

Sokoban.js 專案設定

喬逸偉 (Yiwei Chiao)

1 Httpd

這一章將利用 [Node.js](#) 建立一個簡單的 Web 服務器，以提供 [Sokoban](#) 遊戲內容到瀏覽器端。先從簡單的傳送靜態 Web 網頁開始。

1.1 專案目錄結構

專案目錄結構是程式碼住的小窩。程式碼住的舒服，程式開發與維護才能輕鬆寫意。[sokoban.js](#) 的專案結構規劃如下：

專案目錄結構

D:\sokoban	專案資料夾
.gitignore	.gitignore 檔
LICENSE	授權聲明
README.md	專案說明檔。Markdwon 格式。
—htdocs	客戶端程式資料夾
—assets	客戶端程式資源
—css	.css 檔
—fonts	web 字型檔
—png	.png 檔
—html	.html 檔
—js	.js 檔；客戶端 Javascript/ECMAScript 程式碼
—httpd	伺服器端程式資料夾
index.js	伺服器端程式入口 (main)
js	伺服器端 Javascript/ECMAScript 程式碼

如列表所示，整個專案大致分為兩個資料夾：

- *htdocs*: 客戶端 (瀏覽器) 相關程式與資源。
- *httpd*: 伺服器端 (node.js) 程式。

基本上，專案啟動時是啟動 *httpd* 資料夾內的 *index.js* 檔案 (伺服器端程式進入點)；而使用者利用瀏覽器 (browser) 連上伺服器後，伺服器端會先將放在 *htdocs* 資料夾裡的程式文件與資源按瀏覽器的要求依序傳送到客戶端執行或顯示。從而完成網頁應用程式的執行。

目前因為專案剛起動，所以除了 *httpd* 資料夾下的 *index.js* 檔案之外，其它資料夾下的內容大多是空的。而隨著專案進展，不同的檔案會逐步被引入，同時置於相應的資料夾之內。這樣規劃資料夾 (目錄) 結構的好處也就是不同用途，作用的檔案會存在不同的資料夾之內，使得專案發展時不會被不是當下相關的檔案干擾。

如果不清楚怎麼在 Linux/MacOS/Windows 作業環境下建立這樣的目錄結構，可以先跳到最後附錄一節有簡單的操作說明。

1.2 問題與思考

如果這個專案目錄結構 (或它的變形) 是可以接受的，那麼為了省下每次開一個新的專案都要手動建立一次同樣的目錄結構的麻煩：

- 有沒有一個自動化的方式可以在給定了專案名稱之後就自動建立相應的專案目錄結構？
- 進階：自己寫一個小工具來作這件事如何？(提示：[shell scripts](#)，[scripting languages](#))
- 進進階：你寫的工具，可不可以 **客製化** (customize)；讀取一個 **設定檔** (config file) 之後，依設定檔的內容自動建立不同的目錄結構？

2 Hello World

專案先由傳統的 *Hello World!* 開始。

2.1 *index.js*

一開始的 *index.js* 的內容如下：

index.js

```
1: 'use strict';
2:
3: let http = require('http');
4:
5: http.createServer((request, response) => {
6:   // 傳送 HTTP header
```

```
7: // HTTP Status: 200 : OK
8: // Content Type: text/plain
9: response.writeHead(200, {
10:   'Content-Type': 'text/plain'
11: });
12:
13: // 傳送回應內容。
14: response.end('Hello World!\n');
15:
16: console.log('request.headers: \n', request.headers)
17: }).listen(8088);
18:
19: // log message to Console
20: console.log(' 伺服器啟動，連線 url: http://127.0.0.1:8088/');
```

將 index.js 存檔後，在命令提示字元 (cmd)/終端機 (terminal) 下，輸入 node index.js，如圖 Figure 1，

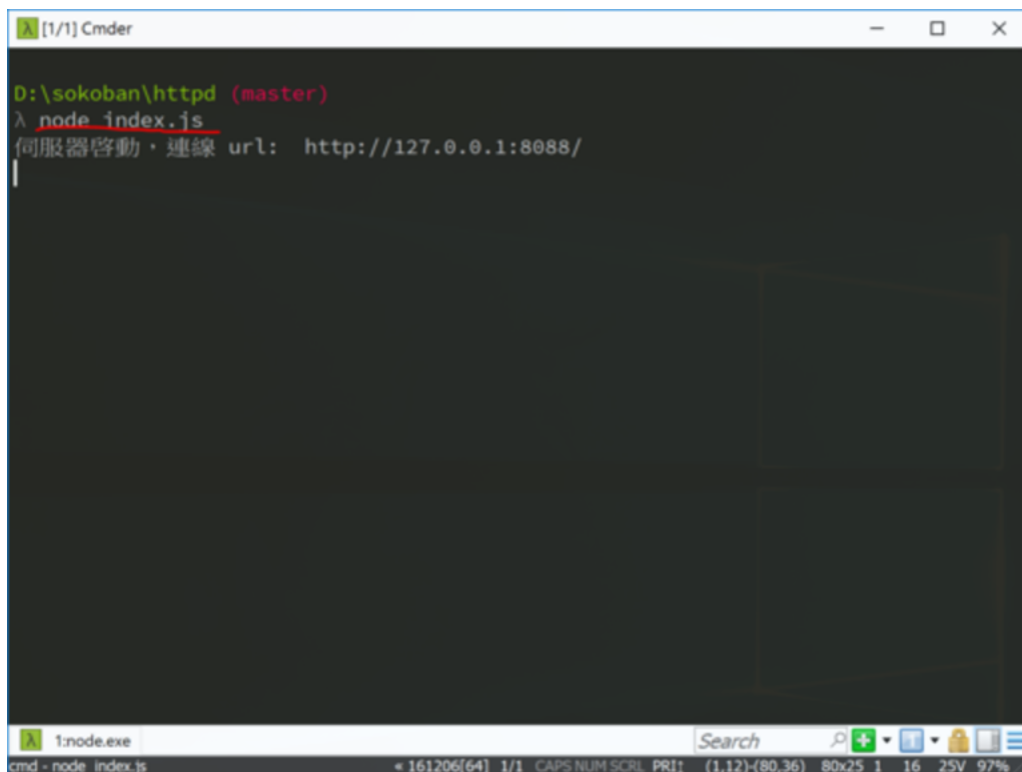


Figure 1: 啟動伺服器

就可以啟動剛剛寫作的簡易網頁伺服器；再利用網頁瀏覽器，依伺服器訊息指示，開啟 <http://127.0.0.1:8088> 就會看到如下的畫面 (Figure 2)。

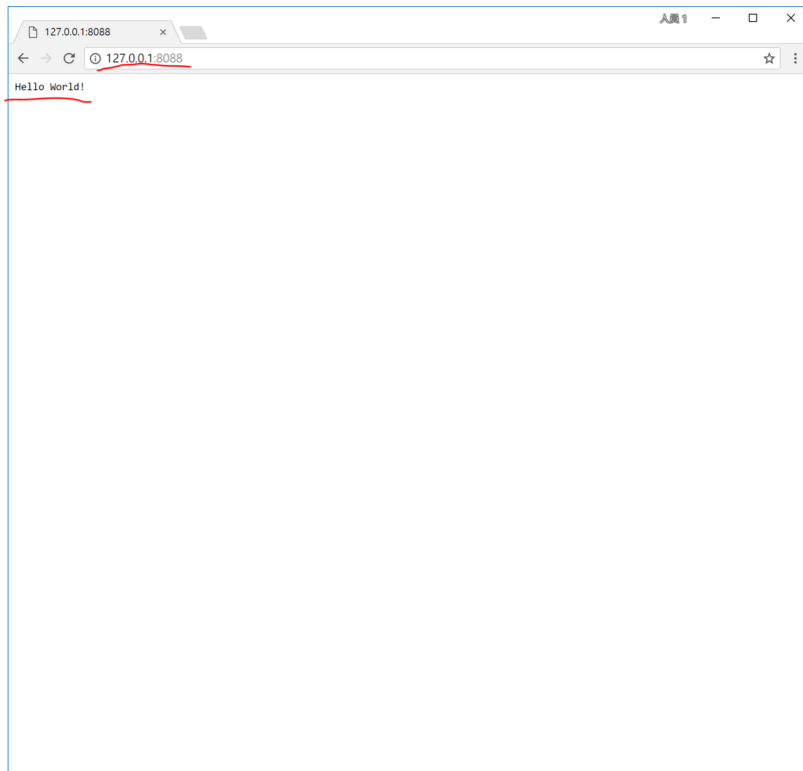


Figure 2: Hello World!

