它不能被外部的人呼叫。exports 是 Node.js 的一個 Global object，用來讓我們匯出模組裡的函數，成為 Public Function。

目前為止，我們發現了一些觀念：

- Frontend 與 Backend 都使用 JavaScript 做為主要的程式語言
- Frontend 與 Backend 都要模組化，並引入 Closure 觀念
- Frontend 與 Backend 的 Module / Closure，相念相通，實作方式不同

Chaining Pattern

另外，server.js 裡也做了一些改寫。程式碼第 4 行的地方，以具名函數的方式重新實作，目的是讓程式碼更具可維護性。此外，程式碼第 14 行的地方：

```
1 http.createServer(onRequest).listen(8080);
```

物件接著下一個物件來連續呼叫多個方法的寫法，就叫 Chaining Pattern（鏈接模式）。這個設計模式的目的，同樣

是為了提升程式碼的可維護性：不但能簡化程式碼，更能讓程式碼能構成一個句子。

在接下來的範例裡，我們將善用具名函數以及 Chaining Pattern 來提昇程式碼的可維護性。

URL Routing

這是處理 URL（HTTP Request）與 Query String 的核心觀念，這是利用 Node.js 開發 Web Service 的重要步驟。讓我們先來了解 Routing 的寫法，再來探討它的觀念。

首先，先改寫 server.js 模組如下：

03-route/server.js

```
1 1 var http = require("http");
2 2 var url = require("url");
3 3
4 4 function start(route) {
5 5   function onRequest(request, response) {
6 6     var pathname = url.parse(request.url).pathname;
7 7     console.log("Request for " + pathname + " received.");
8 8     console.log("Request url: " + request.url);
9 9
10 10    route(pathname);
11 11
12 12    response.writeHead(200, {"Content-Type": "text/plain"});
13 13    response.write("Hello World");
14 14    response.end();
15 15   }
16 16
17 17   http.createServer(onRequest).listen(8888);
18 18   console.log("Server has started.");
```

```
21 19 }  
22 20  
23 21 // Export functions  
24 22 exports.start = start;
```

Routing 觀念的主要用途是處理 URL，所以我們利用 url 模組來取出 URL 裡的 `pathname`，並將 `pathname` 交給 `route()` 函數來處理。這裡很特別的地方是，`start()` 函數裡所呼叫的 `route()` 函數，是透過參數列傳遞進來的，這和模組的 Closure 特性有關係，觀念說明如下：

- `route()` 函數實作在 `router` 模組，而不是 `server.js` 模組
- 目的是將 Routing 的功能，拆成單獨的模組來維護
- `route()` 函數由 `router.js` 模組提供，必須引用 `router` 模組

在這裡範例裡，我們由 `index.js` 引入 `router.js` 模組，並且將裡頭的 `router()` 函數，透過參數列交給 `start()` 函數。如此一來，`start()` 也可以呼叫到 `route()` 函數了。

這個部份，可以選擇另外一個實作方法：在 `server.js` 裡引用 `router.js` 模組。不過，就概念上來說，範例的實作方式好一些。原因如下。

- Decomposition: 將 `router.js` 與 `server.js` 模組的相依性解除
- Component-based software engineering: 將 `router.js` 與 `server.js` 做成獨立的模組，他們之間如果沒有相依性，就可以做為二個不同的模組來使用。例如，將 `router.js` 模組抽換成其它專案的 Routing 模組，並且 `server.js` 可以重用。Node.js 的軟體架構，主軸是模組化，即 Component-based 軟體工程的觀念

目前透過 `npm` 指令，不但可以安裝到各式不同的 Node.js 模組，甚致可以將自己的模組出版 (Publish) 給其他開發者使用。

Node.js 的事件處理機制，採用典型的 Callback Functions 做法。

接著，要開始處理 Pathname 與 Query String 的解析，請先參考圖 2.2。

改寫 index.js 主程式如下：

03-route/index.js

```
1 1 var server = require("./server");
2 2 var router = require("./router");
3 3
4 4 server.start(router.route); // 傳遞 route 物件
```

將 Routing 的演算法製作成獨立的模組，並將 router() 函數傳遞給 start()，函數的參數，可以傳遞一個函數，這個觀念就是 Lambda。router.js 完整程式碼如下：

03-route/router.js

```
1 1 function route(pathname) {
2 2     console.log("Route this request: " + pathname);
3 3 }
4 4
5 5 exports.route = route;
```

請注意，這個範例雖然陽春，但是展示了一個非常重要的觀念：

- 函數就是物件，所以我們把 route 物件交給 start() 函數，讓 start() 函數去使用物件
- 直接在 start() 裡呼叫 route() 函數也可以，為什麼不這樣做？因為這不是 JavaScript 的觀念，倒是有點像是標準 C 語言呼叫函數的觀念，同時也會降低程式碼的可維護性

接下來，要讓 `route()` 解析 `pathname`。例如，我們定義了二個 API:

- `http://localhost:8080/start`，用來連接伺服器並接收即時訊息
- `http://localhost:8080/send`，送出文字訊息

分別要處理二個 `pathname` 如下:

- `/start`，呼叫專屬的 Handler `'start()'` 來處理
- `/send`，呼叫專屬的 Handler `'send()'` 來處理

實作的關鍵來了，我們要利用 Request Handler 的觀念來實作，首先，修改 `index.js` 如下:

04-request-handlers/index.js

```
1 1 var server = require("./server");
2 2 var router = require("./router");
3 3 var handlers = require("./requestHandlers");
4 4
5 5 // 使用 Object 來對應 pathname 與 request handlers
6 6 var req = {
7 7   "/": handlers.start,
8 8   "/start": handlers.start,
9 9   "/send": handlers.send
10 10 };
11 11
12 12 // 傳遞 request handler
13 13 server.start(router.route, req);
```

上述的二個 Handler 函數: `start()` 與 `send()` 將另行實作於 `requestHandlers` 模組。`requestHandlers` 模組匯出 `start()` 與 `send()` 函數，分別處理相對應的 `pathname`。

因此，主程式在第 6 行到第 10 行的地方，利用 *req* 物件來對應這個關係。在呼叫 *start()* 時，將 *req* 物件傳入。

另外，JavaScript 雖然不是物件導向式語言，但仍要以物件的觀念來撰寫。所以，我們將 *req* 以 *var* 語法定義成 *object*。很多時候，或許也能以 *associative array* 來實作，但並不是很建議。

以下就是一個以 *associative array* 的實作範例，原則上不推薦：

```
1  1 var server = require("./server");
2  2 var router = require("./router");
3  3 var handlers = require("./requestHandlers");
4  4
5  5 // 使用 associative array 來對應 pathname 與 request handlers
6  6
7  6 var req = {};
8  7
9  8 req["/"] = handlers.start;
10 9 req["/start"] = handlers.start;
11 10 req["/send"] = handlers.upload;
12 11
13 12 // 傳遞 request handler
14 13 server.start(router.route, req);
```

修改後的 *router.js* 如下：

04-request-handlers/router.js

```
1 1 function route(pathname, handlers, response) {
2 2     console.log("Route this request: '" + pathname + \
3 3     "'");
4 3
5 4     // 檢查 pathname 是否有對應的 request handlers
6 5     if (typeof handlers[pathname] == "function") {
7 6         handlers[pathname](response);
8 7     } else {
9 8         console.log("No request handler for this pat\
10 hname: '" + pathname + "'");
11 9     }
12 10 }
13 11
14 12 exports.route = route;
```

再次修改 server.js 如下:

04-request-handlers/server.js

```
1 1 var http = require("http");
2 2 var url = require("url");
3 3
4 4 function start(route, handlers) {
5 5     function onRequest(request, response) {
6 6         var pathname = url.parse(request.url).pathname;
7 7         console.log("Request for " + pathname + " receiv\
8 ed.");
9 8
10 9         route(pathname, handlers, response);
11 10
12 11         response.writeHead(200, {"Content-Type": "text/p\
13 lain"});
14 12         response.write("Hello World");
```

```
15 13     response.end();
16 14   }
17 15
18 16   http.createServer(onRequest).listen(8080);
19 17   console.log("Server has started.");
20 18 }
21 19
22 20 // Export functions
23 21 exports.start = start;
```

最重要的模組：requestHandlers.js，完整程式碼如下：

04-request-handlers/requestHandlers.js

```
1 1 function start(response) {
2 2     console.log("Handler 'start' is started.");
3 3 }
4 4
5 5 function send(response) {
6 6     console.log("Handler 'send' is started.");
7 7 }
8 8
9 9 exports.start = start;
10 10 exports.send = send;
```

到這裡，已經完成了一份很基本的 Web Service 實作。接下來，我們要將這個成果發展成一個即時聊天軟體，就命名為 NoChat。NoChat 將會是一個完全使用 HTML5 技術開發的即時聊天軟體。

設計 HTTP API

完整的 NoChat 分為二個部份：

- Backend (Server-side) 將基於目前的 Node.js 程式碼繼續完善
- Frontend (Client-side) 手機端的 App 將以 HTML5 + PhoneGap 來製作

NoChat 提供二個 API，現在將 API 詳細定義如下。

- /start，建立與 Client 的 WebSocket 連線
- /send，送出訊息。

‘/send’ API 的 Query String 參數定義如表 3-1。這部份在第 2 章已做過說明。

表 3-1 API 的參數

參數	值	用途說明
m	‘hello’	指定要傳送的訊息 (message)
u	‘jollen’	指定 Username

測試案例

以下設計一個簡單的測試案例，在完成第一個 NoChat 的 Prototype 後，將以下列步驟進行測試：

1. 在 localhost 啟動 Node.js
2. 打開 client.html 聊天網頁
3. client.html 呼叫 API: `http://localhost:8080/start`, Server 回傳 “OK” 訊息
4. client.html 與 Server 建立 WebSocket 連線
5. client.html 開始接收 Node.js 推送 (Data Push) 的即時訊息

傳送訊息給 Node.js 的測試步驟：

1. 開啓一個新的瀏覽器視窗
2. 使用瀏覽器呼叫 API: `http://localhost:8080/send?m=hello`
3. Node.js 收到訊息，並透過 WebSocket 將訊息 Push 給所有的用戶端

這還不算是一個真正的使用案例 (Use Case)，但至少可以幫助我們實作出第一個 Prototype。

關於 Web Service

圖 2.2 是大家所熟悉的 HTTP API 形式。許多網站，像是：Google、Facebook 等，都有開放 HTTP API 供開發者存取它們的服務。以 NoChat 來說，透過上述二個 API 可以向 Server 請求服務。因此，Node.js 的重心，就是在發展 Web Service。

Web Service 的 API 定義，未來將重構為 REST 標準。基於 HTTP 的 Web Service API，是目前為止，我們所學到的重要觀念。

此外，呼叫 HTTP API 的方式，可使用 GET 與 POST 二種 HTTP 方式 (HTTP Method)，這二種方式都是定義在 HTTP 裡的標準。REST 標準，也引用了其它的 HTTP Method。

目前，NoChat 仍暫時以 Query String 的方式來傳遞參數。

解析 Query String

Client 端呼叫 Server 所提供的 Web Service API。所以，現在的關鍵是如何解析 Query String。如圖 2.2，Node.js 使用 `querystring` 模組來解析 Query String。先將 `querystring` 模組匯入，接著呼叫 `parse()` 函數：

```
1 var querystring = require('querystring');
2 var parsedstring = querystring.parse("m=hello&u=jollen" \
3 );
```

解析後的結果存放於 `parsedstring` 物件，回傳結果：

```
1 { m: 'hello', u: 'jollen' }
```

`parse()` 函數有三個參數：

```
1 querystring.parse(str, [sep], [eq])
```

- `str` 是 Query String
- `sep` 是「Separator」，也就是字串的分隔字元，預設是「`$`」，通常不做變數
- `eq` 則是字串與值的對應字元，預設是「`=`」，通常不做變數

了解如何解析 Query String 後，就可以開始進行後續的工程了。再次修改 `requestHandlers.js`，如下：

05-query-string/requestHandlers.js

```
1 1 var querystring = require('querystring');
2 2
3 3 /**
4 4  * Global variables
5 5  */
6 6 var history = [ ];
7 7
8 8 function start(response, query) {
9 9     console.log("Handler 'start' is started.");
10 10    console.log("Query string is: " + query);
11 11 }
```

```
12 12
13 13 function send(response, query) {
14 14     console.log("Handler 'send' is started.");
15 15     console.log("Query string is: " + query);
16 16
17 17     var parsedstring = querystring.parse(query);
18 18
19 19     var obj = {
20 20         message: parsedstring.m,
21 21         username: parsedstring.u,
22 22         timestamp: (new Date()).getTime()
23 23     };
24 24
25 25     history.push(obj);
26 26
27 27     ////////// DEBUG //////////
28 28     for (var i = 0; i < history.length; i++) {
29 29         console.log("[ "+i+" ]: " + history[i].message\
30 30     );
31 31     }
32 32 }
33 32
34 33 exports.start = start;
35 34 exports.send = send;
```

這裡利用一個全域陣列 *history* 來儲存訊息。將收到的訊息封裝成物件後，再使用標準的陣列操作將物件放到陣列裡。另外，我們也將一個時間記號（Timestamp）一併封裝至該物件，用來紀錄接收到訊息的時間。

結論

到這裡完成了 Node.js 入門的學習：

- 學會撰寫第一個 Node.js 程式
- 學會啟動 HTTP Server
- 了解並實作 URL Routing
- 學會解析 Pathname 與 Query String

Node.js 入門 - WebSocket 與 JSON 篇

WebSocket 是 HTML5 裡的一個標準，它是一種 TCP/IP 的連線技術。在協定部份，則是基於 HTTP (Over HTTP) 協定。因此，WebSocket 標準定義了一些 HTTP Headers 來進行 Client/Server 的通訊。

WebSocket 分為 Client 端與 Server 端二個部份，本章要介紹的是利用 Node.js 技術，來開發 WebSocket Server。目前有許多現成的 Node.js WebSocket 模組可使用，實作時，我們就不必自行處理複雜的 WebSocket 協定問題。

本章所使用的 WebSocket 模組，要使用 npm 工具另外安裝。利用 npm 安裝 WebSocket-Node:

```
$ npm install websocket
```

WebSocket-Node 原始碼可由 Github 上取得:

<https://github.com/Worlize/WebSocket-Node>

安裝，依據需要，可引入不同的模組。WebSocket-Node 提供 4 個模組如下:

- `var WebSocketServer = require('websocket').server;`
- `var WebSocketClient = require('websocket').client;`
- `var WebSocketFrame = require('websocket').frame;`
- `var WebSocketRouter = require('websocket').router;`

NoChat 範例，將會使用 'server' 模組。

建立 WebSocket Server

基於第 3 章最新的範例，繼續修改 server.js 的程式碼如下：

06-websocket-with-protocol/server.js

```
1  1 var http = require("http");
2  2 var url = require("url");
3  3 var WebSocketServer = require('websocket').server;
4  4
5  5 function start(route, handlers) {
6  6   function onRequest(request, response) {
7  7     var pathname = url.parse(request.url).pathname;
8  8     var query = url.parse(request.url).query;
9  9
10 10    console.log("Request for " + pathname + " received.");
11 11
12 12    route(pathname, handlers, response, query);
13 13
14 14    response.writeHead(200, {"Content-Type": "text/plain"});
15 15    response.write("Hello World");
16 16    response.end();
17 17   }
18 18
19 19   var server = http.createServer(onRequest).listen(8080, function() {
20 20     console.log("Server has started and is listening on port 8080.");
21 21   });
22 22
23 23   wsServer = new WebSocketServer({
24 24     httpServer: server,
25 25     autoAcceptConnections: false
```

```
30 26   });
31 27
32 28   function onWsConnMessage(message) {
33 29       if (message.type == 'utf8') {
34 30           console.log('Received message: ' + message.utf\
35 8Data);
36 31       } else if (message.type == 'binary') {
37 32           console.log('Received binary data.');
```

```
38 33       }
39 34   }
40 35
41 36   function onWsConnClose(reasonCode, description) {
42 37       console.log(' Peer disconnected with reason: ' +\
43 38   reasonCode);
44 38   }
45 39
46 40   function onWsRequest(request) {
47 41       var connection = request.accept('echo-protocol',\
48 42   request.origin);
49 42       console.log("WebSocket connection accepted.");
50 43
51 44       connection.on('message', onWsConnMessage);
52 45       connection.on('close', onWsConnClose);
53 46   }
54 47
55 48   wsServer.on('request', onWsRequest);
56 49 }
57 50
58 51 // Export functions
59 52 exports.start = start;
```

先將 WebSocket-Node 的 'server' 匯入，如程式碼第 3 行。其它的修改細節條列如下：

- 第 19~26 行：將原本的 HTTP Server 物件，聚合至（傳遞）WebSocket Server。WebSocker Server 的物件名稱為 wsServer
- 第 48 行：在 wsServer 物件裡註冊一個 Request Handler，即 onWsRequest() 函數
- 第 40 行：當 WebSocket 的連線請求發生時，便回呼此函數
- 第 41 行：接受該 WebSocket 連線，第一個參數是 WebSocket Protocol，這是一個自定的協定名稱，用途由開發者定義
- 第 44~45 行：為此連線註冊 Message Handler 與 Close Handler 函數
- 第 28 行：收到用戶端傳送過來的訊息時，回呼此函數，後續將繼續擴充此函數，將收到的訊息儲存，並將訊息即時（Real-time）推送（Push）到所有的用戶端
- 第 36 行：該 WebSocket 連線關閉後，回呼此函數

儲存用戶端 **WebSocket** 連線

要儲存所有的用戶端 WebSocket 連線，最簡便的方式是使用 Global Array:

```
1 // Connected WebSocket clients
2 var clients = [];
```

當用戶端與 Node.js 建立連線時，將會回呼上述提及的 onWsRequest() 函數。所以，儲存連線的程式碼，應該添加至這個地方。繼續加入程式碼如下：

```
1  function onWsRequest(request) {
2    var connection = request.accept('echo-protocol', request.
3    est.origin);
4    console.log("WebSocket connection accepted.");
5
6    // Save clients (unlimited clients)
7    clients.push(connection);
8
9    connection.on('message', onWsConnMessage);
10   connection.on('close', onWsConnClose);
11 }
```

以下目前為止的最新程式碼。

07-websocket-data-push/server.js

```
1  1 var http = require("http");
2  2 var url = require("url");
3  3 var WebSocketServer = require('websocket').server;
4  4
5  5 // Connected WebSocket clients
6  6 var clients = [];
7  7
8  8 function start(route, handlers) {
9    9 function onRequest(request, response) {
10   10 var pathname = url.parse(request.url).pathname;
11   11 var query = url.parse(request.url).query;
12   12
13   13 console.log("Request for " + pathname + " received.");
14   14
15   15 route(pathname, handlers, response, query, clients);
16   16
17   17 response.writeHead(200, {"Content-Type": "text/p\
```

```
20   lain"}});
21   18     response.write("Hello World");
22   19     response.end();
23   20   }
24   21
25   22   var server = http.createServer(onRequest).listen(8\
26   080, function() {
27   23     console.log("Server has started and is listenin\
28   g on port 8080.");
29   24   });
30   25
31   26   wsServer = new WebSocketServer({
32   27     httpServer: server,
33   28     autoAcceptConnections: false
34   29   });
35   30
36   31   function onWsConnMessage(message) {
37   32     if (message.type == 'utf8') {
38   33       console.log('Received message: ' + message.utf\
39   8Data);
40   34     } else if (message.type == 'binary') {
41   35       console.log('Received binary data.');
```

```
42   36     }
43   37   }
44   38
45   39   function onWsConnClose(reasonCode, description) {
46   40     console.log('Peer disconnected with reason: ' + \
47   reasonCode);
48   41   }
49   42
50   43   function onWsRequest(request) {
51   44     var connection = request.accept('echo-protocol', \
52   request.origin);
53   45     console.log("WebSocket connection accepted.");
54   46
```

```
55 47    // Save clients (unlimited clients)
56 48    clients.push(connection);
57 49
58 50    connection.on('message', onWsConnMessage);
59 51    connection.on('close', onWsConnClose);
60 52  }
61 53
62 54    wsServer.on('request', onWsRequest);
63 55  }
64 56
65 57 // Export functions
66 58 exports.start = start;
```

學習 JSON 格式

第 1 章介紹了 JSON 的主要精神，現在要來學習 JSON 表示方法。

在 NoChat 範例裡，Server 要把收到的訊息，Push 給所有的 Client 端。Server 與 Client 所使用的標準資料交換格式，就是 JSON。如何把訊息打包成 JSON 格式？方式非常簡單。以表 4-1 為例，要將這個資料表，撰寫為 JSON 格式，只需要二個步驟：

Step 1: 以 JavaScript 物件表示一筆資料

例如，第一筆個人資料，以 JavaScript 物件來表示的話，只要用 *var* 來宣告此物件即可：

```
1 var obj = {
2   "name": "Jollen",
3   "score": 80
4 };
```

大括號是 JavaScript 表示物件的語法。上述範例，我們宣告了 *obj* 物件。

Step 2: 轉換成標準 JSON 語法

去掉等號，以及等號左邊的物件宣告，結果如下：

```
1 {
2   "name": "Jollen",
3   "score": 80
4 }
```

請注意，結尾的分號也要一併去除。上述的表示方法，就是標準的 JSON 語法。這個例子用 JSON 來表示一筆個人資料。JSON 表示方法非常地簡單，只要會 JavaScript 保證 1 分鐘即可上手，不需要特意學習。

Step 3: 用陣列表示多個物件

表 4-1 資料表

“name” 欄位	“score” 欄位	說明
“Jollen”	80	第 1 筆使用者資料
“Paul”	170	第 2 筆使用者資料
“Peter”	250	第 3 筆使用者資料
“Ellaine”	580	第 4 筆使用者資料

表 4-1 共有 4 筆個人資料，因此需要撰寫 4 個物件，每個物件之間用逗號隔開。試想，過去撰寫程式的經驗裡，我們用哪一個資料結構（Data Structure）來表示多筆型別（Data Type）相同的資料呢？答案是：陣列（Array）。

JavaScript 的陣列用中括號來宣告，例如：

```
1 var string = ['Jollen', 'Paul', 'Peter'];
```

這個例子宣告 *string* 陣列，裡頭有 3 個字串。用 JavaScript 怎麼表示 4 個物件的陣列呢？答案如下：

```
1 var persons = [  
2   {  
3     "name": "Jollen",  
4     "score": 80  
5   },  
6   {  
7     "name": "Paul",  
8     "score": 170  
9   },  
10  {  
11    "name": "Peter",  
12    "score": 250  
13  },  
14  {  
15    "name": "Ellaine",  
16    "score": 580  
17  }  
18 ];
```

我們只要把上述的 4 個物件，用陣列「群組」起來即可。和 Step 2 相同，保留以下的寫法即可：

```
1  [  
2    {  
3      "name": "Jollen",  
4      "score": 80  
5    },  
6    {  
7      "name": "Paul",  
8      "score": 170  
9    },  
10   {  
11     "name": "Peter",  
12     "score": 250  
13   },  
14   {  
15     "name": "Ellaine",  
16     "score": 580  
17   }  
18 ]
```

這就是表 4-1 的 JSON 表示方式了。將上述的 JSON 儲存為純文字，這個純文字檔就叫做 JSON Document，這就是 JSON Document 資料庫的概念。

同理，NoChat Server 收到訊息後，只要把訊息表示成 JSON 後，傳送給 Client 端即可。JSON 相當簡單易學，更是優秀的輕量級資料交換格式。

JSON Stringify

請注意，上述的 JSON 是一個型別 (Type)，是一個 Array Type。我們不能儲存或傳送「Type」，所以要將 Type 轉成字串 (String) 後，才能儲存或傳送。例如，對電腦來說，這是一個物件 (Object)：

```
1 { "name": "James" }
```

我們把這個物件轉成字串：

```
1 "{ \"name\": \"James\" }"
```

對電腦來說，這才是字串。所以，將 JSON 物件 (Object) 轉成字串後，才能儲存或傳送。這個動作就叫 JSON Stringify (字串化)。當然，字串化過的 JSON 字串，要使用時，也要解析 (Parse) 回物件。

在 Node.js 裡如何做 JSON Stringify 呢？只要呼叫 JSON.stringify() 函數即可。例如：

```
1 var obj = { "name": "James" };    // 一個物件
2 var str = JSON.stringify(obj);    // 字串化
```

下面是簡約的寫法：

```
1 var str = JSON.stringify({ "name": "James" });    // 字串化
```

沿續第 3 章的 NoChat 實作，將 requestHandler.js 加入 JSON Stringify 的處理。完整程式碼如下：

07-websocket-data-push/requestHandler.js

```
1 1 var querystring = require('querystring');
2 2
3 3 /**
4 4  * Global variables
5 5  */
6 6 var history = [ ];
7 7
8 8 function start(response, query, clients) {
```



```
9 9 console.log("Handler 'start' is started.");
10 10 console.log("Query string is: " + query);
11 11 }
12 12
13 13 function send(response, query, clients) {
14 14 console.log("Handler 'send' is started.");
15 15 console.log("Query string is: " + query);
16 16
17 17 var parsedstring = querystring.parse(query);
18 18
19 19 var obj = {
20 20 message: parsedstring.m,
21 21 username: parsedstring.u,
22 22 timestamp: (new Date()).getTime()
23 23 };
24 24
25 25 history.push(obj);
26 26
27 27 ////////// DEBUG //////////
28 28 for (var i = 0; i < history.length; i++) {
29 29 console.log("[ "+i+" ]: " + history[i].message\
30 );
31 30 }
32 31
33 32 var json = JSON.stringify({ type: 'message', dat\
34 a: obj });
35 33
36 34 // Data push to all clients
37 35 for (var i = 0; i < clients.length; i++) {
38 36 clients[i].sendUTF(json);
39 37 }
40 38 }
41 39
42 40 exports.start = start;
43 41 exports.send = send;
```

第 19 行到第 23 行，將訊息封裝到物件裡，同時也加入使用者名稱，以及時間戳記（Timestamp）。第 32 行將物件字串化，這就是一個標準的 JSON 資料了。接著，第 35 行到第 37 行，將這筆 JSON 資料，傳送給所有的 WebSocket Client 端。

製作 WebSocket 用戶端

現在是製作一個 Frontend 的時機了，透過一個簡單的 Frontend 來測試目前的實作成果。我們要 Frontend 的文件，命名為 client.html。實作 client.html 前，要瀏覽器必須支援 HTML5 的 WebSocket 標準。目前新版的瀏覽器都支援 WebSocket 標準。本書皆使用 Chrome 18+ 瀏覽器進行測試。

實作 client.html

現在，我們可以不考慮 HTML5 的相容性問題，假設使用者的瀏覽器都支援 WebSocket。不過，實作時，可以考慮加入一段程式碼，來檢查並提示使用者，目前的瀏覽器是否支援 WebSocket。

完整程式碼如下：

client/01-ws-open.html

```
1  1  <!DOCTYPE html>
2  2  <head>
3  3  </head>
4  4  <body>
5  5  <div id="header"></div>
6  6  <button onClick="WebSocketTest();">Get News</button>
7  7
8  8  <script type="text/javascript">
9  9  function WebSocketTest()
```

```
10 10 {
11 11   if ("WebSocket" in window)
12 12   {
13 13       // Let us open a web socket
14 14       var ws = new WebSocket("ws://svn.moko365.com:80\
15 80/", "echo-protocol");
16 15       ws.onopen = function(evt)
17 16       {
18 17           alert("open");
19 18       };
20 19       ws.onclose = function(evt)
21 20       {
22 21           alert("close");
23 22       };
24 23   }
25 24   else
26 25   {
27 26       // The browser doesn't support WebSocket
28 27       alert("WebSocket NOT supported by your Browser!\
29 ");
30 28   }
31 29 }
32 30 </script>
33 31 </body>
34 32 </html>
```

說明如下：

- 第 11 行：檢查瀏覽器是否支援 WebSocket
- 第 14 行：與伺服器建立 WebSocket 連線，第一個參數是伺服器位址，第二個參數是前面提及過的自定協定

當 WebSocket 建立成功後，瀏覽器會回呼 `onopen` 函數，即程式碼第 15 行。當 WebSocket 連線關閉後，則回呼 `onclose` 函

數。伺服器將訊息推送給瀏覽器時，瀏覽器會回呼 `onmessage` 函數。這個部份將在後續做說明。使用瀏覽器將 `01-ws-open.html` 文件打開後，可以看到一個按鈕，如圖 4.1。按下按鈕時，呼叫 `WebSocketTest()` 函數。



圖 4.1: 範例 `01-ws-open.html`

使用 jQuery 模式

第 1 章介紹的 jQuery 模式，在這裡派上用場了。

上述的例子雖然很直覺，不過還有一些缺點。第一件事情就是以 jQuery 模式來重構。將用戶端改寫如下：

client/02-ws-jquery-plugin-pattern.html

```
1  1  <!DOCTYPE html>
2  2  <head>
3  3  <script type='text/javascript' src='./jquery.min.js'\
4  ></script>
5  4  </head>
6  5  <body>
7  6  <div id="message"></div>
8  7
9  8  <script type="text/javascript">
10 9  $.fn.createWebSocket = function () {
11 10   if ("WebSocket" in window)
12 11   {
13 12       alert("WebSocket is supported by your Browser!"\
14 13   );
15 14       // Let us open a web socket
16 15       var ws = new WebSocket("ws://localhost:8080/sta\
17 16   rt");
18 17       ws.onopen = function()
19 18       {
20 19           // Web Socket is connected, send data using \
21 20   send()
22 21       ws.send("Message to send");
23 22   };
24 23   ws.onmessage = function (evt)
25 24   {
26 25       var received_msg = evt.data;
27 26       this.html(received_msg);
28 27   };
29 28   ws.onclose = function()
30 29   {
31 30       // websocket is closed.
32 31   };
33 32 }
```

```
34 30     else
35 31     {
36 32         // The browser doesn't support WebSocket
37 33         alert("WebSocket NOT supported by your Browser!\
38 ");
39 34     }
40 35 };
41 36
42 37 $("#message").createWebSocket();
43 38 </script>
44 39 </body>
45 40 </html>
```

重點說明如下：

jQuery 模式的精神

非常簡單，就可以將程式碼重構為 jQuery 插件模式（請參考第 1 章）。但是，一定要這麼做嗎？這要從 jQuery 的精神說起。

一般來說，JavaScript 最害怕去操作物件（Object）。根據 XXX 在他的著作「JavaScript Design Pattern」中的說明，使用選擇器（Selector）模式可以提升 JavaScript 程式碼的效能。效能的提昇關鍵為：選擇器模式以很有效率的方式去使用 DOM。jQuery 就是選擇器模式，並且能以高品質的代碼，提昇 DOM 的操作效率。

筆者使用另外一種更簡單的方式來說明。在 02-ws-jquery-plugin-pattern.html 裡有一個 Div 區塊叫做 'message'，重構後的例子使用了 jQuery 選擇器，並且呼叫了 'message' 的 createWebSocket() 方法。從物件導向的角度來看，createWebSocket() 被封裝在 'message' 物件裡了。所以，createWebSocket() 是 'message' 物件的一個方法，這個觀念得到二個好處：

- `createWebSocket()` 函數的操作範圍 (Scope) 是在 'message' 物件裡面；簡單來說
- 在 `createWebSocket()` 裡可以使用 'this' 物件，這實際上是一個參考 (Reference)，指向「物件自己」

重構前，因為沒有使用 jQuery 模式，所以差別如下：

- `createWebSocket()` 函數的操作範圍是全域環境 (Global)
- 無法使用 'this' 物件
- `createWebSocket()` 函數，操作的是外部物件

這是二個版本最大差異。所以，將程式碼重構為 jQuery 模式後，能給我們帶來許多好處。

考慮 Closure

重構後的 02-ws-jquery-plugin-pattern.html 還有一個需要考量的地方：Closure (封閉性)。

首先，利用第 1 章所介紹的 Module Pattern 觀念將程式碼全都「封閉起來」。Closure 是為了避免變數的污染：全域變數很容易受到其它地方程式碼的改寫。這就像老師開始上課時，要把教室門關起來一樣的道理：為了避免外界的干擾。

如果沒有把程式碼「關」起來，外界的程式碼可能干擾到我們，例如：全域變數被修改。再進行第二次的重構，結果如下：

client/err-ws-jquery-module-pattern.html

```
1  1  <!DOCTYPE html>
2  2  <head>
3  3  <script type='text/javascript' src='./jquery.min.js'\
4  ></script>
5  4  </head>
6  5  <body>
7  6  <div id="message"></div>
8  7
9  8  <script type="text/javascript">
10 9  (function($) {
11 10 $.fn.createWebSocket = function () {
12 11     if ("WebSocket" in window)
13 12     {
14 13         alert("WebSocket is supported by your Browser!"\
15 14 );
16 15         // Let us open a web socket
17 16         var ws = new WebSocket("ws://localhost:8080/sta\
18 17 rt");
19 18         ws.onopen = function()
20 19         {
21 20             // Web Socket is connected, send data using \
22 21 send()
23 22         ws.send("Message to send");
24 23     };
25 24     ws.onmessage = function (evt)
26 25     {
27 26         var received_msg = evt.data;
28 27         this.html(received_msg);
29 28     };
30 29     ws.onclose = function()
31 30     {
32 31         // websocket is closed.
33 32     };
34 33 }
```



```
34 30      ws.onerror = function()
35 31      {
36 32          this.html("<h1>error</h1>");
37 33      };
38 34  }
39 35  else
40 36  {
41 37      // The browser doesn't support WebSocket
42 38      alert("WebSocket NOT supported by your Browser!\
43  ");
44 39  }
45 40  };
46 41
47 42  })($);
48 43
49 44  $("#message").createWebSocket();
50 45  </script>
51 46  </body>
52 47  </html>
```

從程式碼第 9 行與第 42 行可以很明顯看出，我們將原本的程式碼封閉起來了。所以原本的程式碼具有了封閉性。並且根據第 1 章提到的觀念，jQuery 的選擇器 (\$) 要以參數傳遞的方式匯入 (Import) 到 Module 內部後再使用。

另外，這裡的實作也加入了 onmessage 與 onerror 二個回呼函數。當伺服器透過 WebSocket 傳送訊息過來時，onmessage 便會被呼叫。後續我們將擴充此函數，處理伺服器 Push 過來的即時訊息。

關於 *this* 物件

接下來，要解決一個 Bug。解掉這個 Bug 後，可以學到「this」的重要觀念。上述範例第 24 行與第 32 行的 *this* 並不是網頁

裡的 'message' 區塊。這和我們想達成的目的不同。所以執行 `client.html` 並無法看到我們想像中的結果。

上述範例第 24 行與第 32 行中所使用的 `this` 物件，是 `WebSocket` 類別的實例化，而不是 'message' 區塊。這與我們想實作的結果不同。原本我們期望可以直接使用 `this` 將訊息直接放到 'message' 區塊裡面，但問題出在哪裡？

原來，第 21 行與第 30 行的函數，其實都是物件。這就是 JavaScript 非常重要的觀念之一：函數即物件。意思是：

- 第 21 行用定義了一個 function 給 `onmessage`，而 function 就是物件，所以 `onmessage` 是一個物件
- 第 30 行同上
- 因此，在 24 行裡的 `this` 其實是上述的 `onmessage` 物件

第 24 行的 `this` 代表的是 `onmessage` 物件。要修正這個問題並不然，我們只要想清楚，在程式碼什麼位置，`this` 才是表示 'message' 這個區塊物件實例化即可。

修正後的正確版本如下：

`client/03-ws-jquery-module-pattern.html`

```
1  1 <!DOCTYPE html>
2  2 <head>
3  3 <script type='text/javascript' src='./jquery.min.js'\
4  ></script>
5  4 </head>
6  5 <body>
7  6 <div id="message"></div>
8  7
9  8 <script type="text/javascript">
10 9 (function($) {
11 10 $.fn.createWebSocket = function () {
12 11
```

```
13 12 // This 'this' is '#message' object according to t\  
14 his sample  
15 13 var div = this;  
16 14  
17 15 if ("WebSocket" in window)  
18 16 {  
19 17     alert("WebSocket is supported by your Browser!"\  
20 );  
21 18     // Let us open a web socket  
22 19     var ws = new WebSocket("ws://localhost:8888/sta\  
23 rt");  
24 20     ws.onopen = function()  
25 21     {  
26 22         // Web Socket is connected, send data using \  
27 send()  
28 23         // eg, ws.send("Message to send");  
29 24     };  
30 25     ws.onmessage = function (evt)  
31 26     {  
32 27         var received_msg = evt.data;  
33 28         div.html(received_msg);  
34 29     };  
35 30     ws.onclose = function()  
36 31     {  
37 32         // websocket is closed.  
38 33  
39 34         // 'this' here is [Object WebSocket]  
40 35         div.html("<h1>onclose</h1>");  
41 36     };  
42 37     ws.onerror = function()  
43 38     {  
44 39         // 'this' here is [Object WebSocket]  
45 40         div.html("<h1>onerror</h1>");  
46 41     };  
47 42 }
```

```
48 43     else
49 44     {
50 45         // The browser doesn't support WebSocket
51 46         alert("WebSocket NOT supported by your Browser!\
52 ");
53 47     }
54 48 };
55 49
56 50 })($);
57 51
58 52 $("#message").createWebSocket();
59 53 </script>
60 54 </body>
61 55 </html>
```

程式碼第 13 行就是關鍵： *div* 物件所儲存的 *this* 代表的是 'message' 這個區塊物件；因此，修改後的程式碼第 35 行與第 40 行，就可以成功將資料加入到 *message* 這個 *<div>* 裡了。

透過 jQuery Plugin Pattern 與 Module Pattern 的使用，程式碼的實作觀念，變成是在物件裡「操作自己」。如此一來，消除掉了對 DOM 的操作，也就是不必要的「Select」。這就是效能得以提昇的關鍵。

Binding 'this'

上述的觀念，又稱為 Binding *this*。有 JavaScript Framework 開發經驗的開發者，都知道「binding *this*」的重要性。這可是 JavaScript 開發模式的靈魂。所以，上述的寫法其實還有一個改善空間。

為了更容易了解這個觀念，我將「binding *this*」的觀念，修改為「Save *this*」。Binding *this* 的實作方式，就是把自己的 *this* 物件，儲存起來，留待後續使用。

重構為 jquery.websocket.js

首先，將 JavaScript 的部份重構為獨立的檔案，命名為 jquery.websocket.js。修改後的 HTML5 頁面如下：

client/04-ws-jquery-module-closure.html

```
1  1 <!DOCTYPE html>
2  2 <head>
3  3 <script type='text/javascript' src='./jquery.min.js'\
4  ></script>
5  4 <script type='text/javascript' src='./jquery.websocket\
6  et-1.0.js"></script>
7  5 </head>
8  6 <body>
9  7 <div id="message"></div>
10 8
11 9 <div id="admin">
12 10 <input id="msg" type="text" value="Input message"></\
13  input>
14 11 <button id="send">Send</button>
15 12 </div>
16 13
17 14 <script type="text/javascript">
18 15 $("#message").createWebSocket();
19 16 $("#send").sendMessage();
20 17 </script>
21 18 </body>
22 19 </html>
```

此外，我們也在 jquery.websocket.js 裡加上幾個新功能：

- 將變數 `ws` 重構為全域變數，由於我們採用了 Module Pattern 的觀念，因此 `ws` 被「封閉」在模組裡，不會受到外界的影響

- 修改 onWsMessage(), 利用 JSON API 來解析 JSON 文件

在這裡，還是要不煩其煩地強調 Closure 的觀念。要學好 JavaScript 程式開發，Closure 是重要的第一課。完成後的最新版本如下：

client/jquery.websocket-1.0.js

```
1  1 (function($) {
2  2
3  3 // WebSocket object
4  4 var ws;
5  5
6  6 // The Div element selected by jQuery selector
7  7 var div = this;
8  8
9  9 function onWsMessage(message) {
10 10     var json = JSON.parse(message.data);
11 11
12 12     if (json.type === 'message') {
13 13         content.prepend('<p>' + json.data.message + '</p>');
14 14     };
15 15 }
16 16
17 17 $.fn.receiveWebSocket = function () {
18 18     content = this;
19 19
20 20     ws.onmessage = onWsMessage;
21 21 };
22 22
23 23 $.fn.createWebSocket = function () {
24 24     if ("WebSocket" in window)
25 25     {
26 26         // Let us open a web socket
```

```
28 27     ws = new WebSocket("ws://svn.moko365.com:8080/s\
29 tart", ['echo-protocol']);
30 28     ws.onopen = function()
31 29     {
32 30         div.append("<h2>Done</h2>");
33 31     };
34 32
35 33     ws.onmessage = onWsMessage;
36 34
37 35     ws.onclose = function()
38 36     {
39 37         // websocket is closed.
40 38     };
41 39     ws.onerror = function()
42 40     {
43 41         div.html("<h1>error</h1>");
44 42     };
45 43 } else {
46 44     // The browser doesn't support WebSocket
47 45     alert("WebSocket NOT supported by your Browser!\
48 ");
49 46 }
50 47 };
51 48
52 49 })(\$);
```

結論

透過一個即時聊天室的開發過程，我們初步學會了重要的 WebSocket 基礎知識：

- 開發 WebSocket Server
- 開發 WebSocket Client

- 建立 Server/Client 的 WebSocket 連線
- 使用 WebSocket 做 Real-time Data Push

關於 NoChat 範例，目前還沒有使用資料庫來儲存訊息。後續將介紹 Node.js 如何使用資料庫的，有幾個相關的重要觀念，先整理如下：

- 如果要將即時訊息儲存下來，就要使用到資料庫。目前 Node.js 所使用的資料庫，是一種以 JSON Document 形式儲存資料的技術，並非傳統的 SQL 資料庫（關聯式資料庫）技術
- 這類型的資料庫，是一種 JSON Document-based 資料庫，故名思義，它是一種以 JSON 文件方式來儲存資料的 Database
- 這種不以 SQL 查詢語言為主的資料庫，又稱為 NoSQL (Not Only SQL)
- 較知名的有 MongoDB 與 CouchDB