2.2 index.js 程式說明

如 Figure 2 所示，瀏覽器就簡單顯示了 Hello World! 訊息，沒有任何一般網頁的風範。這是因為伺服器真的什麼都沒作，就只送了那兩個英文字。比對程式碼，可以發現 Hello World! 出現在程式碼的第 14 行。由上下文的觀察可以猜出第 5 行到第 17 行是整個回應的核心。下面就來逐行解析這個程式的內容。

2.2.1 Line 1: `'use strict';`

之前有提過，[JavaScript](#) 在 2009 年左右開始了一場標準化的變革，以應對愈來愈重要的網頁平台需求。為了擁抱改變，但又不能捨棄過去，所以 [ECMAScript 5](#) 引入了 strict 模式。

strict 模式對語法有更嚴格的要求，對過去一些模糊的語法作出了明確的規範或捨棄。舉例而言，strict mode 要求變數一定要宣告，但傳統的 [JavaScript](#) 不用。更詳細的討論可以見 [MDN \(Mozilla Developer Network\)](#) 對 [strict mode](#) 的介紹。

利用 strict 模式的宣告讓程式設計師可以有簡單的機制將傳統和新的標準化 [JavaScript](#) 作連結。

第一行的 `'use strict';` 就是聲明這個程式是以 strict 模式撰寫。

2.2.2 Line 3: `let http = require('http');`

- `let` 是 ECMAScript 6 (又稱 ECMAScript Harmony/ES Harmony/ ES 2015) 引入的 **關鍵字**之一。用途是 **宣告**一個 **範圍變數** (scoped variable)；相對於 `var` 宣告的變數則是 **全域** (global vairable) 或與函數綁定 (closure)。
- `require` 是 Node.js 提供的 **模組** (module) 系統 API 函數之一。作用為 **載入** (load) 指定的 JavaScript 模組。這裡載入的是名為 `http` 的 Node.js 標準模組。

透過 `require` 載入的模組就初始化為一個 JavaScript 的程式物件 (object instance)，可以直接呼叫使用。

`http` 模組是 Node.js 提供的打造 HTTP 服務器或客戶端的核心模組。它會負責處理和網路通訊相關真正麻煩的部份，專案需要作的，只是提供它必要的資訊。

2.2.3 Line 5: `http.createServer(...)`

- `HTTP.createServer`
如它的名字所示，是 `http` 的內建方法，作用就是建立起一個 http protocol 的伺服器物件 (`http.Server`)。 `HTTP.createServer` 要求一個 **函數參數**: `requestListener`。

因為 `http` 伺服器物件知道如何處理網路協定，但無法知道如何回應使用者要求。這個傳入的 `requestListener` 函數就是 `http.Server` 物件接收到使用者要求 (*request* 事件) 時，需要 **呼叫執行**來回應使用者要求的函數。因為 `requestListener` 函數不是由程式設計師在編寫程式時主動呼叫，而是由程式物件在程式執行時期自己呼叫，所以稱為：**回呼函數** (*callback function*)。

Callback function 在類似 JavaScript 這類的函數式語言是個很重要的概念。

`requestListener` 函數需要接收兩個參數：

- `request`: JavaScript 物件，代表瀏覽器傳遞的使用者要求。
- `response`: JavaScript 物件，用來回傳伺服器想傳送給瀏覽器的資料。

這裡專案程式碼利用 **匿名函數** `(request, response) => {...}` 實作了 `HTTP.createServer` 要求的 `requestListener`

- `(request, response) => {...}`
就是 `function (request, response) {...}` 的另一種寫法。也就是宣告了一個 **匿名函數** (anonymous function)。
`(...) => {...}` 稱為 *arrow function* 除了語法 (syntax) 的改變外，在語義 (semantic) 上，*arrow function* 和傳統函數也有些細節的不同，這裡先略過，有興趣可以先去看參考資料。

2.2.4 Line 9 ~ Line 14: **response**

上面的 `response` 物件其實是 Node.js 提供的 `ServerResponse` 類別的實例 (instance)。

`response` 物件回應，基本上可以分為三 (3) 步走：

1. `ServerResponse.writeHead(statusCode[, statusMessage][, headers])`: 傳送 HTTP header 資訊。參見 `ServerResponse.writeHead`
2. `ServerResponse.write(chunk[, encoding][, callback])`: 傳送回應內容。參見 `ServerResponse.write`。
3. `ServerResponse.end([data][, encoding][, callback])`: 回應結束。一定要呼叫。參見 `ServerResponse.end`。

由上述流程中可以看到 `ServerResponse.writeHead(...)` 基本回應了瀏覽器這次訪問是否成功 (statusCode)，瀏覽器應該如何解讀回應內容 (headers) 等。

header 裡主要是放回應內容的編碼方式 (encoding='utf8')，內容型別 (MIME types，如: 'Content-Type: text/plain') 等。

因為這個小程式只需要回應 'Hello World!'，所以就直接省略了對 `ServerResponse.write(...)` 的呼叫；直接以 'Hello World!' 作為參數呼叫 `ServerResponse.end(...)`。

2.2.5 Line 16 **console.log(...)**

Node.js 的 `console` 物件代表的就是 **命令提示字元/終端機** (terminal)。主要用途是 log 訊息到螢幕輸出，讓系統管理員能夠知道系統發生了些什麼事情。

而裡面的 `request` 物件是 Node.js 提供的 `ClientRequest` 類別的實例 (instance)。記錄了瀏覽器傳送的資料細節。

目前專案沒有用到它的內容，這裡單純將 `request.headers` 的資訊倒到 `console` 上。

2.2.6 Line 17: **listen(8088)**

到 17 行，整個 `HTTP.createServer(...)` 就完成了呼叫，傳回一個 `http.Server` 的實例。但這個伺服器物件還沒有起動。第 17 行尾端的 `listen(8088)` 就是起動這個伺服器物件，同時指定它在 8088 這個通訊埠 (port) 等待使用者的連線。

2.2.7 Line 20 **console.log(...)**

如果一切順利，到這兒，一個 http 伺服器程序已經在背後運行，但是沒有任何訊息呈現在 `console`，系統管理員無從判斷系統狀態。所以這裡在 `console` 印出一個提示訊息，說明伺服器已起動，同時印出它的 ip 地址和 port 號碼。

2.3 Javascript 模組系統簡介

這小節是關於 JavaScript 模組系統的一些簡略背景介紹，和專案程式碼沒有直接關係。如果沒有興趣可以跳過。

傳統 JavaScript 本身沒有模組系統 (ECMAScript 6 之後才有)。程式碼不管分成幾個檔案，執行後全部混在一起，所有的變數全部都是 global，包在物件或 **閉包** (closure) 裡。對發展大型專案或初學者而言是個充滿陷阱的雷區。

為解決這個問題，Node.js 和其它一些 JavaScript 的函式庫，利用 JavaScript 的物件 (objects) 和 closure 建構了自己的模組系統。當然，各家的實作方法都不太相同。為了保持中立，ECMAScript 6 提出的模組系統 當然又是另一種。所以說，JavaScript 是個充滿新鮮感與驚奇的語言。

Node.js 使用的模組系統繼承自 CommonJS，為伺服器端 JavaScript 程式；另一個廣泛使用的格式稱作 AMD (Asynchronous Module Definition) 是針對 Web 瀏覽器設計；最後一種，稱為 IIFE (Immediately-Invoked Function Expression)。IIFE 本身其實是應用 JavaScript function 物件的一個進階程式技巧，並不是什麼模組系統。但實作簡單，應用廣泛，在這個專案的進行中也會引入 IIFE 的寫作方式。

Sokoban.js 專案在：

- 伺服器端，使用 Node.js，使用 Node.js 的模組系統。
- 客戶端，使用 ECMAScript 6 的模組系統。

3 版本管理 - git 基礎使用

專案開發，避免不了的是相關檔案及內容的增刪修改。更煩人的是，有可能增刪修改了某個部份，結果導至專案有了非預期的行為，想要回復之前的狀態卻發現忘了修改那些地方；或者，修改的地方太多，已經沒有力氣回頭了。

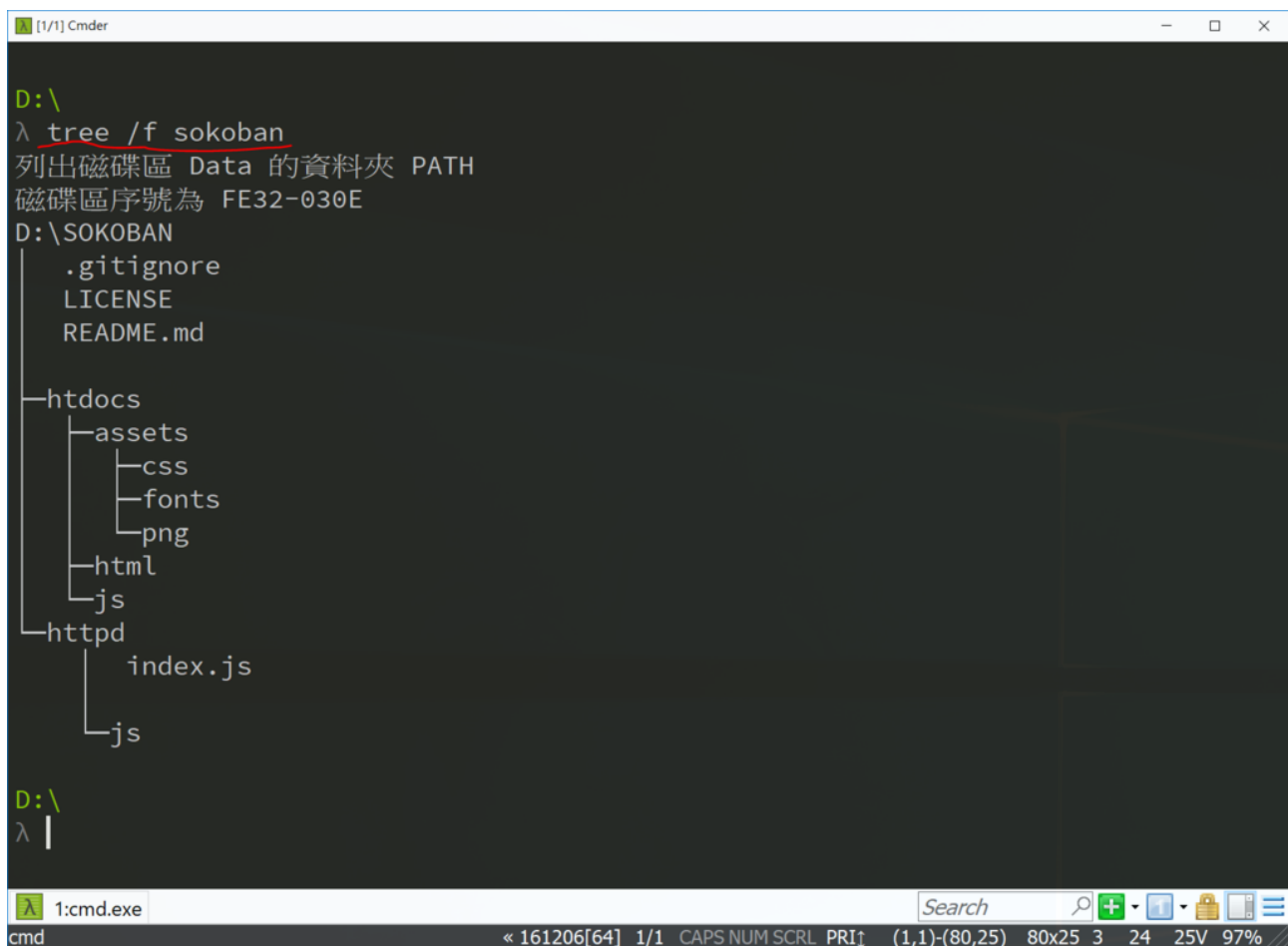
這些情境，還有其它的一些問題，是**版本管理系統** (Version Control System) 發展的背後動力。

Sokoban.js 專案放在 GitHub 上，GitHub 是利用 git 架設的雲服務。而 git 本身正是一個**版本管理系統**。

目前專案寫了第一版的 index.js 檔，可以在瀏覽器上顯示 Hello World!。正好用來建立專案在 git 系統裡的第一個 **節點**。

3.1 專案目前狀態

如果沒有其它變數，目前的專案檔案結構應該長的很像 Figure 3 所示。



```
D:\>
λ tree /f sokoban
列出磁碟區 Data 的資料夾 PATH
磁碟區序號為 FE32-030E
D:\SOKOBAN
|
| .gitignore
| LICENSE
| README.md
|
+-- htdocs
|   |
|   +-- assets
|   |   |
|   |   +-- css
|   |   +-- fonts
|   |   +-- png
|   |
|   +-- html
|   +-- js
|
+-- httpd
|   |
|   +-- index.js
|   +-- js
|
D:\>
λ |
```

Figure 3: 專案目錄結構

3.1.0.1 Note

- tree 是 Windows 的指令；
- Linux 環境：
 - 如果系統內沒有，去找該 Linux 發行版的套件庫，
 - 或者到這個連結自己安裝：<http://mama.indstate.edu/users/ice/tree/>
- MacOS 環境：
 - 如果沒裝 Homebrew，先安裝 Homebrew；
 - 有安裝 Homebrew，直接下 `brew install tree`

3.2 專案目前狀態 - git status

Figure 3 是檔案系統內看到的樣子；另一方面，由 git 的觀點來看，專案目前的狀態應該類似 Figure 4：


```
D:\
λ cd sokoban

D:\sokoban (master)
λ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md
        httpd/

nothing added to commit but untracked files present (use "git add" to track)

D:\sokoban (master)
λ |
```