由於這種資料庫，並不是以傳統的 SQL 查詢與儲存，因此也稱為「NoSQL」的資料庫。

軟體思惟 - Lambda 篇

本章的目標，是提昇初學者的開發功力。經過前面 4 個章節的介紹，學習到許多基本觀念與技術。接下來，將從三個不同的層面，深化目前所學到的觀念與技術：

- JavaScript 語言
- Web Service 架構
- Node.js 觀念

本章先從 JavaScript 語言開始。JavaScript 最重要的觀念是 Closure，這可以說是 JavaScript 初學者的第 1 堂。Closure 與匿名函數有很緊密的關係，這要由 Lambda 的觀念開始講起。

Lambda

Lambda (λ) 是一個希臘字母 (Λ 是它的大寫字母)，用來表示許多觀念：

- 物理學家用來表示波長的符號
- 數學家用來表示空字串的符號
- 電腦科學家用來表示匿名函數 (Anonymous Function) 的符號

Lambda 在電腦科學領域，用來表示匿名函數，目的是進行運算。為了尋找一個語法簡易的運算表示方式，電腦科學家會這麼做。

Step 1: 取得一個具名的函數

例如一個加法函數：

```
1 sum(x) = x + 2
```

函數 `sum()` 是具名函數，利用 JavaScript 來實作的話，寫法如下：

```
1 function sum(x) {  
2     return x + 2;  
3 }
```

Step 2: 改寫為匿名函數

Lambda 希望可以找到一個能運算的表示方法，而且要簡單，去除掉函數名稱，就是一個方式。沒有具體名稱的函數，就稱為匿名函數（Anonymous Function）。如果表示匿名函數呢？上述範例，以 Lambda 來表示的話，只要改寫成：

```
1 λx.x + 2
```

用 JavaScript 來實作的話，要如何撰寫呢？方式如下：

```
1 (function(x) {  
2     return x + 2;  
3 })();
```

這就是第 1 章介紹的 Closure 觀念，最後加上的一對括號，稱為立即函數，意思是立即執行此匿名函數的意思。立即函數，以物件導向的角度來看，也可以解釋為立即實例化。

Step 3: 使用 `var` 來宣告匿名函數

JavaScript 的 `var` 關鍵字用來宣告變數，所以也可以把匿名函數做為運算子（Operator）來宣告變數。例如：

```
1 var lambda = (function(x) {  
2     return x + 2;  
3 })();
```

變數 *lambda* 被指定 (Assign) 為一個匿名函數。從 JavaScript 語言的角度來看, Closure 用來封裝出 Module, 所以 *lambda* 變數也可以解釋成「一個模組」。宣告一個匿名函數的變數時, 可以不需要 Closure。也可以採用以下的寫法:

```
1 var lambda = function(x) {  
2     return x + 2;  
3 };
```

事實上, 這個寫法更為普遍。一些文章也把這種寫法, 稱做 Function Expressions。

有一個重要的觀念要釐清, 上述的寫法, 正規的解釋方式是「宣告匿名函數的變數」, 所以 *lambda* 是一個變數, 不能解釋為函數。如果說 *lambda* 是一個函數宣告, 觀念上就不對了。不能單就 JavaScript 語法的角度來做解釋。

以 JavaScript 來說, 函數宣告的寫法為:

```
1 function lambda(x) {  
2     return x + 2;  
3 };
```

這個時候, *lambda* 就是一個函數。那一種寫法比較好呢? 理論上, 採用匿名函數宣告的方式較佳, 因為 JavaScript 本身是一種 Lambda 的程式語言。

Callback Function

Lambda 本質上是一種表示方法, 用來表示 Input 與 Output。所以, 要表示一個平方的運算的話, 寫法如下:

1 $\lambda x. x * x$

這個運算式，也可以做為另一個 Lambda 式子的 Input，請看以下的說明。

Step 1: 撰寫第一個計算平方的 Lambda 表示式

做法如下：

1 $\lambda x. x * x$

Step 2: 把上述的式子做為另一個 Lambda 的輸入

有一個 $f(x) = x + 2$ 的函數，用 Lambda 來表示的話，寫法如下：

1 $(\lambda x. x + 2)$

如果要把 Step 1 的結果，做當上面式子的 x （輸入），合併後的寫法為：

1 $(\lambda x. x * x)(\lambda x. x + 2)$

讓我們來筆算看看：

- $x = 3$ 時， $\lambda x. x * x$ 的 Output 為 9
- 9 做為 $\lambda x. x + 2$ 的 Input，成為 $9 + 2$ ，Output 為 11
- 答案就是 11

$(\lambda x. x * x)(\lambda x. x + 2)$ 等價於 $9 + 2$ 。

Step 3: 使用 JavaScript 來實作

怎麼把 $(\lambda x. x * x)(\lambda x. x + 2)$ 寫成程式碼呢？非常簡單，由於 Output 就是返回值，所以等於「讓匿名函數的返回值，當做另一個匿名函數的參數」。程式碼如下：

```

1  var lambda = function(x) { return x + 2 };
2  var result = lambda(function(x) { return x * x } (3) );
3
4  console.log("Result: " + result);

```

這個輸出的輸出結果為 11。這可不是在賣弄程式碼，而是實現出 $(\lambda x.x*x)(\lambda x.x+2)$ 這個 Lambda 演算。以匿名函數來表示 Lambda 是很常見的做法，但其實反應出 JavaScript 語法上的不足。如果能有一個更簡易的語法，讓我們表示 Lambda，這段程式碼就會比較精簡。

所以，這很可能要從修改 JavaScript 語法的角度，來做強化。未來，新的 ECMAScript (JavaScript 的語法標準) 或許可以讓我們這樣寫程式：

```

1  function(x) { return x + 2 }  // 複雜的寫法
2  x => x + 2                    // 希望可以有這種精簡

```

再舉一個例子：

```

1  var result = (x => x * x) 3;  // 左結合寫法，輸入值 3 放在最右邊，最後 r\
2  esult 為 9

```

如果將 Lambda 做為函數的參數：

```

1  [1, 2, 3, 4, 5]
2      .map(function(x) { return x * x });

```

未來或許能簡化為：

```

1  [1, 2, 3, 4, 5]
2      .map(x => x * x);          // 希望可以有這種精簡的語法

```

在 ECMAScript 還沒有正式加入相關語法前，我們目前還是只能使用匿名函數的寫法。

使用 TypeScript

在深入了解 Lambda 的觀念後，就能知道根本之道在於「新的 JavaScript 語法」。目前有一些 Open Source 程式庫，就試著在解決這個問題。唯有透徹底了解 Lambda 的觀念，才能知道這些程式庫目地何在。

筆者推薦的解決方案是：TypeScript。這是一個由 Microsoft 所開發的工具，實際上是一個 Compiler。TypeScript 提供了擴充的 JavaScript 語法，可藉由 TypeScript 編譯為標準的 JavaScript 語法。

首先，必須先使用 npm 安裝 TypeScript 工具：

```
1 $ npm install -g typescript
```

接著，以 TypeScript 的語法撰寫 JavaScript 程式碼。TypeScript 的語法就是 JavaScript 語法，只是提供了許多好用的擴充語言，因此學習上並沒有障礙。以下是一個 TypeScript 範例：

l.ts

```
1 var square = (x) => x * x  
2 console.log(square(3));
```

將程式碼儲存為 l.ts 後，再利用 TypeScript 編譯：

```
1 $ tsc l.ts
```

編譯後可以得到 l.js 檔案，以下是 l.js 的內容：

```
1  var square = function (x) {  
2      return x * x;  
3  };  
4  console.log(square(3));
```

看到編譯後的程式碼後，馬上可以反應出這個觀念：TypeScript 提供一個簡單好用的 Lambda 語法。當然，TypeScript 的功能很豐富，這只是牛刀小試。

軟體思惟 - Web Service 篇

延續第 5 章，在 Web Service 架構的部份，將會介紹 RESTful Architecture。要了解 RESTful Architecture 的精神，就要從 HTTP API 講起。

再探 HTTP API

說明 Web Service 觀念的最快方式，就是透過 HTTP API。例如：

```
1 http://www.moko365.com/api/query?t=users
```

利用 HTTP 協定來呼叫這個 API（最簡單方式是用瀏覽器），Server 會以 JSON 格式回應這個請求。例如，上述 API 向 Server 請求「查詢上線人數」的服務，Server 將以這個 JSON 來回應「目前有 10 個人上線」：

```
1 {  
2     "online": 10  
3 }
```

JSON 格式是由我們自定。這是截至目前為止，我們所學到的觀念。這樣的 API 形式，就是典型的 CGI 程式設計觀念，這部份在前面的章章已做過介紹。

上述的 HTTP API 例子，問號後的字串稱為 Query String。這個例子，瀏覽器要傳遞 `t=users` 的字串給 Server。傳遞 Query String 的方式有 2 種：

- 使用 HTTP 協定的 GET 方法
- 使用 HTTP 協定的 POST 方法

HTTP API 採用 URL 的形式，瀏覽器傳送 URL 請求的做法可分為二種：GET 或 POST。然而，這種 HTTP API 的格式，過於複雜，而且它是從「功能」的角度來描述 API。如果 Web Service 採用這種 API 的寫法，它的軟體架構就會較為複雜，而且也不易於描述「網路上的資源」。

例如，我想開發一個 Web Service 的 API，功能是：新增名為 James 的使用者。要如何定義這個 API 呢？寫法如下：

```
1 http://www.moko365.com/api/add_user.php?username=james&\
2 type=admin&email=who@anywhere.com
```

上述這個 API 必須以 HTTP 的 GET 方法來呼叫，所以整個 HTTP 請求會變成：

```
1 GET http://www.moko365.com/api/add_user.php?username=ja\
2 mes&type=admin&email=who@anywhere.com HTTP/1.1
```

它的缺點：

- API 形式不夠精簡
- Web Service 軟體架構複雜且不夠嚴謹
- 功能導向的描述，不易於描述資源（使用者為一個資源）

這個例子，是實作 Web Service 的幾種方法之一，當然這種方式仍有存在的價值。因此，這裡並非要推翻這個做法，而是要尋找更多的實作技術。

後來出現一些更嚴謹的 Web Service 實作技術，例如：SOAP (Simple Object Access Protocol)。SOAP 的請求與回應，都採用

XML 的文件形式，這讓 API 形式較為精簡，而且 Web Service 軟體架構也更為嚴謹。

SOAP 的另一個貢獻是：標準化了交換格式。Client 與 Server 之間的資料交換，都採用 SOAP 的標準。這不像典型的 CGI 開發，交換格式沒有一定的標準。資料交換格式標準化後，就易於系統間的整合。例如，每家公司的 Web Service 資料格式都是自訂的，這種百家爭鳴的情況，會造成彼此間互換資料的困擾，當然整合與維護就會是個大問題。

REST

SOAP 仍然太過於複雜，為了尋求一個更精簡的做法。Roy Fielding 博士於 2000 年提出一種稱為 REST 的 Web Service 軟體架構。[Roy Fielding⁶](#) 是 Apache HTTP Server 專案的共同創辦人，他在 HTTP 協定上有非常重要的貢獻，REST 架構則是他的博士論文題目。

REST 本質上是一種軟體架構，採用 REST 來定義 API 並開發 Web Service 軟體時，這個軟體就稱為 RESTful ARchitecture。RESTful Architecture 的 API 風格更為精簡，並且以「資源」的角度去描述 Web 服務。例如，新增名為 James 的使用者，定義這個服務的思惟邏輯如下：

Step 1: 找出資源項目

以「James 這個使用者」的角度來看，資源的項目應該是「使用者」，所以這個 RESTful Web API 的定義可寫為：

1 <http://www.moko365.com/resource/user>

這是一種 URI 的形式，上述的 URL 也是 URI 的一種寫法。RESTful Web API 以 URI 形式定義，並且要遵循 REST 標準的風格，遵循 REST 風格可以讓 API 更精簡。

⁶http://en.wikipedia.org/wiki/Roy_Fielding

Step 2: 描述一位使用者

使用者的名稱是 James，所以 RESTful Web API 為：

1 <http://www.moko365.com/resource/user/james>

Step 3: 說明請求 (CRUD)

所謂的說明請求，就是說明我們想對“James”所做的動作。對於一個資源來說，動作不外乎：

- 新增 (Create)
- 讀取 (Read)
- 更新 (Update)
- 刪除 (Delete)

以上 4 個動作，就稱為 CRUD。怎麼跟 Web Service 說明動作呢？RESTful Web API 是基於 HTTP 協定，所以就是使用 HTTP 協定裡的 4 個方法 (HTTP Method)：

- GET
- PUT
- POST
- DELETE

舉個例子，現在要新增 James 使用者，就要使用 POST 方法。完整的 HTTP 請求如下：

1 POST <http://www.moko365.com/resource/user/james>

HTTP 協定所提供的 4 個方法，可以完整地對應到 CRUD。

CRUD 與 HTTP Method

哪一個 HTTP Method 對應到 Create？整理如表 6.1。

表 6-1 CRUD 的對應

CRUD	HTTP Method	用途說明
Create	POST	新增
Read	GET	讀取
Update	PUT	更新
Delete	DELETE	刪除

RESTful Web API 的定義，只有風格，沒有很固定的標準。所以只要依照這個風格，來定義自己的 API 即可。

了解 REST 的觀念後，馬上可以聯想到：要對第 3 章與第 4 章的 NoChat 範例做調整。原本傳送訊息的 API:

```
1 http://localhost:8080/send?m=hello
```

要修改為儲存資源的概念，即 REST 架構:

```
1 POST http://localhost:8080/resource/message/hello
```

用 POST 方法，來描述 “/resource/message/hello” 這項資源，意思是新增 “hello” 訊息。上述的二個 API 是等價的，都是儲存訊息，但概念有很大的不同:

- 定義 “/send” API，這是一個「傳送」的動作，這是以功能為導向的軟體思惟
- 定義 “/resource/message” 項目，這是一個資源
- “/resource/message/hello” 描述 “message” 資源裡的一個資源（項目）
- 用 POST 來描述 “/resource/message/hello” 資源，表示對資源做新增的動作

如果要加入「讀取所有訊息」的 Web 服務，採用先前的觀念，會定義這個 API:

1 `http://localhost:8080/query?type=all`

採用 REST 風格的話，寫法為：

1 `GET http://localhost:8080/resource/message`

除了 API 風格要改變外，NoChat 裡的「URL Routing」實作，也要一併做修改。

實務上，我們會尋求一個現成的框架（Framework），讓我們更容易處理 URL Routing；除非有特別的原因，通常不會從零自行實作。Web Application Framework（或稱為 Web Service Framework）都具備 URL Routing 的處理能力。

目前搭配 Node.js 最熱門的 Web Application Framework 就是 [Express.js](http://expressjs.com/)⁷，我們將在第 8 章導入這項重要的技術。

結論

最後，可以這樣說，RESTful 是一種雲端（Client/Server）軟體架構。實作 Web Service（Back-end）的技術很多，現今最熱門的就是 Node.js。開發 RESTful Architecture 的 Web Service，只需要 2 個大步驟：

- 描述每一個資源
- 實作每一個資源的 CRUD

描述並實作每一個資源的 CRUD，並且以 JSON 格式交換資料，就是 REST + JSON 的觀念。

⁷<http://expressjs.com/>

軟體思惟 - Non-blocking IO 篇

要繼續深入 Node.js 開發，許多 Node.js 的觀念必須陸續建立起來。對初學者來說，最重要的莫過於 Non-blocking IO 的觀念。

Node.js 的網站上，以精簡的 2 個特色來說明其技術特色：

- Event-driven
- Non-blocking IO

第一個特色，在第 1 章做過介紹，以 Event Loop（Event-driven）的觀念來取代 Multi-thread。第二個特色，則是 Node.js 最重要的技術。要了解 Non-blocking IO 的觀念，只要練習做一個題目即可：檔案讀取。



圖 7-1: Node.js 官網

用 Node.js 撰寫二個範例：

- 讀取一個檔案
- 讀取多個檔案

看似簡單的二個題目，思惟邏輯卻經常困擾 Node.js 的新手，特別是從 C/PHP/Java 語言，進入 Node.js 程式設計的初學者。

範例程式可由 Github 取得：

<https://github.com/jollen/nodejs-readfile>

再繼續進行前，大家不妨使用 C/PHP/Java 任何一個語言，撰寫「讀取多個檔案」的程式，並搭配本章的範例，進行觀念的探討。

Node.js File System

Node.js 提供 File System 模組，支援檔案的讀取等功能。參考 Node.js 的 API 手冊，找到 *fs.readFile()* 函數⁸：

```
1 fs.readFile(filename, [options], callback)
```

readFile() 的第三個參數，是一個 Callback Function，這就是第 5 章所介紹的 Lambda 觀念。為什麼要傳遞 Callback Function 做為參數呢？這要從 Non-blocking IO 的觀念說起。

典型的 C/PHP/Java 語言，都支援同步式的 File System 操作。例如，從 C/PHP/Java 語言進到 Node.js 開發的初學者，可能會寫出這段程式碼邏輯：

⁸http://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback

```
1 fs.readFile('hello.txt', data);  
2 console.log(data);
```

請注意，這只是一段虛擬碼，並不能執行。這段虛擬碼的想法如下：

- 呼叫 `readFile()` 來讀取 'hello.txt' 檔案
- 讀取的內容放到 `data` 變數
- 然後將 `data` 變數的內容（檔案內容）送出

這是一個循序邏輯（Procedure）的觀念。Node.js 的 `readfile()` 是非同步式操作，也就是 Non-blocking IO 的觀念。Node.js 並不會等到 `readFile()` 完成工作後，「才繼續往下執行」，而是「立即往下一行執行」。

也就是說，當下一行執行時，`readFile()` 可能還在讀取 'hello.txt' 檔案。所以 `console.log()` 不一定能印出完整內容。`data` 變數也有可能是空的。

Node.js 的特色是支援 Non-blocking IO：它不會等到 IO 完成後，才往下執行。所以，完成檔案讀取時，很可能程式碼已經執行到非常後面了。要如何知道檔案已讀取完成呢？`readFile()` 完成工作後，就會 Callback 我們指定的函數。如此一來，就知道檔案已經讀取完成了。

正確的程式實作如下：

```
1 var fs = require('fs');  
2  
3 fs.readFile(filename, 'utf8', function(err, data) {  
4     console.log("[DATA] " + data);  
5 });
```

Callback Function 是一個暱名函數（Lambda），當它被呼叫時，表示已完成檔案讀取，並且收到二個參數。根據 Node.js 的手冊，第二個參數存放讀取到的內容。

了解 Non-blocking IO 的觀念後，就會知道以下是一個錯誤寫法：

```
1 var fs = require('fs');  
2  
3 fs.readFile(filename, 'utf8', function(err, data) {  
4   });  
5  
6 console.log("[DATA] " + data); // 錯誤寫法
```

這就是 Node.js 最重要的觀念：Non-blocking IO，實作上，搭配 Callback Function 的觀念。

讀取多個檔案

以上是讀取一個檔案的範例。現在，讓我們來改寫程式：

- 讀取 index.txt 檔案，index.txt 列出多個文字檔
- 再讀取 index.txt 裡列出的檔案

例如，index.txt 的內容如下：

```
1 chapter1.txt  
2 chapter2.txt  
3 chapter3.txt  
4 chapter4.txt  
5 chapter5.txt
```

再分別去讀取上述共 5 個檔案的內容。撰寫這個練習題，要先分析幾個觀念：

- 以 Line-by-Line 方式讀取 index.txt

- 讀取 `index.txt` 的動作，是 Non-blocking IO
- 等到 `index.txt` 讀取完成後，再開始讀取其它檔案
- 如上，在 `Callback Function` 裡，撰寫「讀取其它檔案」的程式碼

有一個 Node.js 的模組：`Node-BufferedReader`，可以協助我們做 Line-by-Line 的讀取。必須先安裝這個模組：

```
$ npm i buffered-reader
```

用法如下：

```
1 1 var reader = require ("buffered-reader");
2 2 var DataReader = reader.DataReader;
3 3
4 4 new DataReader ('index.txt', { encoding: "utf8" })
5 5   .on ("error", function (error){
6 6       console.log(error);
7 7   })
8 8   .on ("line", function (line, nextByteOffset){
9 9       console.log("[LINE] " + line);
10 10  })
11 11   .on ("end", function (){
12 12       // finish reading
13 13   })
14 14   .read();
```

`buffered-reader` 會逐行讀取 'index.txt'，完成單行讀取後，觸發 `line` 事件，並 `Callback` 一個函數。這部份實作在第 8 行，`Callback Function` 的第一個參數 `line` 存放讀取到的內容。

接著，將檔名儲存到陣列裡備用：

```
1  1 // the file list
2  2 var files = [];
3  3
4  4 new DataReader (path + 'index.txt', { encoding: "utf\
5  8" })
6  5     .on ("error", function (error){
7  6         console.log (error);
8  7     })
9  8     .on ("line", function (line, nextByteOffset){
10 9         files.push({
11 10             filename: line
12 11         });
13 12     })
14 13     .on ("end", function (){
15 14         // finish reading
16 15     })
17 16     .read();
```

要如何讀取 *files* 陣列裡的檔案呢？只要撰寫一個迴圈即可：

```
1  1 for (var i = 0; i < files.length; i++) {
2  2     var filename = files[i].filename;
3  3
4  4     fs.readFile(filename, 'utf8', function(err, data)\
5  5     {
6  6         console.log("[DATA] " + data);
7  7     });
8  8 }
```