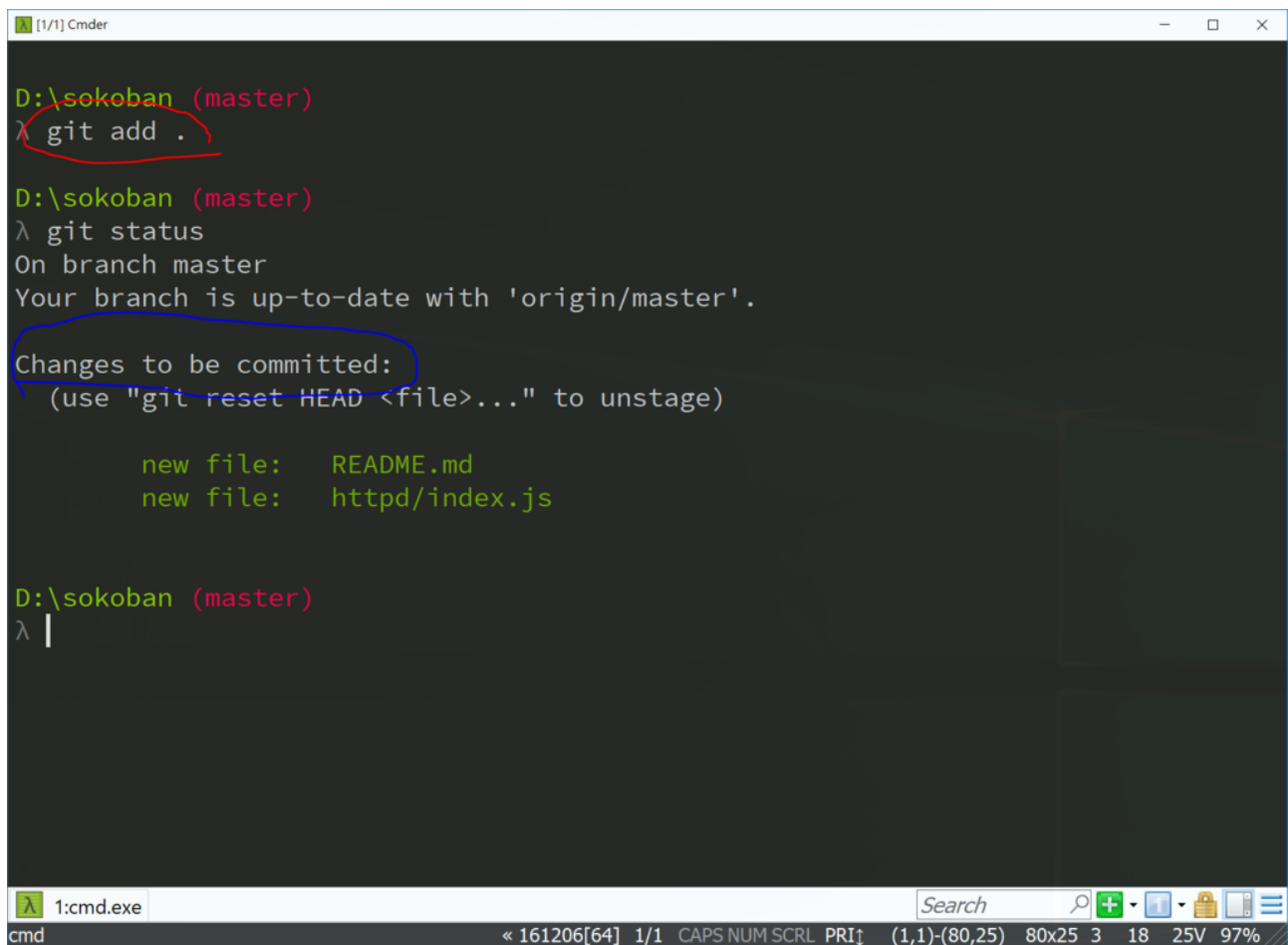
Figure 4: 專案目前狀態 - git status

Figure 4 裡有三 (3) 處用色筆圈起來的部份。

- 紅圈：git status 呼叫 git 回報目前的專案狀態。git 本身是個命令列工具，採用 cli 介面，為了避免與其它的 cli 介面工具有命令名稱衝突 (name colision)，所有的 git 指令都由 git 這個中控指令來處理。如 git status，git add 等。
- 藍圈：git status 如果發現專案目錄結構裡有檔案變動，就會在螢幕上回報。這裡因為專案新增了資料夾 httpd 和其下檔案 httpd/index.js，所以 git 回報有 Untracked files: - 未被追蹤的檔案存在。同時，在藍圈下方的區域可以看到紅色字的列表，列出了 httpd 資料夾和 README.md 檔案。
為什麼 htdocs 和其它的資料夾沒有列出來？因為 htdocs 資料夾下目前沒有任何檔案存在。單純是個空資料夾。git 不會追蹤空資料夾，因為那沒有意義。
- 黃圈：單純是 git 給使用者的建議。因為發現了未被追蹤的檔案存在，git 提示可以利用 git add 指令將它們加入追蹤。

3.3 將檔案加入追蹤 - git add

依 git 的建議，呼叫 git add . 指令後，再呼叫 git status，可以看到類似 Figure 5 的樣子。



```
D:\sokoban (master)
λ git add .

D:\sokoban (master)
λ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

        new file:   README.md
        new file:   httpd/index.js

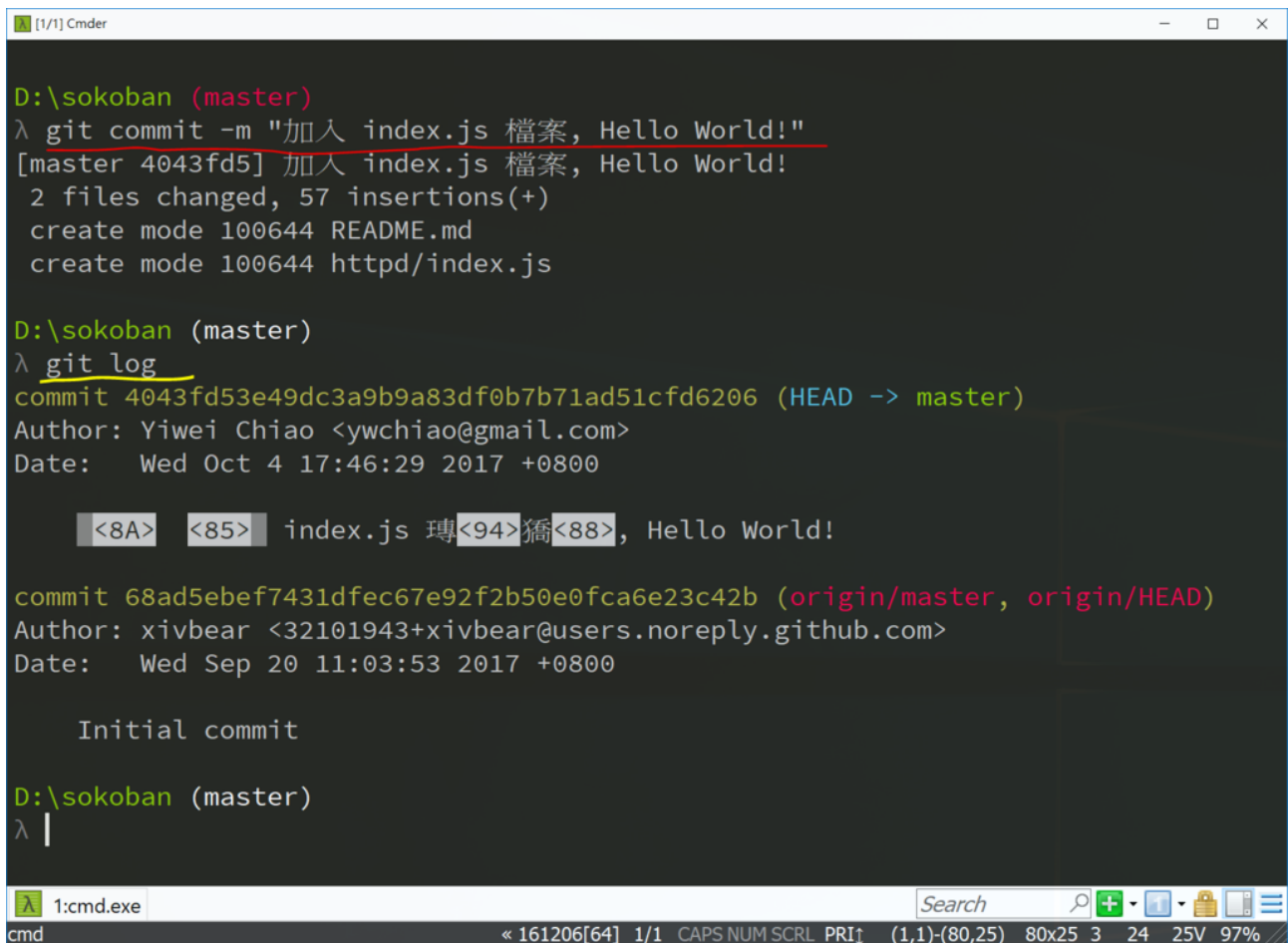
D:\sokoban (master)
λ
```

Figure 5: 將檔案加入追蹤 - git add

- 紅圈：git add . 將 **當前**資料夾下，含所有子資料夾，所有變動的檔案都納入追蹤。注意那個 **句點** (.) 指得是當前資料夾 (目錄)。如果需要的話，也可以 git add filename 只將檔名為 filename 的檔案加入追蹤；只是這樣需要一個一個的加，可能，不是那麼理想。
- 藍圈：呼叫完 git add . 後，再次呼叫 git status 確認目前狀態。git 回報 Changes to be committed. - 準備好要 commit (執行) 的變動列表。就下方區域的綠色字列表。可以再次確認是否真的要將這些列表的檔案加入追蹤。藍圈那行下面的括號內有提示如果反悔要將某個檔案解除追蹤的話可以利用 git reset HEAD <file> 來達成。

3.4 確定執行變動 - git commit

如果一切都確認沒問題，可以如 Figure 6 所示，



```
D:\sokoban (master)
λ git commit -m "加入 index.js 檔案, Hello World!"
[master 4043fd5] 加入 index.js 檔案, Hello World!
 2 files changed, 57 insertions(+)
 create mode 100644 README.md
 create mode 100644 httpd/index.js

D:\sokoban (master)
λ git log
commit 4043fd53e49dc3a9b9a83df0b7b71ad51cfd6206 (HEAD -> master)
Author: Yiwei Chiao <ywchiao@gmail.com>
Date:   Wed Oct 4 17:46:29 2017 +0800

    <8A>   <85>   index.js 璵<94>獮<88>, Hello World!

commit 68ad5ebef7431dfec67e92f2b50e0fca6e23c42b (origin/master, origin/HEAD)
Author: xivbear <32101943+xivbear@users.noreply.github.com>
Date:   Wed Sep 20 11:03:53 2017 +0800

    Initial commit

D:\sokoban (master)
λ |
```

Figure 6: 確認執行變動 - git commit

`git commit -m "index.js ..."` 執行 `commit` 的指令是 `git commit`，後面 `-m` `"..."` 的參數則是作變動的註解，讓自己未來回顧版本變動時，可以知道這個時後發生了什麼事。

`-m "..."` 後面的字串內容當然隨個人的意思打，但 **一定要加**。如果只下 `git commit` 而沒有跟著後面的 `-m "..."`，`git` 會認為你需要更多，遠超過一個字串所能表達的內容要記錄，`git` 會貼心的幫你叫出文字編輯器，等你輸入文字。

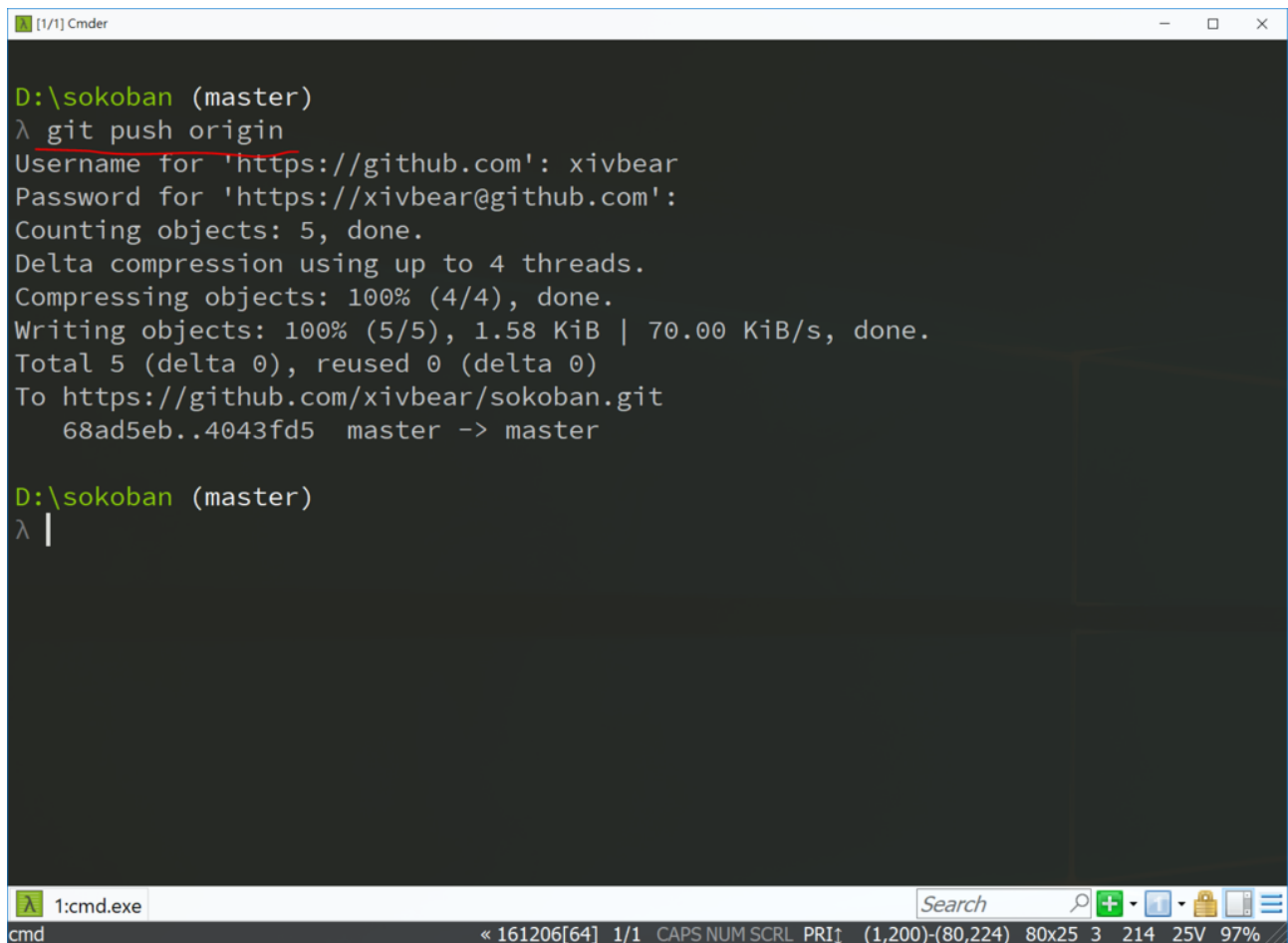
3.5 檢視變動歷史 - git log

在 Figure 6 的下半部。如果呼叫 `git log` 指令，就可以看到這個專案由啟動以來的 `commit` 歷史。也就可以看到剛剛在 `-m "..."` 裡的内容顯示在螢幕上。