這段程式碼，是本章的另一個重點：

- 第 4 行是 Non-blocking IO
- 所以不會等到 *readFile()* 讀完檔案，會立即往下執行，進入下一個迴圈

這個迴圈並不會因為「讀取檔案」，而花費太多的執行時間，這就是 Node.js 能做到「efficient」與「real-time」的關鍵。完整的程式碼如下：

readfile.js

```
1  1 // public modules
2  2 var util = require('util');
3  3 var fs = require('fs');
4  4 var reader = require("buffered-reader");
5  5 var DataReader = reader.DataReader;
6  6
7  7 // the file list
8  8 var files = [];
9  9
10 10 /**
11 11  * Use BufferedReader to read text file line by line.
12 12  *
13 13  * See: https://github.com/Gagle/Node-BufferedReader
14 14  */
15 15 var getFilelist = function(path, cb) {
16 16     new DataReader (path + 'index.txt', { encoding: \
17 17 "utf8" })
18 18     .on ("error", function (error){
19 19         console.log (error);
20 20     })
21 21     .on ("line", function (line, nextByteOffset){
22 22         files.push({
23 23             filename: line
24 24         });
25 25     })
26 26     .on ("end", function (){
27 27         cb(files);
28 28     })
29 29     .read();
30 30 };
```

```
31 30
32 31 // Application
33 32 getFilelist('manuscript/', function(files) {
34 33     for (var i = 0; i < files.length; i++) {
35 34         var filename = 'manuscript/' + files[i].filename\
36 e;
37 35
38 36         fs.readFile(filename, 'utf8', function(err, \
39 data) {
40 37             console.log("[DATA] " + data);
41 38         });
42 39     }
43 40 });
```

以下是執行結果:

```
1 [DATA] This is chapter 1.
2
3 [DATA] This is chapter 2.
4
5 [DATA] This is chapter 3.
6
7 [DATA] This is chapter 4.
8
9 [DATA] This is chapter 5.
```

要特別注意的是，上述的訊息順序是「非確定性」。由於 *readFile* 是 Non-blocking IO，所以「先讀取的檔案」，不一定「先完成讀取」。例如，'chapter1.txt' 的檔案很大時，就可能比其它後續的檔案，更晚完成讀取。

結論

根據筆者的教學觀察，Non-blocking IO 與 Callback Function 的觀念，是原本 C/PHP/Java 程式設計者，跨入 Node.js 開發的一個障礙。要克服這個障礙，就要建立 Non-blocking IO 的觀念，並訓練程式設計邏輯。這和 Procedure 式的執行行為有很大的不同。

Non-blocking 的觀念，也稱做 Wait-free。利用 Wait-free 的觀念來實作 Web Service 的話，可以達到較佳的 Concurrency 處理能力；例如：提昇「Requests per Second」，增進 Server 的處理性能。

Node.js 應用 - Express.js 入門

經過前幾章的基礎學習後，接下來開始進入 Node.js 的進階主題。首要課題就是導入一個 Web Application Framework，以協助開發者發展 Web Application。現在，我們需要一個 Web Application Framework，來簡化 URL Routing 的開發，一個很好的選擇是 Express.js。

Express.js 除了支援 URL Routing 的功能外，也提供很完整的 Web Application 支援。Express.js 的功能強大，對 Express.js 的初學者來說，要先學好以下 3 個主題：

- 使用 Express.js 做 URL Routing
- Express.js 與 HTML Template 的結合
- 學習 Jade Programming Language

本章的範例可由 Github 取得：

1 <http://github.com/jollen/nodejs-express>

接下來，透過一個連貫性的實例，學習 Express.js、HTML Template 與 Jade 程式語言。

Express.js 初體驗

NoChat 是第 3 章與第 4 章的實例，包含 Server 端與 Client 端。Client 端是一份 HTML5 網頁，這份網頁應該要放到 Web Server 上存取。典型的 Web Server 是 Apache 軟體，所以可以使用 Apache 來架構自己的 Web Server，以便瀏覽 Client 端網頁。

另外一個做法，是使用 Node.js 的 Web Server 功能，將網頁放到 Node.js 的 Web Server 裡。這個功能如果使用 Express.js 來做的話，只需要幾個步驟即可完成。

Step 1: 安裝 Express CLI

利用 npm 安裝 Express 的 CLI (Command Line Interface):

```
1 $ sudo npm -g i express
```

接著利用 express 命令來建立新的 Express.js 專案。

Step 2: 建立 Express.js 專案

先建立一個新的專案目錄，在這個目錄下，使用 npm 安裝 Express.js:

```
1 $ mkdir nodejs-express
2 $ cd nodejs-express
3 $ npm i express
```

安裝完成後，執行 Express.js 命令列工具:

```
1 $ express .
2 destination is not empty, continue? y
3
4   create : .
5   create : ./package.json
6   create : ./app.js
7   create : ./public
8   create : ./public/javascripts
9   create : ./public/images
10  create : ./public/stylesheets
```



```
11     create : ./public/stylesheets/style.css
12     create : ./routes
13     create : ./routes/index.js
14     create : ./routes/user.js
15     create : ./views
16     create : ./views/layout.jade
17     create : ./views/index.jade
18
19     install dependencies:
20       $ cd . && npm install
21
22     run the app:
23       $ node app
```

在執行 `app.js` 前，需要安裝 Express 所需的相依模組：

```
1 $ npm i
```

接著依照提示畫面的說明，執行 Express 的主程式：

```
1 $ node app.js
2 Express server listening on port 3000
```

Express.js 會自動生成 `app.js` 框架，這是 Web Application 的主程式，開發人員只要基於 `app.js` 來做擴充即可。利用瀏覽器開啓 `http://localhost:3000/` 網址，可以看到圖 8-1 的畫面，表示一個基本的 Web Application Framework 已經順利啓動了。

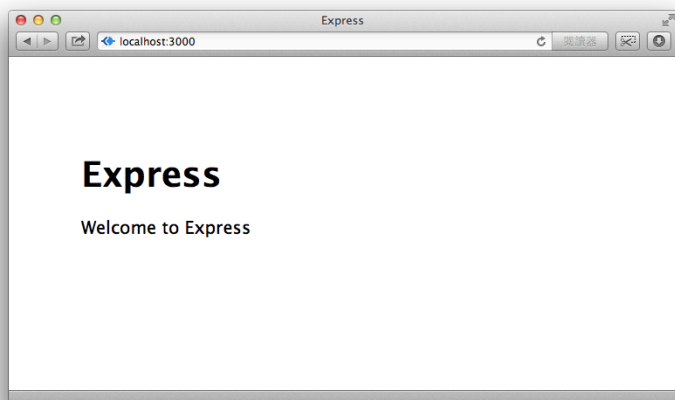


圖 8-1 啟動第一個 Express App

Express.js 是一個 Web Application Framework，也可以很方便地協助我們發展 Web Service API。基本上，Express.js 並不是一個 Web Server 的模組，因此不要誤解 Express.js 只是一個輕量級的 Web Server；Express.js 的本質是一個 Web Application 開發框架。

此外，要學好 Express.js，就要了解它的基本精神。Express.js 的基本精神，是提供了一個基於 URL Routing 機制的應用程式式框架。此外，基於 Express.js 所發展的 Web 應用程式，會以 API 的形式呈現出來。這裡所指的 API 可以是典型的 URL (CGI) 形式，也可以是 REST 的標準。

總結來看，Express.js 並不只是一個提供網頁服務的 Web Server，即使這也是 Express.js 其中一個功能。

Step 3: 認識 Express 專案結構與 MVC 模式

以下是 Express 所生成的專案結構 (Structure)：

- *app.js*: Web Application 原型

- *package.json*: NPM 套件管理程式設定檔
- *public/*: 放置靜態文件
- *routes/*: 處理 URL Routing 的程式碼
- *views/*: 放置 View 文件

關於 View 文件的觀念，在後續討論 MVC 設計模式時再做說明。目前，有一個很簡單的問題：圖 8-1 的主畫面存放在什麼位置？從典型的「網頁製作」觀念來看的話，應該會有一份 *index.html* 的文件，並且可能放置在 *public/* 目錄下。

然而，圖 8-1 的主畫面，實際上是由 *views/index.jade* 所動態產生。與網頁製作的概念比較，可以整理出幾個很基本但相當重要的觀念：

- *index.html* 是一種製作文件（Web Pages）的觀念，這是一個開發網站的概念
- *index.jade* 是一個 HTML5 模板（HTML5 Template），製作模板的語法稱為 Jade
- *index.jade* 是製作應用程式（Web Apps）的觀念，應用程式的 Presentation 與程式碼要切割開來
- Presentation 的部份以 Jade 撰寫，並存儲在 *views/* 目錄下
- 程式碼的部份，以 JavaScript 撰寫，這是一個靜態文件，所以應儲存於 *public/javascripts/* 目錄下
- 也就是說，這個 Web App 的架構採用 MVC 設計模式

Express 框架把基本的 MVC 模式都做好了。簡單來說，利用 Node.js + Express 可以很輕地開發 Web Application，這個 Web Application 的基本能力是提供 Web Service API。

Step 4: 佈署靜態文件

Express 生成的專案結構，規劃了一個存放靜態文件的目錄，常見的靜態文件有 CSS、JavaScript 與圖片等等。要將

Express 當做典型的 Web Server 來使用時，只要將靜態文件放置於 *public/* 目錄下即可。例如，要佈署 Bootstrap 至目前的 Express.js 專案，做法如下：

```
1 $ cd nodejs-express      (切換到專案目錄)
2 $ cd public              (進入 public/ 目錄)
3 $ cd stylesheets         (Express.js 所規劃用來存放 CSS 的目錄)
```

接著將 Bootstrap 的 CSS 下載至這個目錄下：

```
1 $ wget http://netdna.bootstrapcdn.com/bootstrap/3.0.1/css/
2 ss/bootstrap.min.css
```

同理，將 JavaScript 程式碼下載至對應的目錄下：

```
1 $ cd ../javascripts/
2 $ wget http://netdna.bootstrapcdn.com/bootstrap/3.0.1/js/
3 s/bootstrap.min.js
```

可以利用 *tree* 命令，來檢查目前為止的 *public/* 狀態：

```
1 |─ public
2 |   |─ images
3 |   |─ javascripts
4 |   |   |─ bootstrap.min.js
5 |   |─ stylesheets
6 |       |─ bootstrap.min.css
7 |       |─ style.css
```

style.css 是 Express.js 自動產生的檔案。到這裡，經由以上 4 個學習步驟，我們可以知道如何將 NoChat 範例佈署至 Express.js 專案裡，並透過 Express.js 來提供網頁的服務。這部份留給讀者自行練習。

Step 5: 擴充 app.js

app.js 是 Express 生成的 Web Application Framework。開發人員將基於 *app.js* 擴充這個 Web Application Framework 的功能：

- 擴充 URL Routing
- 實作與定義 REST API
- 加入 Middleware
- 使用 HTML Template 撰寫 UI

關於以上 4 個主要的 Express 技術，後續將以獨立的章節進行說明。這是開發 Express.js 的重點戲，對初學者來說，必須先學習 URL Routing 的開發。關於 URL Routing 的觀念，已經在第 3 章做過說明。

Express.js 的學習建議

以下是建議的 Express.js 學習步驟，初學者可依照這個順序進行練習。在熟悉每一個步驟後，再繼續研讀更多主題。

- Stage 1: 學會建立新的 Express.js 專案
- Stage 2: 了解 Express.js 專案的目錄結構
- Stage 3: 練習加入 Bootstrap
- Stage 4: 學習 Jade 語法並撰寫第一個“View”
- Stage 5: 練習新增 URL Routing

截至目前為止，我們已經學會了 Stage 1~3 的做法，接下來繼續說明 Stage 4 與 Stage 5 的做法與觀念。

MVC 與 HTML Template Engine

MVC 模式的全名是 Model-View-Controller，這是一種軟體架構。學習以 MVC 觀念，來重新製作 NoChat 用戶端網頁，很快就能體會到 MVC 的精神。MVC 中的“V”指的是“View”，View 的意思是「可見」，也就是人類眼睛所能看到的畫面。

一般來看，這裡所指的 View 就是 Web UI 的部份；不過 View 也代表 Presentation（視覺），UI 與 Presentation 意義上並不完全相同。關於這點，後續討論 MVP 模式時，再另行說明。以下皆以 Presentation 來表示 Web UI，也就是 View 的部份。

MVC 的基本精神是將 Presentation 與程式碼分開。Express 所產生的 `view/` 目錄就是用來存放 Presentation，也就是 UI。將 UI 與程式碼切割，不助於維護良好的程式碼結構，讓程式設計人員將程式碼組織的更好。

MVC 的觀念在軟體工程裡非常重要，程式設計師可以透過 MVC 模式，讓程式碼的組織更好。有了 MVC 模式，整個專案的管理結構也會更好，例如：結構更好的目錄階層。

Express.js 是一個 MVC 的 Web Application 框架，要展現 MVC 模式的精神，首要的工作就是導入 Template 觀念，利用 Template 來撰寫 View（HTML5）。

學習 Jade 程式語言

HTML 的 Template 選擇很多，目前 Express.js 能支援的 Template Engine 有：

- hjs
- hbs
- Jade

Express.js 已經將 Jade 做為預設的 Template Engine。Jade 的語法風格簡潔，而且易學。Jade 將會成為重要的 HTML Template

Egine，以及 HTML Template 程式語言。建議讀者儘量去先了解它的語法。

Jade 的語法非常容易學習，只要透過幾個基本的練習，就可以很快上手。以下以 3 個練習，介紹 Jade 程式語言的基本語法。關於 Jade 語法的完整說明，請參考：

<http://jade-lang.com>

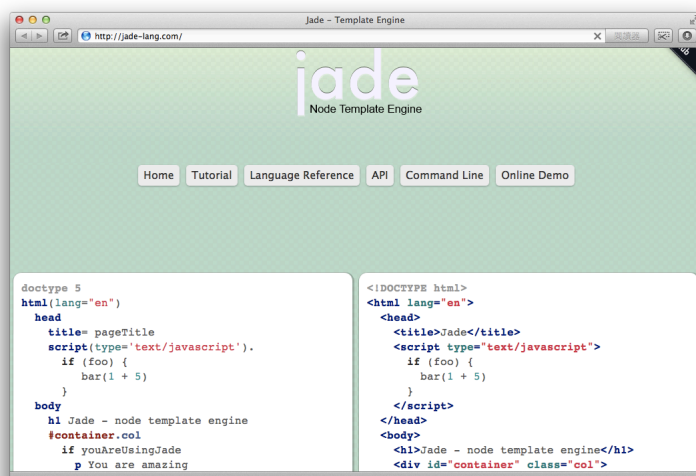


圖 8-2 Jade 是一個 Template Engine

Exercise 1: <h1> 大標題

接下來，說明如何在 Express.js 框架裡加入一個新的 HTML5 頁面。首先，先練習將以下的內容，改寫為 Jade，並放置於專案的 views/ 目錄下：

1 <h1>Hello World!</h2>

改寫為 Jade 語法：

```
1 h1 Hello World!
```

將上述內容儲存為 *views/hello.jade*。經由這個例子，可以知道 Jade 提供了一套很簡約的 HTML5 標籤語法。使用 Jade 來撰寫 HTML5 標籤，更不必擔心漏寫了結尾標籤，非常方便。

Exercise 2: <p> 段落

如果要寫入一段文字的話呢？HTML5 利用 <p></p> 標籤來書寫段落文字：

```
1 <h1>Hello World!</h2>
2 <p>這是一個文章段落。</p>
```

改寫成 Jade 如下：

```
1 h1 Hello World!
2 p 這是一個文章段落。
```

Jade 的語法就是這麼簡單：只要懂得 HTML5 標籤語法，就可以很直接地將它改寫為 Jade。

Exercise 3: 完整的 HTML5 文件

一個標準的 HTML5 文件，需包含 *doctype*、<html> 標籤、<body> 標籤、<head> 區域等內容。這些同樣都可以用 Jade 來撰寫。

繼續修改 *hello.jade*，將它發展成完整的 HTML5 文件。

views/hello.jade

```
1 1 doctype 5
2 2 html
3 3   head
4 4     title= title
5 5   body
6 6     h1 Hello World!
7 7     p 這是一個文章段落。
```

圖 8-3 *hello.jade* 輸出結果

圖 8-3 是 *hello.jade* 的輸出結果，如果「檢視原始碼」的話，可以看到解析出來的 HTML5 內容：

```
1 <!DOCTYPE html><html><head><title>學習 Jade</title></head>
2 ><body><h1>Hello World!</h1><p>這是一個文章段落。</p></body></ht\
3 ml>
```

Express.js 預設會輸出最小化（Minify）後的 HTML5 文件內

容。要如何使用瀏覽器來瀏覽 *hello.jade* 呢？這就是學習如何新增 URL Routing 的時候了。

解析 **app.js**

要知道如何新增 URL Routing，就要對 *app.js* 有基本的了解。*app.js* 是 Express.js 建立專案時，自動建立的 Web Application 主程式，內容如下：

app.js

```
1  1
2  2 /**
3  3  * Module dependencies.
4  4  */
5  5
6  6  var express = require('express');
7  7  var routes = require('./routes');
8  8  var user = require('./routes/user');
9  9  var http = require('http');
10 10 var path = require('path');
11 11
12 12 var app = express();
13 13
14 14 // all environments
15 15 app.set('port', process.env.PORT || 3000);
16 16 app.set('views', path.join(__dirname, 'views'));
17 17 app.set('view engine', 'jade');
18 18 app.use(express.favicon());
19 19 app.use(express.logger('dev'));
20 20 app.use(express.json());
21 21 app.use(express.urlencoded());
22 22 app.use(express.methodOverride());
23 23 app.use(app.router);
24 24 app.use(express.static(path.join(__dirname, 'public'))\
```

```
25   });
26
27   // development only
28   if ('development' == app.get('env')) {
29     app.use(express.errorHandler());
30   }
31
32   app.get('/', routes.index);
33   app.get('/users', user.list);
34
35   http.createServer(app).listen(app.get('port'), function() {
36     console.log('Express server listening on port ' + \
37 app.get('port'));
38   });
```

程式碼說明如下：

- 第 6~10 行，匯入外部 Node.js 模組，其中 *routes/* 目錄就是存放 URL Routing 程式碼的位置
- 第 12 行，匯入 Express.js 模組
- 第 15~17 行，呼叫 Express.js 的 *set()* 函數來定義常數，當中的 'port' 常數用來定義 Express.js 的 Listening Port 編號
- 第 18~24 行，呼叫 *use()* 函數，來載入（使用）Middleware，Express.js Middleware 的觀念後續再做說明
- 第 32~32 行，這裡就是 URL Routing 的關鍵，呼叫 *get()* 函數來指定 URL 的 Handler Function

以第 32 行為例，當瀏覽器發出 '/' 的請求時（例如：*http://localhost:3000/*），Express.js 就會呼叫（Callback）*routes.index* 函數來做處理。

讓我們來了解一下 *routes.index* 函數的實作：

routes/index.js

```
1 1
2 2 /*
3 3  * GET home page.
4 4  */
5 5
6 6 exports.index = function(req, res){
7 7   res.render('index', { title: 'Express' });
8 8 };
```

這是一個 Node.js 的模組，裡面匯出了 *index* 函數，當 *'/'* 請求發生時，Express.js 就會 Callback 這個函數，並且傳入二個參數：

- *req* 是 Request 物件，存放這此請求的所有資訊
- *res* 是 Response 物件，用來回應該請求

在程式碼第 7 行的地方，呼叫了 *res.render* 函數，這個函數透過 Jade Template Engine 將 *index.jade* 解析（Rendering）成 HTML5 後回應（Response）給用戶端。

Express URL Routing

Express.js 框架基本上，幫助開發者解決了 4 個基本問題：

- URL Routing
- REST API 定義與實作
- Middleware
- Template Engine

首先，以 *hello.jade* 做為例子，練習如何新增 URL Routing。如何讓使用者瀏覽 *http://localhost:3000/hello*，並且讓 Express 解析並輸出 *views/hello.jade* 呢？請跟著以下的步驟進行練習。

Step 1: 新增 *routes/hello.js*

根據前文介紹的觀念，可以很快知道，新增 URL Routing 的第一個步驟就是加入 Handler Function。在 *routes/* 下新增 *hello.js* 檔案：

routes/hello.js

```
1 1
2 2 /*
3 3  * GET home page.
4 4  */
5 5
6 6 exports.index = function(req, res){
7 7   res.render('hello');
8 8 };
```

程式碼第 7 行的地方，表示要解析 *views/* 目錄下的 *hello.jade* 文件。Express.js 是一個 MVC 模式的開發框架，並且 Express.js 的專案已定義好目錄結構，因此請特別注意相關檔案的擺放位置。

Step 2: 修改 *app.js*

以下是修改後的 *app.js* 完整內容：

app.js

```
1  1
2  2 /**
3  3  * Module dependencies.
4  4  */
5  5
6  6 var express = require('express');
7  7 var routes = require('./routes');
8  8 var user = require('./routes/user');
9  9 var http = require('http');
10 10 var path = require('path');
11 11 var hello = require('./routes/hello');
12 12
13 13 var app = express();
14 14
15 15 // all environments
16 16 app.set('port', process.env.PORT || 3000);
17 17 app.set('views', path.join(__dirname, 'views'));
18 18 app.set('view engine', 'jade');
19 19 app.use(express.favicon());
20 20 app.use(express.logger('dev'));
21 21 app.use(express.json());
22 22 app.use(express.urlencoded());
23 23 app.use(express.methodOverride());
24 24 app.use(app.router);
25 25 app.use(express.static(path.join(__dirname, 'public'\
26 26  )));
27 27
28 27 // development only
29 28 if ('development' == app.get('env')) {
30 29   app.use(express.errorHandler());
31 30 }
32 31
33 32 app.get('/', routes.index);
```

```
34 33 app.get('/users', user.list);
35 34 app.get('/hello', hello.index);
36 35
37 36 http.createServer(app).listen(app.get('port'), funct\
38 ion(){
39 37   console.log('Express server listening on port ' + \
40 app.get('port'));
41 38 });
```

這裡修改了二個地方，分別說明如下：

- 第 11 行，匯入 Step 1 的 *hello.js* 模組
- 第 34 行，新增 `/hello` 的 URL Routing 處理

當 Express.js 收到 `/hello` 的請求時，Express.js 就會 Callback *hello.index* 函數，接著 *hello.index* 函數會解析 *hello.jade* 文件，並 Response 給用戶端。

只需要二個很簡單的步驟，就可以新增 URL Routing，這就是 Express.js 框架帶來的便利性之一。

Middleware 的觀念

Middleware 的觀念與插件 (Plugins) 很像，開發者可以在 URL Routing 的「中間」加入一個 Middleware。這個架構可以做到 2 個功能：

- 開發者可以為這個 URL 加入額外的處理程序，也就是 URL Plugins 的觀念
- 更簡潔地控制程式邏輯

URL Plugins 是筆者為了說明 Express Middleware 觀念，自行創造的說法，實際上在正規的軟體架構理論中，並沒有這樣

的說法。URL Plugins 可以讓開發人員「無限量」地為每一個 URL 加入額外的處理程序。

關於上述的處理程序，最典型的應用案例（Use Case）就是使用者系統（User System）。例如，用戶請求 `/admin` 文件時，可以先透過一個 Middleware 來檢查登入狀態，再交給最後的 Handler Function 處理。

Middleware 的實作分為二個部份：

- 針對所有 URL 加入 Middleware，方式是呼叫 `app.use()` 函數
- 針對特定 URL 加入 Middleware，方式是透過 `app.get()` 函數的第二個參數

關於 Middleware 的觀念與實作，將在第 9 章做介紹。

Express.js 應用 - Middleware

第 8 章提到 Express.js 的 Middleware 分為二個部份：所有 URL 與特定 URL。要了解 Middleware 的觀念，最快的方法就是實作「頁面保護」的功能。現在，讓我們為 '/hello' URL 加上密碼 '123456' 的保護。

使用 *app.get()* 撰寫 Middleware

針對特定 URL 加入 Middleware，必須透過 *app.get()* 函數的第二個參數。實作步驟如下。

Step 1: 加入 Middleware

延續上一章的範例，為 '/hello' 加入一個 Middleware：

```
1 app.get('/hello', function (req, res, next) { }, hello.\n2 index);
```

說明如下：

- 第 2 個參數，是一個匿名函數，這個匿名函數就是 '/hello' 的 Middleware
- 第 3 個參數，是原本處理 URL Routing 的函數

Middleware 會收到 3 個參數：

- *req* 是 Request 物件，存放這此請求的所有資訊
- *res* 是 Response 物件，用來回應該請求
- *next* 用來控制流程，後續說明

當使用者輸入密碼時，就要撰寫一段控制流程來處理，典型的控制流程邏輯如下：

```
1  if (password == '123456') {  
2      send_page();  
3  } else {  
4      end_request();  
5  }
```

但 Express.js 並不是用這種方式，來實作流程方式。如何為 `/hello` 加入控制邏輯呢？請看步驟 2。

Step 2: 找出密碼

使用 Query String 來傳入密碼，這是最簡便的方式（也是最糟的做法）。Express.js 可以幫助我們解析 Query String，這是 Express.js 框架的另一個優點。我們就不必像第 3 章介紹的內容一樣，自己去撰寫解析 Query String 的程式碼。

Express.js 將解析好的 Query String 放在 `req.query` 物件裡，現在先直接將它印出。修改程式碼如下：

```
1  app.get('/hello', function (req, res, next) {  
2      console.log(req.query);  
3  }, hello.index);
```

啟動 `app.js` 後，利用瀏覽器開啓 `http://localhost:3000/hello?passwd=123456` 網址。接著，可以在 Console 畫面看到這段訊息：

```
1  Express server listening on port 3000  
2  { passwd: '123456' }  
3  GET /hello?passwd=123456 200 306ms - 135b
```

還有一個可能讓你嚇一跳的現象：瀏覽器的畫面是空白的，而且一直打轉。感覺好像是卡住了，這是怎麼會事兒呢？修改程式碼如下：

```
1 app.get('/hello', function (req, res, next) {  
2     console.log(req.query);  
3     next();  
4 }, hello.index);
```

Express.js 傳入的 *next* 參數，實際上是一個 Lambda。呼叫 *next()* 表示「進行下一個流程」的意思，所謂的下一個流程，當然就是執行 *hello.index* 函數。

加上 *next()* 後，就能在瀏覽器上看到原本的畫面了。所以，我們只要判斷 *req.query.passwd* 就知道密碼是否正確。不過，還有更好的做法，Express.js 的功能可不是只有這樣。

Step 3: 使用內建 Middleware

Express.js 內建幾個好用的 Middleware:

- *basicAuth()*
- *bodyParser()*
- *compress()*
- *cookieParser()*
- *cookieSession()*
- *csrf()*
- *directory()*

像「使用者驗證」這麼常見的「流程」，Express.js 就有提供 *basicAuth()* Middleware。再次修改程式碼如下：

```
1 app.get('/hello', express.basicAuth('jollen', '12345678\  
2 '), hello.index);
```

basicAuth() 使用 HTTP 的方式做認證，並不是 Query String 的做法。只要再次瀏覽網頁，就可以看到一個非常熟悉的畫面，如下圖。



圖 9-1 使用 *basicAuth()* Middleware

同時，在 Node.js 的 Console 也可以看到以下訊息：

```
1 GET /hello 401 10ms
```

這表示 *basicAuth()* 使用的是 HTTP 401 認證方式。

Step 4: 控制流程

將上述的寫法，重構為更清楚的「流程」觀念：

```
1 app.get('/hello', express.basicAuth('jollen', 'abcdef'))\  
2 );  
3 app.get('/hello', hello.index);
```

這其實是較為常見，而且更好的寫法。觀念整理如下：

- 請注意，過去將第 2 個參數解釋為 URL Routing 的 Handler，在這裡則是解釋為 Middleware

- 所以，'/hello' 現在有 2 個 Middleware
- 瀏覽 URL 時，依「順序」來呼叫 Middleware
- 順序指的是程式碼的寫法順序
- *basicAuth()* 先被呼叫
- *basicAuth()* 會進行使用者驗證
- 如果驗證成功，就會呼叫 *next()* 進到下一個「流程」
- 下一個「流程」就是 *hello.index* 函數

這是 Express.js Middleware 的基本觀念，也是初學者必學的主題。

Middleware 與流程控制

Express.js 使用 Middleware 來實作流程控制，我們可以發揮一些巧思，讓流程控制的程式碼邏輯更優雅。比如說，'/hello' 現在的流程是：

1. 使用者驗證
2. 進行一些環境設定的調整
3. 送出頁面

如果使用典型的 *if...else...* 來實作，肯定會寫出很醜的 Node.js 程式碼。反之，利用 Middleware 的觀念來實作，不但觀念簡單，程式碼也更優雅：

```
1 app.get('/hello', express.basicAuth('jollen', 'abcdef'))\  
2 );  
3 app.get('/hello', hello.config);  
4 app.get('/hello', hello.index);
```

完整的 *hello.js* 如下：

hello.js

```
1 1 exports.index = function(req, res, next) {  
2 2   res.render('hello');  
3 3 };  
4 4  
5 5 exports.config = function(req, res, next) {  
6 6   console.log("Do some configs here...");  
7 7   next();  
8 8 };
```

使用者通過驗證後，會進到 *hello.config* 流程。重要的觀念補充如下：

- 在 *hello.config* 裡要記得呼叫 *next()*
- 請注意，以上 2 個匿名函數都加上了第 3 個參數 *next*，成為 Middleware

Express.js Middleware 很像是 URL 的 Plugin，例如上述的範例，可以想像成是在 *'/hello'* 裡，加入 *hello.config* 的插件。

以下是截至目前為止，最新版本的 *app.js*。

app.js

```
1 1 var express = require('express');  
2 2 var routes = require('./routes');  
3 3 var user = require('./routes/user');  
4 4 var http = require('http');  
5 5 var path = require('path');  
6 6 var hello = require('./routes/hello');  
7 7  
8 8 var app = express();  
9 9  
10 10 // all environments
```

```
11 11 app.set('port', process.env.PORT || 3000);
12 12 app.set('views', path.join(__dirname, 'views'));
13 13 app.set('view engine', 'jade');
14 14 app.use(express.favicon());
15 15 app.use(express.logger('dev'));
16 16 app.use(express.json());
17 17 app.use(express.urlencoded());
18 18 app.use(express.methodOverride());
19 19 app.use(app.router);
20 20 app.use(express.static(path.join(__dirname, 'public'\
21  ));
22 21
23 22 // development only
24 23 if ('development' == app.get('env')) {
25 24   app.use(express.errorHandler());
26 25 }
27 26
28 27 app.get('/', routes.index);
29 28 app.get('/users', user.list);
30 29
31 30 app.get('/hello', express.basicAuth('jollen', 'abcde\
32  f'));
33 31 app.get('/hello', hello.config);
34 32 app.get('/hello', hello.index);
35 33
36 34 http.createServer(app).listen(app.get('port'), funct\
37  ion(){
38 35   console.log('Express server listening on port ' + \
39  app.get('port'));
40 36 });
```

目前為止的範例，都是為特定的 URL 來撰寫 Middleware。

使用 ***app.use()*** 撰寫 Middleware

Express.js 的 Middleware，也能針對所有的 URL，方式是使用 *app.use()* 函數。例如，我想為「所有的 URL」加上使用者認證的「流程」，做法非常簡單。以下是修改後的 *app.js*:

app.js

```
1  1 var express = require('express');
2  2 var routes = require('./routes');
3  3 var user = require('./routes/user');
4  4 var http = require('http');
5  5 var path = require('path');
6  6 var hello = require('./routes/hello');
7  7
8  8 var app = express();
9  9
10 10 // all environments
11 11 app.set('port', process.env.PORT || 3000);
12 12 app.set('views', path.join(__dirname, 'views'));
13 13 app.set('view engine', 'jade');
14 14
15 15 app.use(express.favicon());
16 16 app.use(express.logger('dev'));
17 17 app.use(express.json());
18 18 app.use(express.urlencoded());
19 19 app.use(express.methodOverride());
20 20 app.use(express.basicAuth('jollen', '654321'));
21 21 app.use(app.router);
22 22 app.use(express.static(path.join(__dirname, 'public'\
23  )));
24 23
25 24 // development only
26 25 if ('development' == app.get('env')) {
27 26   app.use(express.errorHandler());
```



```
28 27 }
29 28
30 29 app.get('/', routes.index);
31 30 app.get('/users', user.list);
32 31
33 32 app.get('/hello', hello.config);
34 33 app.get('/hello', hello.index);
35 34
36 35 http.createServer(app).listen(app.get('port'), funct\
37 ion(){
38 36   console.log('Express server listening on port ' + \
39 app.get('port'));
40 37 });
```

各位是否能看出當中的細節？說明如下：

- 第 20 行，使用 *app.use()* 來加入 *basicAuth()* Middleware，表示針對所有的 URL
- 第 21 行，為所有 URL 加入了 URL Router，*app.router* 是 Express.js 內建的 URL Router
- 第 20 行與第 21 行，依照「流程」的邏輯，應該是先進入 *basicAuth()* 流程，再到 URL Routing 的流程
- 如上，也就是說，這 2 行的順序是不能修改的，否則就會變成「先做 URL Routing、再做使用者驗證」的錯誤

此外，我們也發現，Express.js 預設加入了這些 Middleware：

```
1 app.use(express.favicon());
2 app.use(express.logger('dev'));
3 app.use(express.json());
4 app.use(express.urlencoded());
5 app.use(express.methodOverride());
6 app.use(app.router);
7 app.use(express.static(path.join(__dirname, 'public')));
```

express.static() 這個 Middleware 是用來指定 “Static Files” 的路徑。典型的 Use Case 是將 Static Files 放在 `public/` 目錄下，比如，瀏覽器送出這個請求：

```
1 GET /style.css
```

當 *app.router* 無法處理這個檔案的 Routing 時，就會進到 *express.static()* 流程，這時 Express.js 會到 *public/* 子目錄下搜尋這個檔案，最後將 *public/style.css* 送出。Static files 一般指的是 CSS、JavaScript、圖片、影片、靜態 HTML 文件等。

Middleware 的「順序」，非常的重要。所有的 Middleware 都是依照寫作順序逐一呼叫。通常，較早先被使用的 Middleware 為 *express.logger()*，這是 Middleware 的 Log System，用來紀錄 HTTP 的請求。

由上述程式碼的順序來看，*express.logger()* 會紀錄下幾乎所有的資訊，包含 HTTP 請求、送出的 Static Files 等。如果不想把 Static Files 放到紀錄訊息裡呢？只要調整其順序即可：

```
1 app.use(express.favicon());
2 app.use(express.json());
3 app.use(express.urlencoded());
4 app.use(express.methodOverride());
5 app.use(app.router);
6 app.use(express.static(path.join(__dirname, 'public')));
7 app.use(express.logger('dev'));
```

這個順序，可以讓 `express.logger()` 紀錄最少量的資訊。

常用的 Express.js Middleware

以下介紹幾個經常使用的 Middleware。

使用 ***compress()***

Express.js 內建 `express.compress()` Middleware，這個 Middleware 可以把 Response Data 壓縮，節省網路頻寬，當然也就縮短 Response Data 所需的時間。修改後的程式碼如下：

```
1 app.use(express.favicon());
2 app.use(express.logger('dev'));
3 app.use(express.compress());
4 app.use(express.json());
5 app.use(express.urlencoded());
6 app.use(express.methodOverride());
7 app.use(express.basicAuth('jollen', '654321'));
8 app.use(app.router);
9 app.use(express.static(path.join(__dirname, 'public')));
```

請注意，`express.compress()` 的順序要放在比較前面。

使用 **cookieParser()**

這是一個用來處理 HTTP Cookies 的 Middleware。它可以協助我們解析 Cookies，並將所有的 Cookies 放在 `req.cookies` 物件 (Key-Value Pairs 格式)。寫法如下：

```
1 app.use(express.cookieParser());
```

比如說，有一個 Cookies 叫做 `purchase_id`，使用這個 Middleware 就可以透過 `req.cookies.purchase_id` 讀取該 Cookies 的值。

使用 **cookieSession()**

提供 Sessions 機制的 Middleware。為 `app.js` 加入 Sessions 功能：

```
1 app.use(express.cookieSession());
```

因為 Express.js 的 Sessions 是建構在 Cookies 的機制之上，所以為了防止 Sessions 被不當修改，可以傳入 `secret` 參數：

```
1 app.use(express.session({  
2   secret: 'N1j2o3l4l5e6n7'  
3 }));
```

此外，也可以設定 Cookies 的屬性：

```
1 app.use(express.session({  
2   secret: 'N1j2o3l4l5e6n7',  
3   cookie: { path: '/', httpOnly: true, maxAge: null }  
4 }));
```

上述設定是原本的預設值。

結論

Node.js + Express.js 初學者，務必了解 Middleware 的觀念，並且學會使用 Middleware 做流程控制。關於 Middleware 的實作，可分為特定 URL 與所有 URL，這是 Express.js 開發的基本功。網路上有一些支援 Express.js 的 Workflow 模組，也都是基於 Middleware 的觀念來實作。

REST API 架構 - 使用 Express.js

REST API 架構（或稱為 RESTful 架構）是實作 Web Service 的一個方法⁹。Web Service 採用 SOA 設計模式，它的精神在於「Server 要以提供服務的形式存在」，至於提供服務的方式，就是對外公開 API。

所以，可以這樣歸納：RESTful 架構是實作 SOA 模式的一個好方法。技術上，RESTful 就是在定義與實作公開（Open）的 API。

很多時候，RESTful 架構與 SOA 是被劃上等號的，但這其實沒有衝突：

- RESTful 是技術上的說法，著重在技術的角度
- RESTful 是定義與實作 Web Service，即 Open API
- SOA 是偏向商業的說法，SOA 所討論的服務（Service）偏向在策略與商業邏輯

REST 的目標就是要支援 Service-Oriented 的架構¹⁰。

在繼續往下閱讀之前，請先了解以下內容：

- 了解圖 2.1（Web Service 導向架構），以及圖 2.1 的重點整理
- 關於 Web Service 技術面的本質，請參考第 6 章的說明
- 有關 REST 的基本概念，請參考 6.2 節的說明。

另外，第 8.5 節提到 Express.js 框架，能幫助開發者解決 4 個基本問題，其中之一就是「REST API 定義與實作」。

⁹<http://www.ibm.com/developerworks/webservices/library/ws-restful/>

¹⁰<http://www.zdnet.com/7-reasons-service-oriented-architecture-and-rest-are-perfect-together-700007706/>

關於 SOA 與 3-Tier 架構

SOA (Service-Oriented Architecture) 是一個熱門名詞，筆者認為，接下來幾年，它將成為 Web App 開發的主流架構。SOA 架構包含 Server 端，也就是 Web Service 的開發。

Node.js 是實作 Web Service 的優秀技術，因此成為 SOA 架構模式的熱門技術之一，許多新創公司紛紛選擇 Node.js 來開發創業作品。知名的團購網站 Groupon，不久前，[也宣佈將網站架構，轉移到 Node.js¹¹](#)。

要了解 SOA 的核心精神，就要由 SOA 的核心精神談起。SOA 本質上是一個軟體設計與架構的模式，也就是 SOA 是一個 Design Pattern。它主要被應用在 Web Service 軟體開發上，並且以 Event-Driven 的觀念，做為程式設計模式。

Event-Driven 的程式設計模式，與 Data-Driven 模式很像；Event-Driven 並不是 Procedure Programming（循序邏輯）的概念，它會比較偏向狀態機（State Machine）的程式邏輯。

實際上，State Machine 是 Event-Driven 的重要實作方法。Node.js + Express.js 開發者，如何引進 State Machine 的寫法呢？這部份後續再做討論。

3-Tier 架構

如何使用 SOA 設計模式，來架構 Web Service 呢？Web Service 的基礎建設，包含 Client 端與 Servier 端，在現在這個行動通訊裝置時代，Client 端也是 Device 端（設備端）。所以這個問題的答案並不難：採用典型的 Multi-Tier 架構。

Multi-Tier 架構又稱為 N-Tier 架構，最常見的就是 3-Tier¹²，三層次架構。3-Tier 將 Web Service 的基礎建設，分為以下 3 層：

- Presenetation Tier

¹¹<https://engineering.groupon.com/2013/node-js/geekon-i-tier/>

¹²http://en.wikipedia.org/wiki/Multitier_architecture

- Logic Tier
- Data Tier

Presentation Tier 是最上層的部份，也就是 User Interface。Logic Tier 進行邏輯處理與運算，例如：條件判斷、執行命令。Data Tier 負責儲存與讀取資料，資料儲放可使用資料庫，或一般的檔案系統。

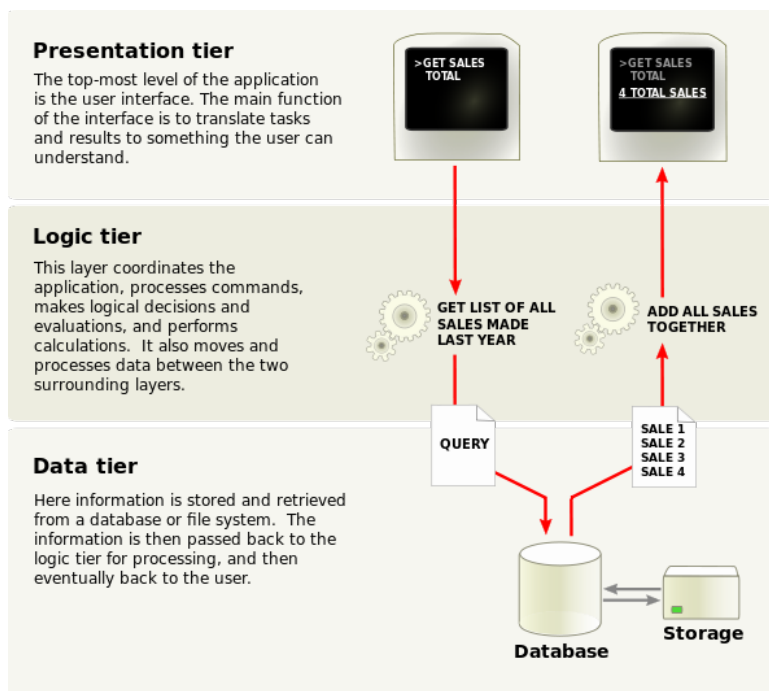


圖 10-1 Three-tiered Application (圖片來源: http://en.wikipedia.org/wiki/Multitier_architecture, 遵循 Public Domain 授權)