可以看到 `git log` 顯示的 `-m "..."` 訊息是一團亂碼。這是因為 Windows 系統雖然支援 Unicode (utf8)，`git` 也使用 Unicode (utf8)，但是因為歷史因素，Windows 在命令提示字元下要顯示文字時，一律會轉換成各地區的編碼。例如，台灣的 Windows 命令提示字元強制設在 CP 950 (微軟字碼頁，實實就是 Big5 碼)，所以會有亂碼問題。解決方案？個人放棄。全使用英文最簡單，免傷腦筋。

3.6 同步至遠端 (GitHub) - `git push origin`

`git commit -m "..."` 之後只是將變動在本機端 (local) 記錄下來 [GitHub](#) 上的專案沒有變。為了讓兩邊的專案變動同步，可以使用 `git push origin` 指令。如 Figure 7。



```
D:\sokoban (master)
λ git push origin
Username for 'https://github.com': xivbear
Password for 'https://xivbear@github.com':
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 1.58 KiB | 70.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To https://github.com/xivbear/sokoban.git
    68ad5eb..4043fd5  master -> master

D:\sokoban (master)
λ |
```

Figure 7: 遠端同步 - `git push origin`

執行 `git push origin` 之後，`git` 會回報它傳了多少資料到遠端去。

3.7 由遠端同步回本機 - `git pull origin`

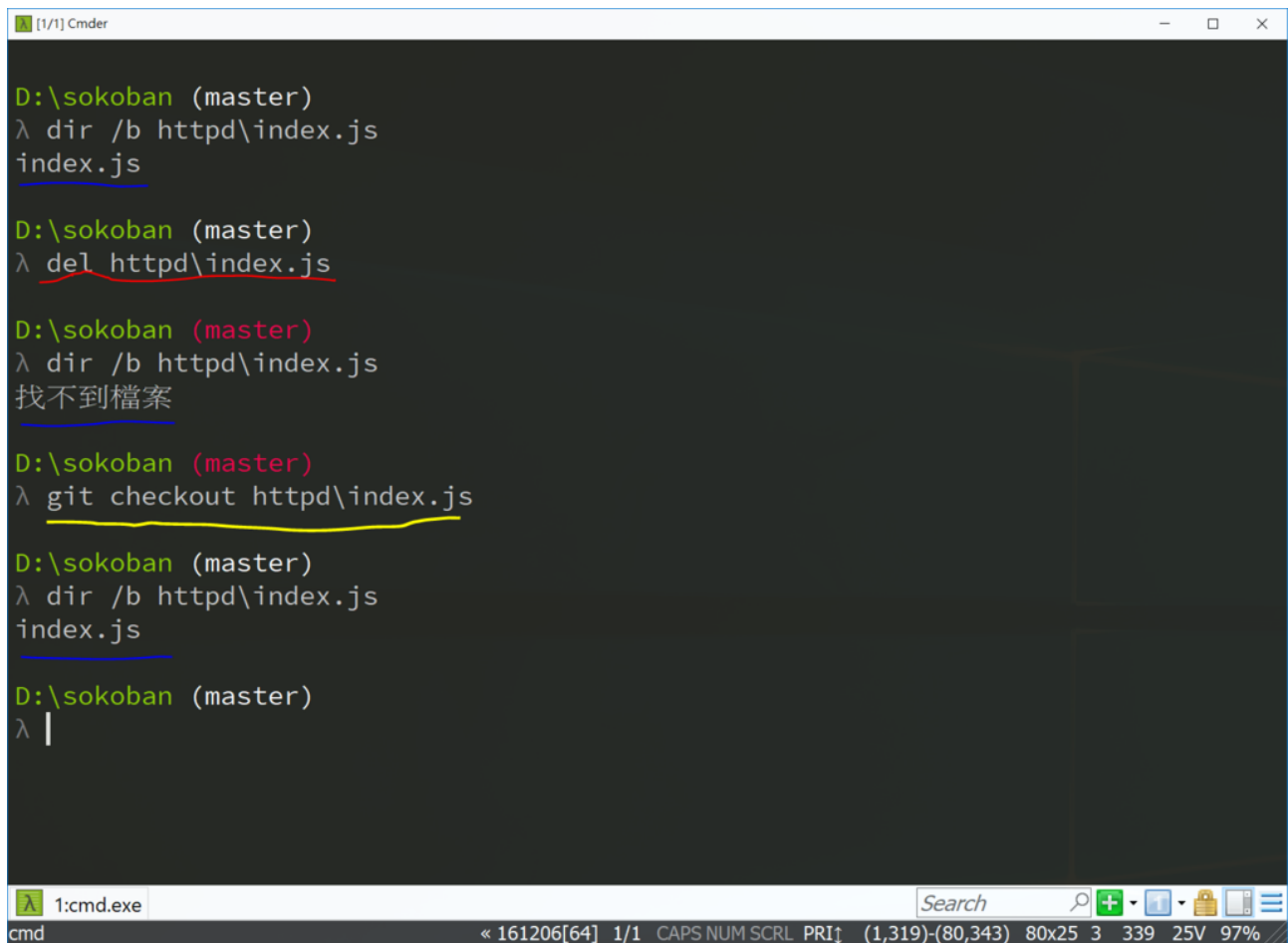
假設使用兩台以上的電腦在寫作這個專案，假定一台稱作 A，一台稱作 B。假定，兩台電腦都已由 [GitHub](#) 利用 `git clone` 取得了同一個專案的專案源碼。

進一步假定 A 電腦對專案作了更新，呼叫了 `git commit`，並利用 `git push origin` 將更新同步到 [GitHub](#) 上；如果 B 電腦想要取得 A 電腦的更新怎麼辦？

可以在 B 電腦利用 `git pull origin` 指令由 [GitHub](#) 將 A 電腦更新的資料拉下來。拉下來之後，除非 A 電腦在 `git push` 後還有更新，否則 A，B 電腦和 [GitHub](#) 上的專案源碼將會是相同的。

3.8 回復之前狀態 - git checkout

開始時提過，git 是個版本管理系統，而版本管理系統的目的就是要管控專案發展中不同的版本分支或變動。現在專案已交由 git 管理，所以可以簡單的來玩一下。如圖 Figure 8



```
D:\sokoban (master)
λ dir /b httpd\index.js
index.js

D:\sokoban (master)
λ del httpd\index.js

D:\sokoban (master)
λ dir /b httpd\index.js
找不到檔案

D:\sokoban (master)
λ git checkout httpd\index.js

D:\sokoban (master)
λ dir /b httpd\index.js
index.js

D:\sokoban (master)
λ |
```

Figure 8: git 測試

- 藍線：在 Figure 8 裡畫藍色底線的部份是利用 dir 指令在看 httpd/index.js 在不在；
- 紅線：畫紅線的部份，利用 del 指令將 httpd\index.js 刪除。由藍線部份也可以看到 httpd/index.js 在那之後真的不存在了。
- 黃線：畫黃線的部份，利用 git checkout 指令到 git 的版本倉庫裡將之前 commit 的 httpd\index.js 取出；於是 httpd/index.js 檔案就又回來了。

也就是 git checkout 是用來將之前儲放在 git 倉庫裡的程式檔案拿出來，取代目前同名的檔案，如果有的話。而當然，git checkout 不是拿來像這個範例一樣，好像是用來 undelete 一樣。它的用途其實主要是版本回溯。只要是由 git commit 產生的版本節點，不管時間多久，改了幾次版，都可以利用 git checkout 時光倒流回到當時的那個版本狀態。而如何知道要回到之前的那個版本節點？就是靠當時 git commit -m "... " 裡，要求放的 -m "... " commit 訊息協助。

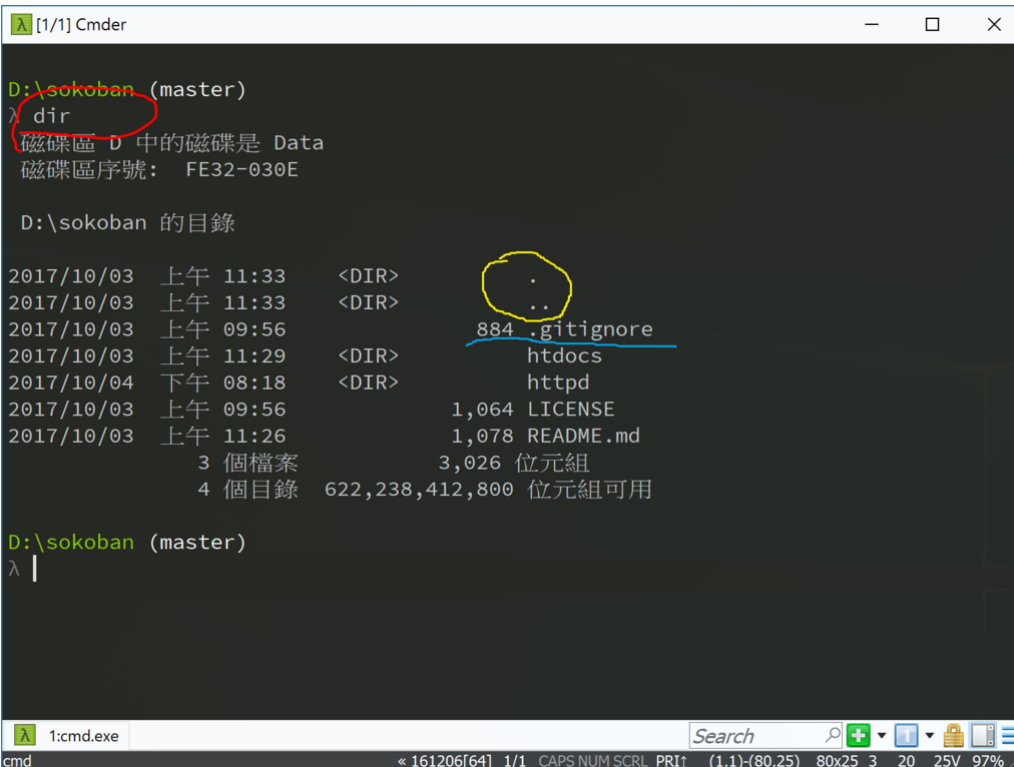
3.9 小結

隨著專案進展，將會不斷的使用這些基礎的 `git` 指令。或者，趁現在專案還年輕，上網或找書，多熟悉 `git` 的使用會是個不錯的主意。

4 基本 shell 指令

這一小節簡單介紹在命令提示字元/terminal 下的操作指令，一般稱為 shell 指令。基本目標為能夠在命令提示字元/terminal 環境下，於專案的目錄結構自在往來為主。範例圖式以 Windows 為主，Linux/MacOS 指令若有差異會再註明。

4.1 `dir`: 列出檔案列表



```
[1/1] Cmder
D:\sokoban (master)
> dir
磁碟區 D 中的磁碟是 Data
磁碟區序號: FE32-030E

D:\sokoban 的目錄

2017/10/03 上午 11:33 <DIR> .
2017/10/03 上午 11:33 <DIR> ..
2017/10/03 上午 09:56 884 .gitignore
2017/10/03 上午 11:29 <DIR> htdocs
2017/10/04 下午 08:18 <DIR> httpd
2017/10/03 上午 09:56 1,064 LICENSE
2017/10/03 上午 11:26 1,078 README.md
                        3 個檔案      3,026 位元組
                        4 個目錄    622,238,412,800 位元組可用

D:\sokoban (master)
λ |
```

Figure 9: 列出檔案列表

如 Figure 9 紅色圈起來的地方所示。在 Windows 環境的命令提示字元下輸入指令：`dir`，命令提示字元會列出當前資料夾下所有的檔案。

專案發展過程，如果需要知道目前所在資料夾，資料夾下有沒有特定檔案，有那些檔案，都可以利用這個指令來查看。

`dir` 指令名稱的來源來自英文的**目錄** (*DIRectory*。)

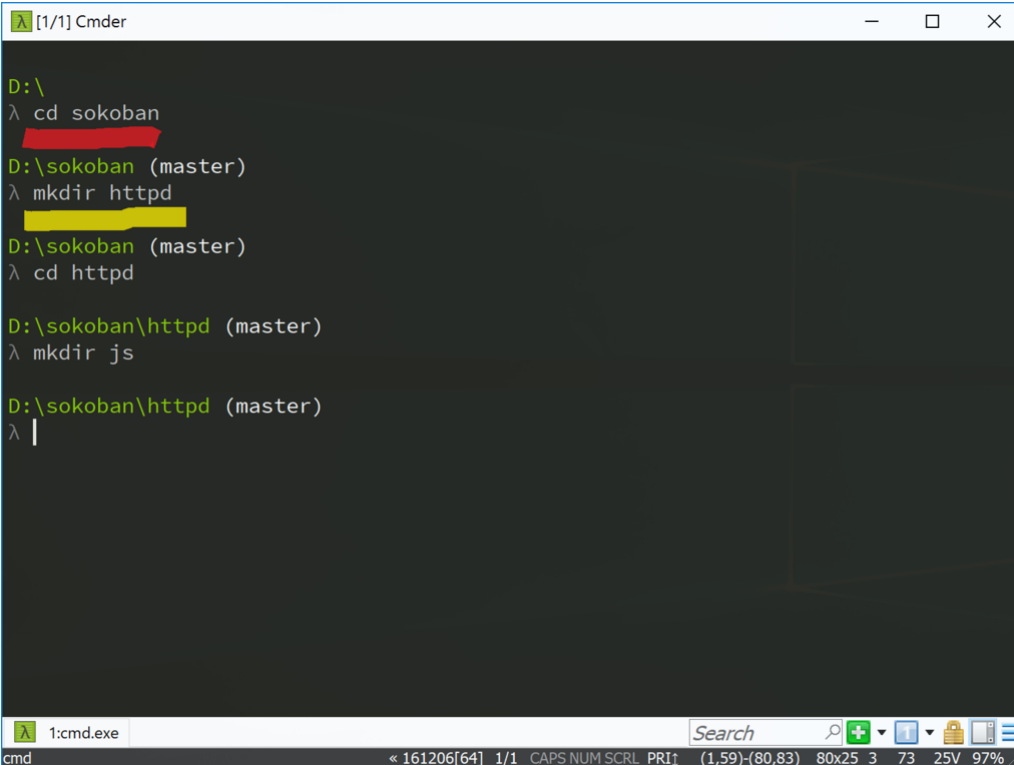
對應的，Linux/MacOS 環境的相對指令稱為 `ls`。`ls` 的命令名稱來源，你猜到了。英文的**列出** (*LiSt* directory。)

在 Figure 9 裡還有幾點值得一提：

- 當前目錄: `.` 與上層目錄: `..`: 在 Figure 9 裡可以看到用黃線圈起來的 `.` 和 `..`，其中，
- `.` 代表 **當前**資料夾，而
- `..` 代表 **上一層**資料夾。在某些指令操作場合很好用。舉例而言，`cd ..` 代表切換到上一層資料夾。
- 畫藍色底線的 `.gitignore`。記得，在 Windows 環境，檔名/目錄名前的 `.` 沒有意義，但在 Linux/MacOS 環境下，代表 **隱藏檔**，也就是，沒特別指定的話，`ls` 指令不會顯示名稱以 `.` 開頭的檔案或資料夾。