3-Tier 架構是 Multi-Tier 架構的一種做法，除了 3 層式架構外，當然還有 4 層式，或是更多層的架構模式。3-Tier 架構的特色，就是把 Presentation Tier 與 Data Tier 完全隔離開來：

- Presentation Tier 無法直接與 Data Tier 溝通

- Logic Tier 負責處理 Data Tier 提供的資料，並搬移到 Presentation Tier

這個觀念，非常像 MVC (Model-View-Controller) 設計模式，實際上 3-Tier 與 MVC 經常被混餉。但 3-Tier 與 MVC 最主要的差異，其實就是上面提到的 2 點。3-Tier 架構中，Presentation Tier 無法直接與 Data Tier 溝通；但 MVC 模式則具備此特性。

關於 MVC 模式的觀念，將在第 11 章，導入 Backbone.js 框架時，再為大家說明。

Presenetation 在 Client 端

有了 3-Tier 與 REST API 架構的觀念後，再重新實作「即時聊天室 App」時，我們會發現整個思惟邏輯都不一樣了。主要的思考重點如下：

- Presentation 儲存在 Client 端，而不是 Server 端
- Express.js 僅負責提供 Presentation 文件，但不進行 Server-Side Rendering
- 透過 REST API 取得 Response Data 後，進行 Client-Side Rendering
- Client 與 Server 透過 API 形式連結，這就是 API Architecture 的觀念

再舉一個例子：實作即時新聞 App。典型的 PHP 開發者，是以「網頁」的概念來架構這個軟體，打造出來的軟體，Scenario 會是這樣：

1. Client 輸入 URL (網址)
2. Server 取得即時新聞
3. Server 將即時新聞 Parse 成 HTML 語法
4. 將完整的 HTML 文件送出

5. Client 端的瀏覽器顯示這份 HTML 文件

但以 REST API 架構觀念重構後，Scenario 應該是：

1. Client 端向 Server 請求 HTML 文件
2. Server 送出模板文件 (Template)
3. Client 端瀏覽器顯示模板文件
4. Client 端呼叫 REST API，向 Server 取得即時新聞
5. Server 端以 JSON 格式返回即時新聞
6. Client 端解析收到的 JSON 資料，並顯示新聞至 UI 上的對應位置 (這個觀念就叫做 ViewModel)

後者才是 REST API 架構的觀念，也是 3-Tier 架構的正確實作方法。這又是 Web Page 與 Web App 的另外一個重要差異：

- Web App 要考慮軟體架構，即 3-Tier 與 API Architecture
- Web Page 只是一份文件，而不是「軟體」

最後，經由以上的討論，可以知道一個很重要的細節：REST API 架構下，Client 與 Server 是經由 API 來連結，Presentation (也就是 HTML5 文件) 的部份，主要是放置在 Client 端。這才是正確的 REST API 架構觀念。當然 API 的定義與實作，並不一定要遵循 REST API 的規範，但重要的觀念是相同的，其通則如下：

- 任何以 API 架構來連接 Client 與 Server 端的實作，都要將 Presentation 放置在 Client 端
- Server 只做運算與服務供應，這就是 Service-Oriented Computing 的觀念
- 在 SOA 模式下，Server 理論上不供應「網頁文件」

思考清楚 REST API、SOA 與 Web Service 的觀念後，就讓我們來重新實作即時聊天室：nodejs-chat。範例可由 Github 取得：

1 <http://github.com/jollen/nodejs-chat>

nodejs-chat 的開發是基於 <http://github.com/jollen/nodejs-express>，因此請在完成前 9 個章節的學習後，再繼續往下閱讀。

Express.js 與 REST API

Express.js 實作 REST API 的方式相當簡單，請參考下表。

表 10-1 REST API 定義

CRUD	HTTP Method	Express.js 實作函數
Create	POST	<code>app.post()</code>
Read	GET	<code>app.get()</code>
Update	PUT	<code>app.put()</code>
Delete	DELETE	<code>app.delete()</code>

Express.js 能支援 CRUD 所需的所有 HTTP Methods。所以，我們只要知道，在實作 URL Routing 時，要如何指定特定的 HTTP Methods 即可。例如：

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

這是先前我們一直採用的寫法，`get()` 函數的另外一個涵義就是 GET HTTP Method。所以，當用戶端以 GET Method 請求 `/'` 這個 URL 時，才會被 Routing 到 `routes.index` 函數。同理，以下的例子：

```
1 app.post('/', routes.index_add);
```

當用戶端以 POST Method 做請求時，就會被 Routing 到 `routes.index_add` 函數。同樣的 URL，可以依據不同的 HTTP Request Method，來做不同的處理（即 CRUD）。

此外，在定義 REST API 時，會需要這樣的 URI 格式：

```
1 /user/{username}
```

比如，要讀取 *jollen* 與 *ellaine* 二位使用者的資訊，就要分別呼叫以下的 API:

```
1 /user/jollen  
2 /user/ellaine
```

這二個 URI 理論上應使用同一個 URL Routing Handler，而不是為這二位使用者，實作獨立的 Routing Handler。例如，以下的寫法並不正確:

```
1 app.get('/user/jollen', user.jollen);  
2 app.get('/user/ellaine', user.ellaine);
```

以 REST API 的角度來說，這是相同的 Resource，同樣的 User Resource，使用同一個 Routing Handler 即可。要解決這個問題，只要將程式碼改寫如下即可:

```
1 app.get('/user/:username', user.index);
```

Express.js 支援以變數 (Variable) 的方式來定義 URI，因此，不管是 */user/jollen* 或 */user/ellaine* 使用者，只要符合 */user/:username* 的格式，都會被 Routing 到 *user.index* 函數。

讀取 URI 的參數

如何在 Routing Handler 裡讀取 URI 裡的變數呢？範例如下:

```
1 exports.index = function(req, res){  
2   res.send("Welcome " + req.params.username);  
3 };
```

URI 的變數值，也視為「參數」的觀念，這些工作 Express.js 都幫我們做好了。如何讀取 `:username` 變數的值呢？只要直接使用 `req.params` 物件即可。以上述範例來說，如果在瀏覽器輸入以下 URI：

```
1 http://localhost:3000/user/jollen
```

就能看到以下的執行結果。從這個結果得知：`:username = jollen`。

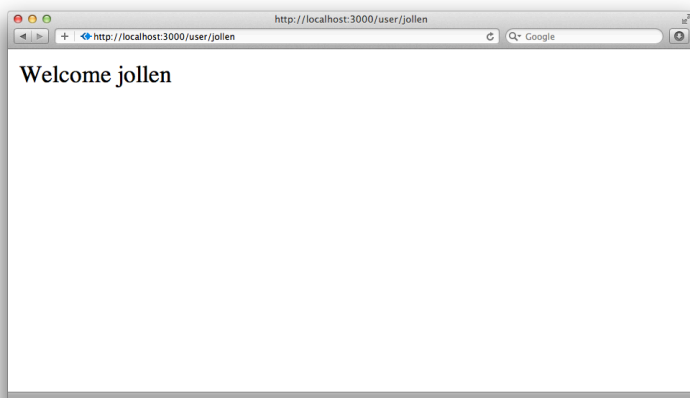


圖 10-2 範例的執行結果

RESTful 架構實務 - 即時聊天室

歸納 REST API 的觀念，得到以下 4 個 RESTful 實作原則¹³：

¹³<http://www.ibm.com/developerworks/webservices/library/ws-restful/>

1. 使用 HTTP Method 實作 CRUD（請參考 6.3 節與表 6-1）
2. Stateless（無狀態模式）
3. 以目錄結構形式定義 URI（請參考 6.1 與 6.2 節）
4. 使用 JSON 做為資料交換格式（請參考 2.12 節與第 4 章）

除了 Stateless 外，其餘所需要的背景知識，都已經做過介紹。關於 nodejs-chat 專案的功能，整理如下。

Nodejs-Chat Server

Server 的部份，應實作以下功能：

- 產生最新訊息的 JSON，並回應給 Client 端應用程式
- 接收 Client 端傳送的訊息，並儲存最新的 n 筆訊息

Nodejs-Chat Client Application

Client Application 的部份，應實作以下功能：

- 輸入即時訊息
- 呼叫 REST API，傳出訊息給 Server
- 經由 WebSocket 接收即時訊息
- 呼叫 REST API，取得最新的 n 則訊息

接下來，請依以下步驟，練習實作 nodejs-chat 專案。

Step 1: 定義 REST API

假設主機名稱為 chatservice.org，接著定義 nodejs-chat 專案的 CRUD 操作，如表 10-2。

表 10-2 REST API 定義

CRUD	HTTP Method	URI 格式
Create	POST	http://chatservice.org/discussion/{message}
Read	GET	http://chatservice.org/discussion/latest/{items}
Update	PUT	無
Delete	DELETE	無

這是一份最低消費的定義：僅支援新增與讀取訊息的功能。
例如，現在要傳送“Hello” 訊息，Client 端應呼叫的 API 為：

```
1 http://chatservice.org/discussion/hello
```

如果要讀取最新 10 筆訊息，應呼叫的 API 為：

```
1 http://chatservice.org/discussion/latest/10
```

請注意幾個重要觀念：

- 儘可能避免使用 Query String
- 不要使用空白字元（Space）
- 全部小寫

此外，REST API 的 URI 類似於目錄結構，所以我們可以這樣解釋：

```
1 $ mkdir discussion
2 $ cd discussion
3 $ mkdir 5d41402abc4b2a76b9719d911017c592
4 $ cd 5d41402abc4b2a76b9719d911017c592
5 $ echo "hello" > db.txt
```

接著，會得以下的 Tree-Structure：

```

1  .
2  └─ discussion
3      └─ 5d41402abc4b2a76b9719d911017c592
4          └─ db.txt

```

上述的數字串，是一個 MD5 編碼，用來做為這筆資料的「ID」。如果第二筆資料，這個樹狀結構會成長成這樣：

```

1  .
2  └─ discussion
3      └─ 5d41402abc4b2a76b9719d911017c592
4          └─ db.txt
5      └─ 7d793037a0760186574b0282f2f435e7
6          └─ db.txt

```

一開始學習 REST API 定義的初學者，經常犯的錯誤，就是誤解了 REST API 的「Resource」意義。比如說，以下這個結構是不好的：

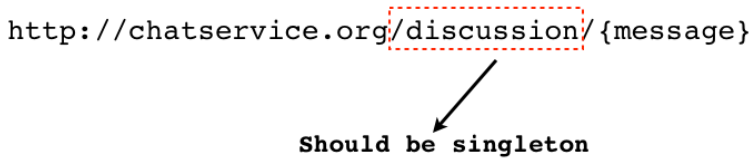
```

1  .
2  └─ discussion1
3      └─ db.txt
4  └─ discussion2
5      └─ db.txt
6  └─ discussion3
7      └─ db.txt

```

REST API 的根節點（root node），也就是 `/discussion`，它是一個資源名稱（Resource）；而這個資源在整個系統裡，應該只有一個（一份）。理論上，root node 要符合 Singleton 設計模式的要求。

`http://chatservice.org/discussion/{message}`



Should be singleton

圖 10-3 Root Node 是 Singleton

Step 3: 實作 URL Routing

修改 `app.js`，加入以下程式碼：

```
1 var discussion = require('./routes/discussion'); // ./r\
2 outes/discussion.js
3
4 app.post('/discussion/:message', discussion.create);
5 app.get('/discussion/latest/:items', discussion.read);
```

參考表 10-1 與表 10-2，得到以下結論：

- 建立訊息要使用 POST Method，因此使用 `app.post()` 函數來建立 URL Routing
- 讀取訊息要使用 GET Method，因此使用 `app.get()` 函數來建立 URL Routing

URL Routing 的實作部份，只需要將 URI 的變數讀出，再放入資料庫即可。由於尚未介紹 Node.js 與資料庫的應用，因此現階段暫時以 Global Array 來存放。`discussion.js` 目前的實作如下：

```
1  /*
2   * URL Routing Handlers
3   */
4
5  exports.create = function(req, res){
6    console.log("CREATE: " + req.params.message);
7  };
8
9  exports.read = function(req, res){
10    console.log("ITEMS: " + req.params.items);
11  };
```

如果要新增 'hello' 訊息，就要呼叫這個 API:

```
1  http://localhost:3000/discussion/hello
```

但是，如果直接使用瀏覽器去開啓的話，會出現「404」找不到網頁的錯誤，為什麼呢？因為瀏覽器是以 GET Method 來做請求，這可以由 Node.js 的 Console 訊息得知：

```
1  $ node app.js
2  Express server listening on port 3000
3  GET /discussion/hello 404 9ms
```

所以，現在我們改用另外一個工具 *curl* 來做測試：

```
1  $ curl -X POST http://localhost:3000/discussion/hello
```

使用 -X 參數即可指定 HTTP Method。測試時，可以看到以下的 Console 訊息：

```
1 $ node app.js
2 Express server listening on port 3000
3 CREATE: hello
4 POST /discussion/hello 200 120013ms
```

程式碼框架完成後，就可以進行實作工作了。以下是 *discussion.js* 的完整程式碼。

routes/discussion.js

```
1 1 /*
2 2  * URL Routing Handlers
3 3  */
4 4
5 5 var history = [];          // 存放訊息
6 6
7 7 exports.create = function(req, res){
8 8   console.log("CREATE: " + req.params.message);
9 9
10 10  // 先打包為 JSON 格式
11 11  var obj = {
12 12      message: req.params.message
13 13  };
14 14
15 15  // 再儲存至 history 陣列
16 16  history.push(obj);
17 17
18 18  // 製作 Response 訊息 (JSON 格式)
19 19  var response = {
20 20      status: "OK"
21 21  }
22 22
23 23  // 回傳 Response 訊息
24 24  res.write(JSON.stringify(response));
25 25  res.end();
```

```
26 26 };  
27 27  
28 28 exports.read = function(req, res){  
29 29   console.log("ITEMS: " + req.params.items);  
30 30  
31 31   // 取出最新的 {req.params.items} 筆訊息  
32 32   var latest = history.slice(0 - req.params.items);  
33 33  
34 34   // 回傳  
35 35   res.write(JSON.stringify(latest));  
36 36   res.end();  
37 37 };
```

可以利用以下的連續命令進行測試:

```
1 $ curl -X POST http://localhost:3000/discussion/hello1  
2 $ curl -X POST http://localhost:3000/discussion/hello2  
3 $ curl -X POST http://localhost:3000/discussion/hello3  
4 $ curl -X GET http://localhost:3000/discussion/latest/2  
5 $ curl -X GET http://localhost:3000/discussion/latest/1  
6 $ curl -X POST http://localhost:3000/discussion/hello4  
7 $ curl -X POST http://localhost:3000/discussion/hello5  
8 $ curl -X GET http://localhost:3000/discussion/latest/3  
9 $ curl -X GET http://localhost:3000/discussion/latest/5
```

以下是相對應的 Console 訊息:

```
1 $ node app.js
2 Express server listening on port 3000
3 CREATE: hello1
4 POST /discussion/hello1 200 30ms
5 CREATE: hello2
6 POST /discussion/hello2 200 2ms
7 CREATE: hello3
8 POST /discussion/hello3 200 1ms
9 ITEMS: 2
10 GET /discussion/latest/2 200 1ms
11 ITEMS: 1
12 GET /discussion/latest/1 200 1ms
13 CREATE: hello4
14 POST /discussion/hello4 200 2ms
15 CREATE: hello5
16 POST /discussion/hello5 200 0ms
17 ITEMS: 3
18 GET /discussion/latest/3 200 1ms
19 ITEMS: 5
20 GET /discussion/latest/5 200 9ms
```

一切就緒後，就可以開始製作 Front-end 了。在這之前，可以開始試著建立測試程式，並加入簡單的 Test Case。否則，等到 API 數量更多，以及 Use Scenario 更複雜時，測試工作就更費勁了；因為將會有更多、更複雜的測試命令等著我們 Key In。

Step 4: 實作 Test Case

Test Case 是軟體開發很重要的一環，上述以 *curl* 指令進行測試，基本上這是比較不正式的做法。如果能撰寫一些測試程式，未來在開發的過程，就能很方便而且快速地測試新版本軟體。Test Case 一般是在軟體的開發過中，不斷地累積，並且隨著我們的專案釋出；因此，一開始就養成撰寫 Test Case 的習慣，絕對會比事後補上更好。

筆者建議持續以 Node.js (JavaScript) 來為 nodejs-chat 專案開發 Test Case。要將上個步驟的 *curl* 指令，取代為 Test Case，就要知道如何以 Node.js 來製作 HTTP Client：方式是使用 Node.js 的 *http* 模組。不過，還有更簡便的解決方案：使用 [Requestify](http://ranm8.github.io/requestify/)¹⁴。

開啓 nodejs-chat 專案下的 *package.json* 檔案，加入 *requestify* 模組：

```
1 {
2   "name": "application-name",
3   "version": "0.0.1",
4   "private": true,
5   "scripts": {
6     "start": "node app.js"
7   },
8   "dependencies": {
9     "express": "3.4.4",
10    "jade": "*",
11    "requestify": "*"
12  }
13 }
```

安裝相依模組：

```
1 $ npm i
```

新增 *tests/* 字目錄，用來存放 Test Case：

```
1 $ mkdir tests
2 $ cd tests
```

在 *test/* 目錄下建立第一個測試程式，將檔案命名為 *01-test-discussion-create.js*；*requestify* 的使用方式非常簡單，完整程式碼如下。

¹⁴<http://ranm8.github.io/requestify/>

tests/01-test-discussion-create.js

```
1 1 var requestify = require('requestify');
2 2
3 3 requestify.post('http://localhost:3000/discussion/hel\
4 lo', {
5 4   // POST 的 Request Body, 例如: 訊息夾帶的圖片。目前為空。
6 5 })
7 6 .then(function(response) {
8 7   // 取得 Response Body
9 8   response.getBody();
10 9 });
```

上述步驟，說明了實作 nodejs-chat 的重要觀念，完整的 REST API 實作，請參考完整的範例程式。關於 nodejs-chat 的完整程式碼，可由 <http://github.com/jollen/nodejs-chat> 取得。這個範例僅只是簡單的 Test Case，各位可以試著自行撰寫更多的測試程式，並設計更完整的 Test Case。

結論

學習 REST API 的定義，以及如何利用 Express.js 框架來實作 API，是本章的重點。關於 Client Application 的製作，將會需要討論幾個觀念：

- MVC 設計模式的使用
- 使用 jQuery 呼叫 REST API
- 使用 Backbone.js 呼叫 REST API（又稱為 Backbone Way）

Client Application 的製作，將於第 11 章另行說明。

REST API 應用 - 使用 jQuery

在上一章的 nodejs-chat 專案裡，我們已經實作出基本的 REST API，接下來就是 Client Application 的製作。RESTful 是一種彈性非常大的架構，它讓 Client 端能以更簡單的方式，來整合 Web Service。

使用 jQuery 來製作 Client Application 時，「將不考慮 MVC 架構模式」。為什麼要使用 MVC 模式？以及，為什麼要導入 Backbone.js 呢？相信經由本章的範例製作過程，就可以幫助初學者建立另一個重要的思惟：使用 MVC 架構模式是必要的。

呼叫 REST API - 使用 jQuery

上一章也提到，從 REST API 的角度來看，Presentation 的部份是位於 Client 端。Presentation 就是 Client 的網頁（UI）。如果把 Client Application 的觀念延伸到手機 App，Presentation 的觀念就更清楚了：

- 可以使用 Android SDK 來製作 Client Application，此時 Presentation 可以使用 Android 的 View 組件，或是以 HTML5 來製作
- 同理，Client Application 也可以使用 iOS SDK 來製作，當然也可以是 iOS SDK + HTML5

從這個角度來看，Presentation 自然是位於 Client 端。

以下以幾個步驟，來製作一個簡單 Client Application。

Step1: 建立專案

在 nodejs-chat 專案裡建立 *client/* 目錄，將主要的 HTML5 文件命名為 *chat.html*。同時建立基本的目錄結構：

```
1 .
2 |└─ chat.html
3 |└─ images
4 |└─ javascripts
5 |└─ stylesheets
```

接著使用 Bootstrap 3 來建立 UI，並且使用 jQuery 來呼叫 REST API。將 Bootstrap 3 與 jQuery 安裝至專案目錄下：

```
1 .
2 |└─ chat.html
3 |└─ images
4 |└─ javascripts
5 |   |└─ app.js
6 |   |└─ bootstrap.min.js
7 |   |└─ jquery.min.js
8 |   |└─ jquery.websocket.js
9 |└─ stylesheets
10 |└─ bootstrap.min.css
```

Step 2: 製作 UI

Bootstrap 是一個簡單易用的 CSS Framework，也包含字型 (Typographics)。目前已經是非常受歡迎的 Web App 製作框架了。除了以文字編輯器，自行撰寫 Bootstrap 網頁外，網路上有一些所見即所得的 Layout 工具。方便起見，筆者使用 [Layoutit](http://www.layoutit.com)¹⁵ 規劃了一個非常簡單的 UI，如圖 11-1。

¹⁵<http://www.layoutit.com>

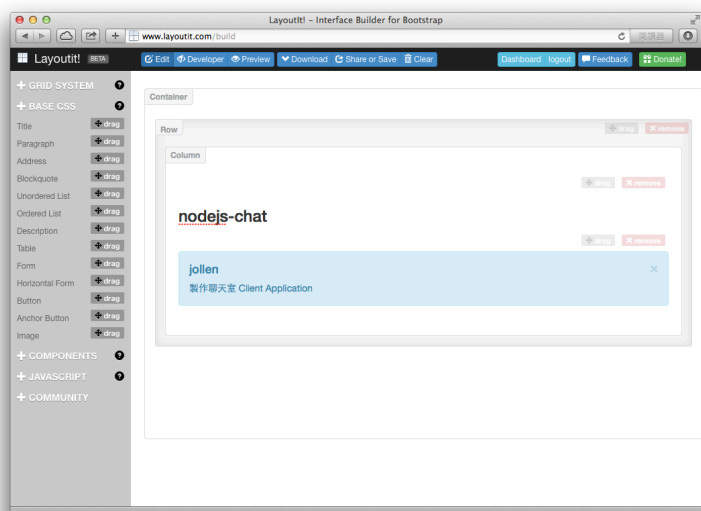


圖 11-1 製作 Client Application UI

將 Layoutit 產生的標籤剪貼至 *chat.html*。完整的 *chat.html* 內容如下：

client/00_chat.html

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <link rel="stylesheet" href="stylesheets/bootstrap.min.\
5 css" />
6 <script type='text/javascript' src="javascripts/jquery.\
7 min.js"></script>
8 <script type='text/javascript' src="javascripts/bootstr\
9 ap.min.js"></script>
10 <script type='text/javascript' src="javascripts/jquery.\
11 websocket.js"></script>
12 <script type='text/javascript' src="javascripts/app.js"\
13 ></script>

```

```
14 </head>
15
16 <body>
17
18 <div class="container">
19   <div class="row clearfix">
20     <div class="col-md-12 column">
21       <h3>
22         nodejs-chat
23       </h3>
24       <div class="alert alert-dismissable alert-i\
25 nfo">
26         <button type="button" class="close" da\
27 ta-dismiss="alert" aria-hidden="true">×</button>
28         <h4>
29           jollen
30         </h4> 製作聊天室 Client Application
31       </div>
32     </div>
33   </div>
34 </div>
35
36 </body>
37 </html>
```

Step 3: 讀取最新訊息

上一章，我們利用了 *curl* 命令來測試 `*/discussion/latest/`

```
1 $ curl -X GET http://localhost:3000/discussion/latest/3
2 [{"message":"hello3"}, {"message":"hello4"}, {"message":"\
3 hello5"}]
```

現在要將這個測試過程，撰寫成實際的程式碼。因此需要撰寫 JavaScript 程式碼，來呼叫 REST API；我們可使用現有

的 JavaScript 程式庫，來進行這項工具。有二個廣受歡迎的 JavaScript 程式庫：

- jQuery
- Backbone.js

採用 jQuery 是簡單方便的方式，初學者可以先使用 jQuery 程式庫來實作 Client Application；感受一下 RESTful 架構的精神。Backbone.js 則是一個 MVC 架構的實作框架，它可以讓 Client Application 的程式碼結構更清楚、易於維護與擴充。

使用 jQuery 的 *ajax* 函數來呼叫 REST API。開啓 app.js，加入以下程式碼：

client/01_app.js

```
1 // 等候 HTML 文件完成載入
2 $(document).ready(function(){
3     initSubmitForm();
4 });
5
6 var initSubmitForm = function() {
7     // 使用 ajax() 來呼叫 REST API
8     $.ajax({
9         url: 'http://localhost:3000/discussion/latest/3\'
10     ',
11     type: "GET",
12     dataType: "json",
13     complete: function(data, textStatus, jqXHR) {
14         console.log(textStatus);
15     },
16     success: function (data, textStatus, jqXHR) {
17         console.log(data);
18     }
19     });
```

```
20  
21     return false;  
22 };
```

關於 *ajax()* 的重點說明如下：

- *ajax()* 的 *success* 屬性是一個 Callback Function，我們可以在這個 Callback Function 的第 1 個參數取得傳回值。
- 在 API 呼叫完成後，則是 Callabck *complete*，可以在這個 Callback Function 的第 2 個參數，取得狀態值（成功或失敗）。

可以先使用 Firebug 進行測試，如圖 11-2。觀察 Node.js 的 Console 訊息，也可以知道 Client Application 是否有成功呼叫 REST API。

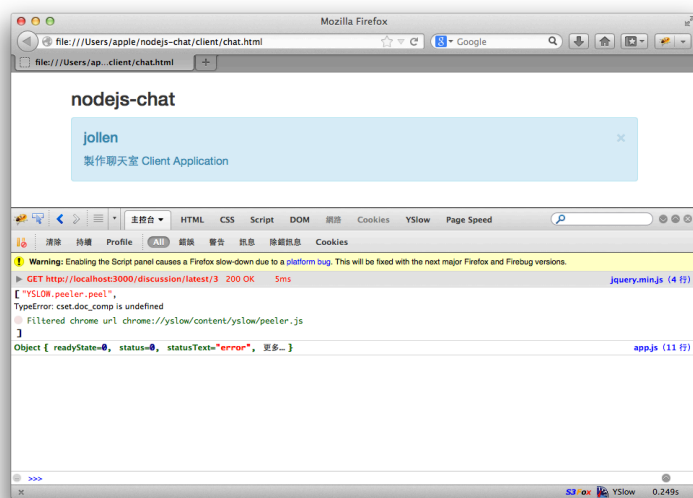


圖 11-2 測試 Client Application

Step 4: 處理 JSON 資料

接著，遇到一個小問題：如何將 REST API 回傳的 JSON 資料，轉換成 HTML5 標籤，並顯示在 UI 上。

要解決這個問題，就需要一個 MVC 框架，這類框架能提供以下的重要功能：

- 製作 HTML5 Template
- 將 JSON 資料「套用」到 Template

下一章將正式引用 Backbone.js 來優化 Client Application。目前，暫時使用 Dirty 的方式來進行；也藉此感受 MVC 架構的優點。

先從 *chat.html* 裡，將顯示留言的 HTML5 片段取出，並修改 *app.js* 加入 *dataMapping()* 函數，將 JSON 裡的資料「對應」到 HTML5 裡。完整的 *app.js* 如下：

client/app.js

```
1  // 等候 HTML 文件完成載入
2  $(document).ready(function(){
3      initSubmitForm();
4  });
5
6  var initSubmitForm = function() {
7      // 使用 ajax() 來呼叫 REST API
8      $.ajax({
9          url: 'http://localhost:3000/discussion/latest/3\
10 ',
11          type: "GET",
12          dataType: "json",
13          complete: function(data, textStatus, jqXHR) {
14              console.log(textStatus);
15          },
```

```

16         success: function (data, textStatus, jqXHR) {
17             dataMapping(data);
18         }
19     });
20
21     return false;
22 };
23
24 var dataMapping = function(data) {
25     for (i = 0; i < data.length; i++) {
26         var htmlCode =
27             "<div class=\"alert alert-dismissable alert\
28 -info\">"
29             + "        <button type=\"button\" class=\"clo\
30 se\" data-dismiss=\"alert\" aria-hidden=\"true\">×</but\
31 ton>"
32             + "        <h4>jollen</h4>"
33             + data[i].message
34             + "</div>";
35
36         $('#message').append(htmlCode);
37     }
38 }

```

再修改原始的 *chat.html* 文件如下：

- 刪除測試訊息
- 在原來顯示訊息的位置，加入一個名為 *message* 的 `<div>` 區塊
- 後續會以 jQuery 將取得的訊息，放進上述區塊

完整的 *client/chat.html* 內容如下：

client/chat.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <link rel="stylesheet" href="stylesheets/bootstrap.min.\
5  css" />
6  <script type='text/javascript' src="javascripts/jquery.\
7  min.js"></script>
8  <script type='text/javascript' src="javascripts/bootstr\
9  ap.min.js"></script>
10 <script type='text/javascript' src="javascripts/jquery.\
11 websocket.js"></script>
12 <script type='text/javascript' src="javascripts/app.js"\
13 ></script>
14 </head>
15
16 <body>
17
18 <div class="container">
19     <div class="row clearfix">
20         <div class="col-md-12 column">
21             <h3>
22                 nodejs-chat
23             </h3>
24             <div id="message"></div>
25         </div>
26     </div>
27 </div>
28
29 </body>
30 </html>
```

測試前，先利用 *curl* 寫入幾筆訊息：


```
1 $ curl -X POST http://localhost:3000/discussion/你好
2 {"status":"OK"}
3 $ curl -X POST http://localhost:3000/discussion/測試中
4 {"status":"OK"}
5 $ curl -X POST http://localhost:3000/discussion/Hello
6 {"status":"OK"}
```

請注意，使用 Firefox 可能無法成功取得 REST API 的回傳資料。如果有問題，請先使用其它瀏覽器。以下是使用 Safari 7.0 的測試畫面。

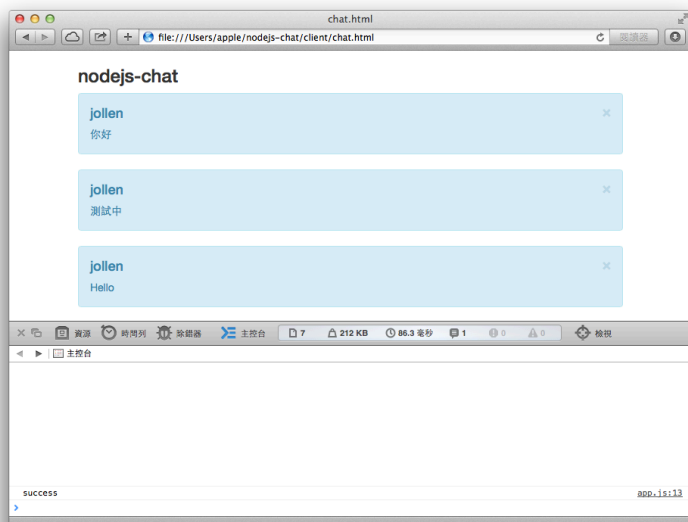


圖 11-3 使用 Safari 7.0 測試 Client Application

到這裡，我們已經完成一個重要的里程碑了：整合 REST API 與 Client Application。

經由上述的範例，我們看出了幾個問題：

- JavaScript 與 HTML5 混雜在一起，維護或擴充都是個大

問題

- 需要一個良好的框架，讓我們可以解決上述問題，Backbone.js 就是一個不錯的解決方案
- 需要一個更簡單的方式，將 JSON 資料「對應」到 HTML5 裡，同樣 Backbone.js 可以幫助我們解決這個問題

Step 5: 說明 MIME Types

使用 `ajax()` 時，我們指定 `dataType` 屬性的值為 “json”：這代表期望 Server 回傳的資料是 JSON 格式。為了讓 Server 在回傳資料時，能「明確」告知「資料類型」，所以最好能加入「Content-Type」的 HTTP Header。

例如，以下是目前的 REST API 回應檔頭：

```
1 $ curl -I -X GET http://localhost:3000/discussion/lates\
2 t/3
3 HTTP/1.1 200 OK
4 X-Powered-By: Express
5 Date: Thu, 05 Dec 2013 05:45:25 GMT
6 Connection: keep-alive
7 Transfer-Encoding: chunked
```

最好可以加入一行「Content-Type」檔頭如下：

```
1 $ curl -I -X GET http://localhost:3000/discussion/lates\
2 t/3
3 HTTP/1.1 200 OK
4 X-Powered-By: Express
5 Content-Type: application/json
6 Vary: Accept-Encoding
7 Date: Thu, 05 Dec 2013 05:45:05 GMT
8 Connection: keep-alive
9 Transfer-Encoding: chunked
```

“Content-Type”是標準的 HTTP Header，用來說明回應的資料類型。資料類型稱為 [MIME types](#)¹⁶，根據 [RFC 4627](#)¹⁷ 的定義，JSON 的 MIME types 字串為 “application/json”。

要回應 HTTP Header，只需要呼叫 `writeHead()` 函數即可：

```
1 response.writeHead(200, {"Content-Type": "application/j\
2 son"});
```

修改 `routes/discussion.js` 如下：

`routes/discussion.js`

```
1  /*
2   * URL Routing Handlers
3   */
4
5  var history = [];          // 存放訊息
6
7  exports.create = function(req, res){
8    console.log("CREATE: " + req.params.message);
9
10   // 先打包為 JSON 格式
11   var obj = {
12     message: req.params.message
13   };
14
15   // 再儲存至 history 陣列
16   history.push(obj);
17
18   // 製作 Response 訊息 (JSON 格式)
19   var response = {
20     status: "OK"
```

¹⁶http://en.wikipedia.org/wiki/Internet_media_type

¹⁷<http://tools.ietf.org/html/rfc4627>

```
21   }
22
23   // HTTP 檔頭
24   res.writeHead(200, {"Content-Type": "application/json\
25   "});
26
27   // 回傳 Response 訊息
28   res.write(JSON.stringify(response));
29   res.end();
30 };
31
32 exports.read = function(req, res){
33   console.log("ITEMS: " + req.params.items);
34
35   // 取出最新的 {req.params.items} 筆訊息
36   var latest = history.slice(0 - req.params.items);
37
38   // HTTP 檔頭
39   res.writeHead(200, {"Content-Type": "application/json\
40   "});
41
42   // 回傳
43   res.write(JSON.stringify(latest));
44   res.end();
45 };
```

如果想達到即時性 (Real-Time) 的功能呢? 只需要再加入 WebSocket 的機制即可:

- Client Application 開啓時, 與 Server 建立 WebSocket 連線
- Server 收到新訊息時, 透過 WebSocket 發出通知 (Notification) 給 Client Application
- Client Application 呼叫 REST API 取得最新資訊

認識 Key-Value Pairs 觀念

Backbone.js 如何解決上一節 Step 4 所提到的問題呢？分述如下：

- 宣告 HTML5 Template: Backbone.js 內建 Underscore，我們可以使用 Underscore 來宣告 Template
- 將 JSON「套用」到 Template: 使用 Key-Value Pairs 來做對應

JSON 的資料格式，又稱為是“Key-Value Pairs”的格式，例如：

```
1 { "name": "jollen" }
```

與 Key-Value Pairs 的關係說明如下：

- *name* 是“Key”，又可視為「變數」
- *jollen* 是“Value”，也就是 *name* 這個 Key 的值

所以，在製作 Template 時，只要把“Key”當做變數的觀念，到時就可以直接「對應」了。例如：

```
1 <div>
2   <p>${name}</p>
3 </div>
```

並且有一份 JSON 資料如下：

```
1  [  
2      { "name": "Peter" },  
3      { "name": "Paul" },  
4      { "name": "John" }  
5  ]
```

此時，Template Engine 就能直接把 Template 和 JSON 對應成以下的結果：

```
1  <div>  
2    <p>Peter</p>  
3    <p>Paul</p>  
4    <p>John</p>  
5  </div>
```

這麼方便的做法，是導入 Backbone.js 的第一個目的。這種將以 JSON 來表示 Key-Value Pairs 資料，並將 Key-Value Pairs 對應到 HTML5 Template 的觀念，就是一個被稱為「ViewModel」的觀念。ViewModel 能解決的問題不止於此，這只是 View Model 觀念的一部份。

ViewModel 可以強化 MVC 模式中的 C (Controller)，因此把它叫做 MVVM 模式。[MVVM 模式](#)¹⁸是由微軟所提出的新概念，MVC & MVVM 也經常被放在一起討論。

從軟體工程的角度，把 MVC 架構模式加入了 Key-Value Pairs 觀念，可以達到以下目的：

- 讓 MVC 模式更為完整
- 用 View Model 來串連 Model 與 View，而不是用 Controller
- 如上，儘可能讓 View 與程式碼無關，又稱為 Code Ignorance

¹⁸http://en.wikipedia.org/wiki/Model_View_ViewModel

- 如上，讓 UI 設計是 Code Ignorance：讓做 UI 不用寫程式

簡單說，從上述的例子可以發現，UI 設計師只需要把 HTML5 Template 設計好即可，不需要寫程式，就可以將 Server 回傳的資料，顯示在畫面上。

結論

學習 MVC 架構模式，先了解它的必要性，會有很大的幫助。如果 MVC 架構模式，在 HTML5 開發方面並沒有太大的必要性，或許可以忽略不學。但經由本章的開發思惟，可以了解到 MVC 架構模式的必要性。

從第 10 章與第 11 章的開發過程中，還可以了解 REST API 的精神，以及它和 Client 端的整合觀念。在本章的最後，透過 Client Application 的開發過程，我們總結一件重要的事情：MVC 與 MVVM 架構模式是必要的。在第 12 章，將針對 MVC 架構模式做討論，也將正式引入 Backbone.js 框架。第 12 章開始，將進入 Frontend 的領域，也是本書「入門篇」的最後一個章節。

本章範例可由 <http://github.com/jollen/nodejs-chat> 取得。

MVC 架構實作 - Backbone.js 入門

現代的 Web App 開發，是以 Web Service 為主軸，以 API 的方式整合 Client (Device) 與 Server。API 為主軸的架構中，又以 RESTful 最受到矚目，RESTful 將可能成為是未來的 Web Service 主流架構。RESTful 易於 Web Service 與不同裝置的整合，例如：Desktop、Phone、Tablet 與 TV 等

前一個章節，我們學到幾個重要觀念：

- 以 JSON 做為主要的資料交換格式
- Client 端使用樣板 (Client-side HTML Template)
- Key-Value Pair

有了這幾個觀念後，就能開始導入 Backbone 框架，並且將 nodejs-chat 發展的更完善。首先，先為 `/discussion/:message` API 製作一個輸入介面，並且採用 Backbone 框架來呼叫 REST API。

初學 Backbone Way

Step 1: 安裝 Underscore 與 Backbone

先到 Underscore 網站下載 `underscore-min.js`:

1 <http://underscorejs.org/underscore-min.js>

再到 Backbone 網站下載 `backbone-min.js`:

1 <http://backbonejs.org/backbone-min.js>

將上述檔案存放至 nodejs-chat 專案的 *client/javascripts/* 目錄下。Backbone 必須與 Underscore 搭配使用。

Step 2: 修改 *chat.html*

加入以下二行：

```
1 <script type='text/javascript' src="javascripts/undersc\  
2 ore-min.js"></script>  
3 <script type='text/javascript' src="javascripts/backbon\  
4 e-min.js"></script>
```

將 Underscore 與 Backbone 加入 Frontend 頁面。幾個重要事項：

- 順序不能改變，必須先引入 Underscore 後，再引入 Backbone 才不會出錯
- Backbone 是一個 MVC 框架，其中 View 的部份由 Underscore 支援

Step 3: 重新撰寫 *client/javascripts/app.js*

將原本 *client/javascripts/app.js* 裡的内容全數刪除，並加入以下程式碼：

```
1  /**
2   * SETUP
3   **/
4   var app = app || {};
5
6  /**
7   * MODELS
8   **/
9
10
11 /**
12  * VIEWS
13  **/
14  app.MessageView = Backbone.View.extend({
15    events: {
16    },
17  });
18
19 /**
20  * BOOTUP
21  **/
22  $(document).ready(function() {
23    app.messageView = new app.MessageView();
24  });
```

這是撰寫 Backbone 的起點，幾個重要觀念解說如下：

- Backbone 是一個 MVC 框架，一開始先定義 Model 與 View
- Model 就是「表示資料的模型」，也就是將會顯示在畫面上的資料
- View 的部份將負責處理 Template 與 Model 的對應，必須先了解 Key-Value Paris 觀念
- View 的部份，也負責處理控制的部份，例如：Button 的 click 事件

Step 4: 文字輸入欄位與按鈕

在 *chat.html* 裡加入文字輸入欄位與按鈕。使用到 Bootstrap 的 [Grid System](#)¹⁹ 與 [Buttons](#)²⁰ 樣式。

以下是 *chat.js* 的完整內容：

client/chat.html

```
1  1 <!DOCTYPE html>
2  2 <html>
3  3 <head>
4  4 <link rel="stylesheet" href="stylesheets/bootstrap.m\
5  in.css" />
6  5
7  6 <script type='text/javascript' src="javascripts/jque\
8  ry.min.js"></script>
9  7 <script type='text/javascript' src="javascripts/boot\
10 strap.min.js"></script>
11 8 <script type='text/javascript' src="javascripts/jque\
12 ry.websocket.js"></script>
13 9 <script type='text/javascript' src="javascripts/unde\
14 rscore-min.js"></script>
15 10 <script type='text/javascript' src="javascripts/back\
16 bone-min.js"></script>
17 11 <script type='text/javascript' src="javascripts/app.\
18 js"></script>
19 12 </head>
20 13
21 14 <body>
22 15
23 16 <div class="container">
24 17     <div class="row">
25 18         <div class="col-md-9">
```

¹⁹<http://getbootstrap.com/css/#grid>

²⁰<http://getbootstrap.com/css/#buttons>

```
26 19                                     <input class="form-control" type="text" name="
27 sage">
28 20                                     </div>
29 21                                     <div class="col-md-3">
30 22                                     <button class="btn btn-large btn-primary btn
31 age-save">送出</button>
32 23                                     </div>
33 24                                     </div>
34 25     <div class="row clearfix">
35 26         <div class="col-md-12 column">
36 27             <h3>
37 28                 nodejs-chat
38 29             </h3>
39 30             <div id="message"></div>
40 31         </div>
41 32     </div>
42 33 </div>
43 34
44 35 </body>
45 36 </html>
```

重點說明如下：

- 第 19 行：給輸入欄位的 *name* 是 “message”，後續將以 jQuery 選擇器搭配 *name* 屬性來找出這個物件
- 第 22 行：額外給予這個 `<button>` 元件一個 class name ‘btn-message-save’
- 不使用典型的 `<form>` 表單方式來處理「送出」的動作，將使用 Backbone 的事件處理框架

Step 5: Backbone.View 事件處理

對初學者來說，Backbone 似乎是一個堅深難懂的 MVC 框架，但如果從觀念的角度來討論，Backbone 不但變的簡單易學，

而且也能感受到它的強大威力。筆者認為，學習 Backbone 的第一個步驟，應該是了解 `Backbone.View.extend`²¹ 的觀念。

在 `chat.html` 裡有一個 `<div>` 區塊，包含輸入欄位與按鈕，這裡是單純的 HTML5 語法，在瀏覽器裡開啓後，可以看到圖 12.1 的畫面。

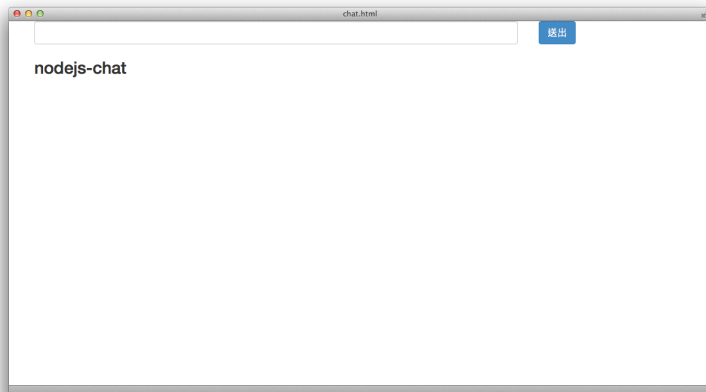


圖 12-1 `chat.html` 瀏覽畫面

按下按鈕後，要進行一些動作，這個部份稱為 Logic。因此，View 可以再細分為二個觀念：

- View: 單純眼睛看到的畫面
- Logical View: 隱藏在畫面下的控制邏輯，通常是程式碼的部份

簡單說，`Backbone.View`²² 就是用來表示 Logical View 的元件。更白話一點，就是 View 裡面的程式碼。

所以，現在我們想要為以下的 `<div>` 區塊，加入 Logic：

²¹<http://backbonejs.org/#View-extend>

²²<http://backbonejs.org/#View>

```
1         <div class="row">
2             <div class="col-md-9">
3                 <input class="form-control" type="text" name="message">
4 e">
5             </div>
6             <div class="col-md-3">
7                 <button class="btn btn-large btn-primary btn-mes
8 -save">送出</button>
9             </div>
10        </div>
```

就要將這個區塊，表示成 Backbone.View。先為這個區塊加入名字：

```
1         <div class="row" id="message-save">
2             <div class="col-md-9">
3                 <input class="form-control" type="text" name="message">
4 e">
5             </div>
6             <div class="col-md-3">
7                 <button class="btn btn-large btn-primary btn-mes
8 -save">送出</button>
9             </div>
10        </div>
```

加入「id="message-save"」表示這個區塊叫做“message-save”。接著修改 *client/javascripts/app.js* 如下：

```
1 1 app.MessageView = Backbone.View.extend({
2 2   el: '#message-save',
3 3   events: {
4 4     'click .btn-message-save': 'save'
5 5   },
6 6   save: function() {
7 7     alert("Saving...");
8 8   }
9 9 });
```

重要觀念說明：

- 第 1 行：呼叫 *Backbone.View.extend()* 來宣告（即擴充出）一個新的 Logical View，名字叫做 *MessageView* 並存放於 *app* 物件裡
- 第 2 行：*el* 欄位是“Element”的意思，指定這個 Logical View 的 View
- 第 3 行：*events* 欄位用來定義 Logical View 裡的事件觸發函數
- 第 4 行：當 *.btn-message-save* 元件的‘click’事件觸發時，呼叫 *save()* 函數，*.btn-message-save* 是一個 Button，使用者按下 Button 時，*save()* 函數就會被呼叫；Backbone.View 的做法，取代了典型的 `<form>` 與 Submit Button
- 第 7 行：測試程式碼，按下 Button 會出現圖 12.2 的畫面

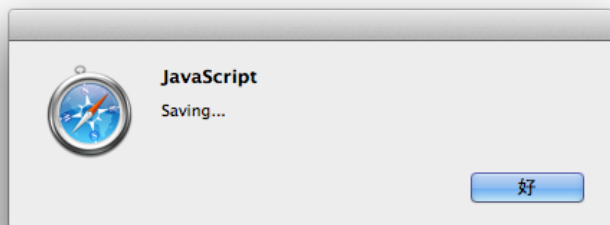


圖 12-2 測試畫面

Step 6: 呼叫 REST API

最後一個步驟：

- 讀取輸入欄位的值
- 呼叫 `/discussion/:message` 儲存訊息

使用 jQuery 來呼叫 REST API 比較容易理解，目前先暫時採用這個寫法。實作上述的 `save()` 函數如下：

```
1  app.MessageView = Backbone.View.extend({
2    el: '#message-save',
3    events: {
4      'click .btn-message-save': 'save'
5    },
6    save: function() {
7      var message = $('input[name="message"]').val();
8
9      $.ajax({
10        url: '/discussion/' + message,
11        type: 'POST',
```



```
12         dataType: "json",
13         success: function (data, textStatus, jqXHR)\
14     {
15         alert(" 已儲存成功");
16     },
17         complete: function (data, textStatus, jqXHR\
18 ) {
19     }
20     });
21 }
22 });
```

測試時，請記得透過 Web Server 方式來開啓 *chat.html*，直接開啓「檔案」的話，*jQuery.ajax()* 將無法成功呼叫 REST API。可以修改 Express 的主程式－*app.js*，將“static”的目錄改為 *client/*：

```
1 app.use(express.static(path.join(__dirname, 'client')));
```

並瀏覽以下網址：

```
1 http://localhost:3000/chat.html
```

輸入訊息“hello”並按「送出」後，可以看到以下的 Node.js Console 訊息：

```
1 CREATE: hello
2 POST /discussion/hello 200 4ms
```

雖然測試後沒有什麼問題，不過感覺好像只是把 jQuery 的程式碼，用 Backbone 框起來而已。當然，這還不是正確的 Backbone 做法；所以，接下來將開始重構 *save()* 程式碼。

認識 *View.\$el*

*View.\$el*²³ 是 Backbone 初學者的第二個關卡。根據 Backbone 官方文件的說明，*View.\$el* 的用途為：

A cached jQuery object for the view's element.

簡單說，這就是一個 jQuery 物件。所以，應該把以下的程式碼：

```
1 var message = $('input[name="message"]').val();
```

重構為：

```
1 var message = this.$el.find('input[name="message"]').val();
2
```

this 是一個物件，代表「這個 View」的意思。*this* 物件就是 *Backbone.View* 類別。

使用 *this.\$el* 而不是直接使用 *\$* 物件的原因，是為了避免不必要的 DOM 操作。另外，是不是也要把 *\$.ajax* 改成 *this.\$el.ajax* 呢？答案是否定的。正確的做法，應該是用 Model 的方式來取代。

認識 *Backbone.Model*

*Backbone.Model*²⁴ 是 Backbone 初學者的第三關。*Backbone.Model* 用來表示（存放）資料，這是一個「資料模型」的類別。接下來以幾個步驟，重新實作第 11 章的「顯示留言」功能。

Step 1: 宣告 Model

有了 *Backbone.Model* 後，才能表示資料。修改 *client/javascripts/app.js* 加入以下程式碼：

²³[http://backbonejs.org/#View-\protect\char"0024\relax](http://backbonejs.org/#View-\protect\char)

²⁴<http://backbonejs.org/#Model>

```
1 1 app.Message = Backbone.Model.extend({
2 2   defaults: {
3 3     success: false,
4 4     errors: [],
5 5     errfor: {},
6 6
7 7     message: 'No message yet.'
8 8   }
9 9 });
```

說明如下：

- 第 1 行：使用 *Backbone.Model.extend()* 宣告（擴充）一個新的 Data Model，命名為 *Message* 並儲放於 *app* 物件
- 第 2 行：使用 *defaults*²⁵ 為這個 Model 定義預設的資料：
- *success*: Express.js 定義的欄位，目前暫不使用
- *errors*: Express.js 定義的欄位，目前暫不使用
- *errfor*: Express.js 定義的欄位，目前暫不使用
- *message*: 自行定義的欄位，用來儲存訊息

Step 2: 宣告 Template

原本的做法，將程式碼與 HTML5 語法混在一起，非常不好維護。現在改用 Underscore 與 Backbone 後，就可以使用 Template 的方式。以下是原本的實作：

²⁵<http://backbonejs.org/#Model-defaults>

```

1  var dataMapping = function(data) {
2      for (i = 0; i < data.length; i++) {
3          var htmlCode =
4              "<div class=\"alert alert-dismissable alert\
5  -info\">"
6              + "      <button type=\"button\" class=\"clo\
7  se\" data-dismiss=\"alert\" aria-hidden=\"true\">×</but\
8  ton>"
9              + "      <h4>jollen</h4>"
10             + data[i].message
11             + "</div>";
12
13         $('#message').append(htmlCode);
14     }
15 }

```

將 HTML5 語法的部份，獨立出來，並且宣告成 Template。修改 *chat.html*，加入以下片段：

```

1  <script type='text/template' id='tmpl-message'>
2      <div class="alert alert-dismissable alert-info">
3          <button type="button" class="close" data-dismiss="ale\
4  rt" aria-hidden="true">
5              ×</button>
6              <h4>jollen</h4>
7              <%= message %>
8          </div>
9  </script>

```

使用

- “type” 必須定義為 “text/template”
- 給予一個名字，上述範例將 Template 命名為 “tmpl-message”

- 使用 `<%= variable-name` \Rightarrow 來取用變數值，變數名稱在上述的 Model 裡定義，Underscore 會將 Model 的變數與 Template 做對應，並且用變數值取代

很明顯，Model 與 Template 是一個「對應關係」，這裡有幾個重要的技術觀念：

- ViewModel: 這個對應關係就是 ViewModel 的觀念
- Key-Value Pairs: Model 裡的資料，要用 Key-Value Pairs 的格式表示，JSON 就是 Key-Value Pairs 的格式
- Template: 將 Model 裡的資料，顯示到畫面上，是透過 Template，並且是由 Underscore 來完成
- Code Ignorance: 如上，顯示資料到畫面上，不需要寫程式；對設計師來說，只要修改 Template 即可，不會有程式碼的困擾

Step 3: 將 Model 加入 View

修改 `client/javascripts/app.js` 加入以下程式碼：

```
1 1   initialize: function() {
2 2       this.model = new app.Message();
3 3       this.template = _.template($('#tmpl-message').\
4 html());
5 4
6 5       this.model.bind('change', this.render, this);
7 6       this.render();
8 7   }
```

程式碼第 2 行，就是在 View 裡面加入 Model 的做法：

- 實例化 View 時，constructor 與 `initialize` 會被叫
- 如上，實作 `initialize` 函數來加入 Model

- 將 Model 的實例化存放到 *View.model* 裡

有了 Model 後，就可以開始做 “Data Mapping” 了。先使用 jQuery 取得定義好的 Template:

```
1 $( '#tmpl-message' ).html()
```

接著要在 Backbone.View 裡定義一個 *template*²⁶ 函數。我們採用的 Template 系統是 Underscore，要為 Backbone.View 定義想要使用的 Template 系統，就要定義 *template*。

如何讓 Backbone.View 使用 Underscore 做為 Template 系統呢？只要將 *Backbone.View.template* 定義為 *_.template()* 即可。程式寫法如下：

```
1 this.template = _.template( $( '#tmpl-message' ).html() );
```

_ 是 Underscore 的物件（這就是 Underscore 名稱的由來）。將取得的 Template 傳給 *_.template* 即可。

最後是 Data Mapping 的部份，根據 Backbone 的說明，必須在 View 裡實作 *render()*²⁷ 函數。*render()* 會將 Model 與 Template 做對應，對應後的結果就是一份 HTML5 文件。最後把 HTML5 文件顯示在畫面上即可。

以下是 *render()* 的實作：

²⁶<http://backbonejs.org/#View-template>

²⁷<http://backbonejs.org/#View-constructor>

```
1     render: function() {
2         var data = this.template(this.model.attributes);
3
4         this.$el.find('#message').html(data);
5         return this;
6     },
```

先呼叫 *this.template* 函數，這個函數已經被定義為 Underscore Template 系統。*this.model.attributes* 存放的是「Model State」，所謂的 Model State 就是 Model 目前所存放的資料；資料會以 JSON 的格式表示。

簡單來說，*this.model.attributes* 存放了 Model Data，並且是 JSON 格式。上述程式碼的意思就是：將 Data 交給 Template 系統去做 Data Mapping。這行程式碼是 ViewModel 觀念的靈魂。

最後將 Template 系統處理好的 HTML5 文件，放到想顯示的網頁位置即可。以上述範例為例，結果被放到 *#message* 區塊裡。

三個步驟就能入門 Backbone.Model 了。以下是目前為止的 *client/javascripts/app.js* 完整程式碼：

client/javascripts/app.js

```
1  1 /**
2  2   * SETUP
3  3   **/
4  4   var app = app || {};
5  5
6  6 /**
7  7   * MODELS
8  8   **/
9  9   app.Message = Backbone.Model.extend({
10 10     defaults: {
11 11         success: false,
```

```
12 12      errors: [],
13 13      errfor: {},
14 14
15 15      message: 'No message yet.'
16 16    }
17 17  });
18 18
19 19 /**
20 20  * VIEWS
21 21  */
22 22  app.MessageView = Backbone.View.extend({
23 23    el: '#message-save',
24 24    events: {
25 25      'click .btn-message-save': 'save'
26 26    },
27 27    initialize: function() {
28 28      this.model = new app.Message();
29 29      this.template = _.template($('#tmpl-message'\
30 ).html());
31 31
32 31      this.model.bind('change', this.render, this);
33 32      this.render();
34 33    },
35 34    render: function() {
36 35      var data = this.template(this.model.attribut\
37 es);
38 36
39 37      this.$el.find('#message').html(data);
40 38      return this;
41 39    },
42 40    save: function() {
43 41      var message = this.$el.find('input[name="mes\
44 sage"]').val();
45 42
46 43      $.ajax({
```



```
47 44          url: '/discussion/' + message,
48 45          type: 'POST',
49 46          dataType: "json",
50 47          success: function (data, textStatus, jq\
51 HR) {
52 48              alert(" 已儲存成功");
53 49          },
54 50          complete: function (data, textStatus, jq\
55 XHR) {
56 51              }
57 52          });
58 53      }
59 54  });
60 55
61 56 /**
62 57  * BOOTUP
63 58  */
64 59  $(document).ready(function() {
65 60      app.messageView = new app.MessageView();
66 61  });
```

要入門 Backbone.js，就要過三關：

- Backbone.View.extend
- View.\$el
- Backbone.Model

要見識 Backbone.js 框架的威力，就要再過二關：

- Backbone.Model.fetch
- Backbone.Model.save

學會以上 5 個主題後，才能算是真正入門 Backbone.js。

Backbone.Model 除了表示資料外，還提供各種處理模型，最重要的處理模型就是：manage changes。一旦 Model 裡的資料有變動（例如：新增、刪除等），就要重新做“Data Mapping”。這個部份要如何處理？Backbone Way 的處理方式，是透過 Backbone.Model.fetch。

認識 Backbone.Model.fetch

`Backbone.Model.fetch`²⁸ 是 Backbone 初學者的第四關。它的用途是：reset the model's state from the server。

怎麼從 Server 取得資料？這就是 `Backbone.Model.fetch()` 的主要用途。接下來，為 nodejs-chat 加上顯示訊息的功能。目前為止的範例，仍有幾個部份需要加強：

- 使用 `$.ajax()` 來呼叫 REST API 雖然是可行的做法，但是有一個缺點：處理 Response data 的架構不夠嚴謹
- 使用 `$.ajax()` 來呼叫 REST API 時，是直接把 API 當做 `ajax()` 的參數，這個做法有一個缺點：API 與 Response data 是一種比較鬆散的關係

以上二個缺點，都可以用 Backbone.js 的架構來解決：

- 使用 Model fetch 的關係，讓 Response data 的處理更嚴謹
- 使用 Data model 的關係，讓 Data 與 API 偶和（Aggregation）在一起

接下來，繼續重構 nodejs-chat 專案，讓它的架構更加嚴謹。

Step 1: 加入 Backbone.Model.fetch

修改 `client/javascripts/app.js` 加入以下程式碼：

²⁸<http://backbonejs.org/#Model-fetch>

```
1  this.model.fetch({
2      success: function(model, response, options) {
3      }
4  });
```

說明如下：

- 呼叫 *fetch()* 重置 Model State
- *fetch()* 應該要呼叫 REST API — */discussion/latest/:items*, 來取得最新的訊息
- *fetch()* 的參數中, 包含一個 *success* 的 Callback Function
- 呼叫 REST API 成功後, 會呼叫 *success* callback, 第二個參數 *response* 存放的就是 Server 回傳的資料

要如何指定 *fetch()* 所要呼叫的 REST API 呢? 方式是修改 Model。

Step 2: 修改 Model

Backbone.Model.fetch 會根據 *this.model* 裡的 *url* 定義, 來呼叫 REST API。因此, 修改 Data model 的程式碼如下:

```
1  app.Message = Backbone.Model.extend({
2      url: '/discussion/latest/5',
3      defaults: {
4          success: false,
5          errors: [],
6          errfor: {},
7
8          message: 'No message yet.',
9          messages: []
10     }
11 });
```

對初學者來說, 現階段並不需要太深入 Data model 的理論面, 只需要學會定義 Model 即可。

Step 3: 設定 Data Model

呼叫 *Backbone.Model.fetch* 後，Backbone 會幫我們呼叫 REST API，並且取得 Response data。如果可以成功取得 Response data，*Backbone.Model.fetch* 就會 Callback *success* 函數。

我們只要實作 *success* callback 函數，並且將 Response data 「套用」到 Data model 裡即可。程式碼如下：

```
1 this.model.fetch({  
2     success: function(model, response, options) {  
3         self.model.set('messages', response);  
4     }  
5 });
```

Backbone 將取得的 Response data 存放在 *response* 參數。接著，我們必須重新設定 Data model。

self.model.set 的意思是，將取得的 response data 儲存到 Data model 裡。由於 Data model 有了變動，所以 Backbone 便會呼叫 *render()* 函數。在 *render()* 函數裡，再 Render 出新的內容。

學會 *Backbone.Model.fetch* 後，就可以很容易感受到 Backbone 的優點：

- 透過 Data model 整合 REST API，以及 Response data
- 將 REST API 與 Response data 封裝成 Data model
- 在 Data model 變動時，重新 Render 畫面

認識 **Backbone.Model.save**

Backbone.Model.fetch 是一個很神奇、好用與重要的觀念。神奇的地方是：它可以同時支援 RESTful 的 POST 與 PUT 操作。好用的地方是，Backbone 可以自動處理資料的新增（POST）或更新（PUT）。重要的地方是，它緊密結合 Data Model。

檢視 nodejs-chat 範例，目前儲存留言的程式碼實作，仍然是採用 jQuery AJAX 模式：

```
1      save: function() {
2          var message = this.$el.find('input[name="message"]')
3            .val();
4
5          $.ajax({
6              url: '/discussion/' + message,
7              type: 'POST',
8              dataType: "json",
9              success: function (data, textStatus, jqXHR)\
10 {
11             alert(" 已儲存成功");
12         },
13         complete: function (data, textStatus, jqXHR\
14 ) {
15         }
16     });
17 }
```

這段實作，必須重構為 Backbone way。程式碼如下：

```
1      save: function() {
2          var message = this.$el.find('input[name="message"]')
3            .val();
4
5          this.model.save({ message: message});
6      }
```

this.model.save() 執行後，Backbone 會以 POST 方法，呼叫 REST API。這裡有一個疑問，Backbone 怎麼知道現在是要新增訊息，而不是更新訊息？方式非常簡單：

- 如果傳入的 Data 包含 *id* 欄位，表示這筆資料已存在，Backbone 就視為更新資料；透過 PUT 方法呼叫 REST API
- 反之，沒有 *id* 欄位，視為新增資料；以 POST 方法呼叫 REST API

簡單說：

```
1 // PUT (更新)
2 this.model.save({ id: id, message: message });
3
4 // POST (新增)
5 this.model.save({ message: message });
```

結論

透過本章的介紹，可以學習到基礎的 Backbone.js 觀念，包含：為什麼要導入 Backbone.js 框架。導入 Backbone.js 框架的初步原因：

- 需要能表示資料的 Data Model
- 需要能封裝程式碼邏輯的 Logical View
- 需要一個方便處理 Response data 的 Data Model，即 ViewModel 的觀念

本章的範例，以 Backbone 框架的角度來看，完成度已經接近 100% 了。本章的三個基本關卡，讓初學者初步入門 Backbone.js；緊接著的二關進階關卡，讓初學者了解 Backbone.js 的 Data Model 觀念。