4.2 `cd`: 切換資料夾

如果要移動到不同的資料夾下工作。使用的指令是：`cd`，也就是英文的 *Change Directory*。如 Figure 10，畫紅色底線的部份。



```
[1/1] Cmder
D:\
λ cd sokoban
D:\sokoban (master)
λ mkdir httpd
D:\sokoban (master)
λ cd httpd
D:\sokoban\httpd (master)
λ mkdir js
D:\sokoban\httpd (master)
λ |
```

Figure 10: 目錄操作指令

如在 Figure 9 裡說明的，`cd ..` 代表切換至上層目錄；而這是可以組合的。意思是說：`cd ..\..` 代表移至上兩層目錄，`cd ..\..\..` 代表移到上三層目錄等。

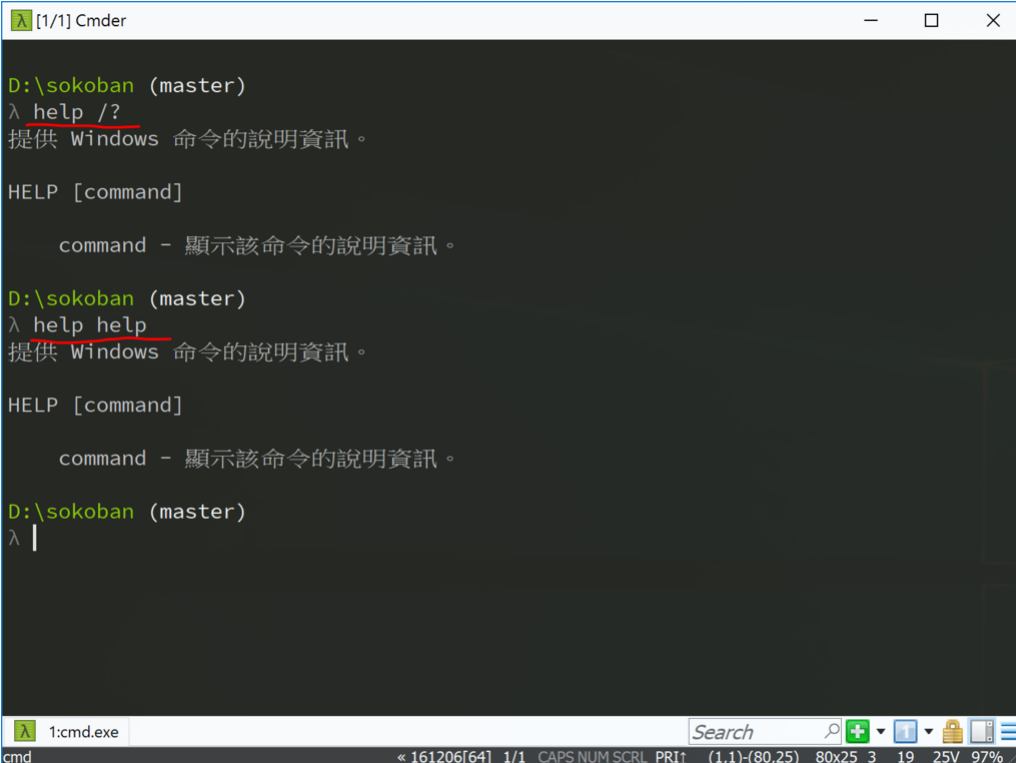
這裡要注意的是 `cd ..\..` 裡的 `\` (*backslash* 字元)；這個 `\` 字元用來作為 **路徑** (*path*) 的資料夾分隔字元；但是，同樣的目的，Linux/MacOS 使用的是 `/` (*slash*) 字元。

4.3 md 或 mkdir: 建立資料夾

Figure 10 裡，畫黃線的部份，呼叫 `mkdir` 來建立新資料夾；同樣指令也可簡寫為 `md` (*Make Directory*。)

4.4 指令說明/幫助

如果對這些指令還有疑問或想查有沒有進階的用法。除了上網問 Google，現在的作業環境都附帶完善的線上說明。



```
[1/1] Cmder
D:\sokoban (master)
λ help /?
提供 Windows 命令的說明資訊。

HELP [command]

    command - 顯示該命令的說明資訊。

D:\sokoban (master)
λ help help
提供 Windows 命令的說明資訊。

HELP [command]

    command - 顯示該命令的說明資訊。

D:\sokoban (master)
λ |
```

Figure 11: 指令幫助

如 Figure 11 所示，在 Windows 環境下有兩種取得指令說明的方式：

- `cmd /?` 和
- `help cmd`

注意第一種方式的 `/` (slash) 字元在 Linux/MacOS 是用作目錄分隔符號。

在 Linux/MacOS 環境下，則簡單的輸入：

- `man cmd` 就行；`man` 來自 **手冊**的英文 `MANual`；可以先
- `man man` 查詢 `man` 指令本身的用法。

5 Atom 基本操作

這一節簡單介紹如何利用 atom 編輯器編輯 sokoban.js 的檔案。

5.1 Open Folder

啟動 atom 後，先到左上角的 File 選擇 Open Folder。如 Figure 12

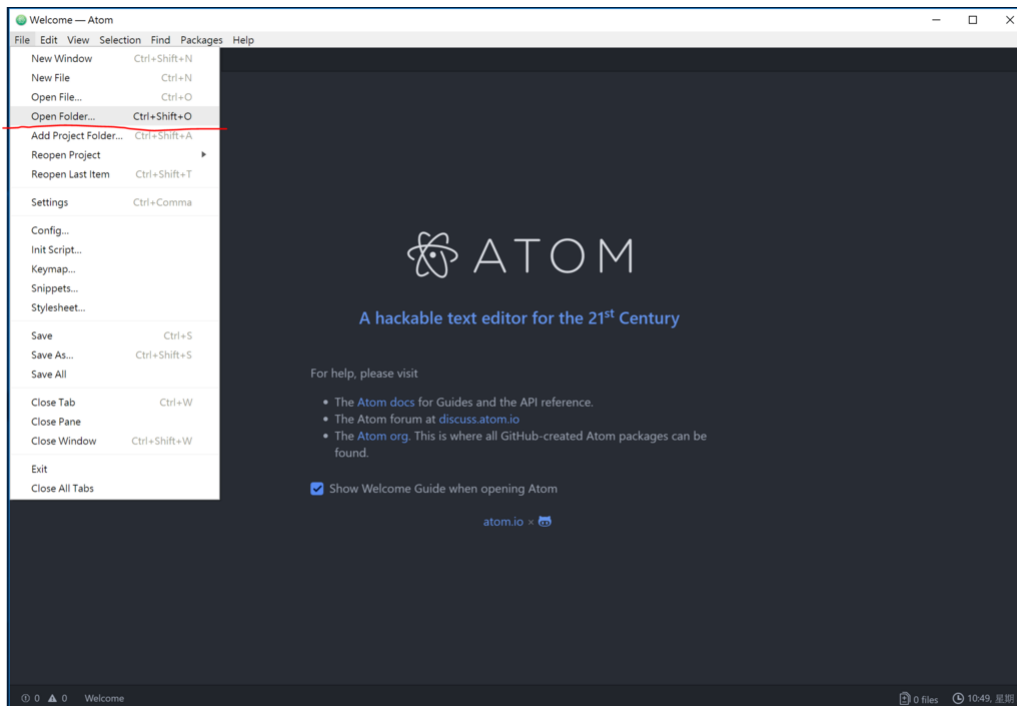


Figure 12: atom: 開啟資料夾

在檔案總管視窗選擇了專案資料夾後，畫面會變成如 Figure 13 所示。

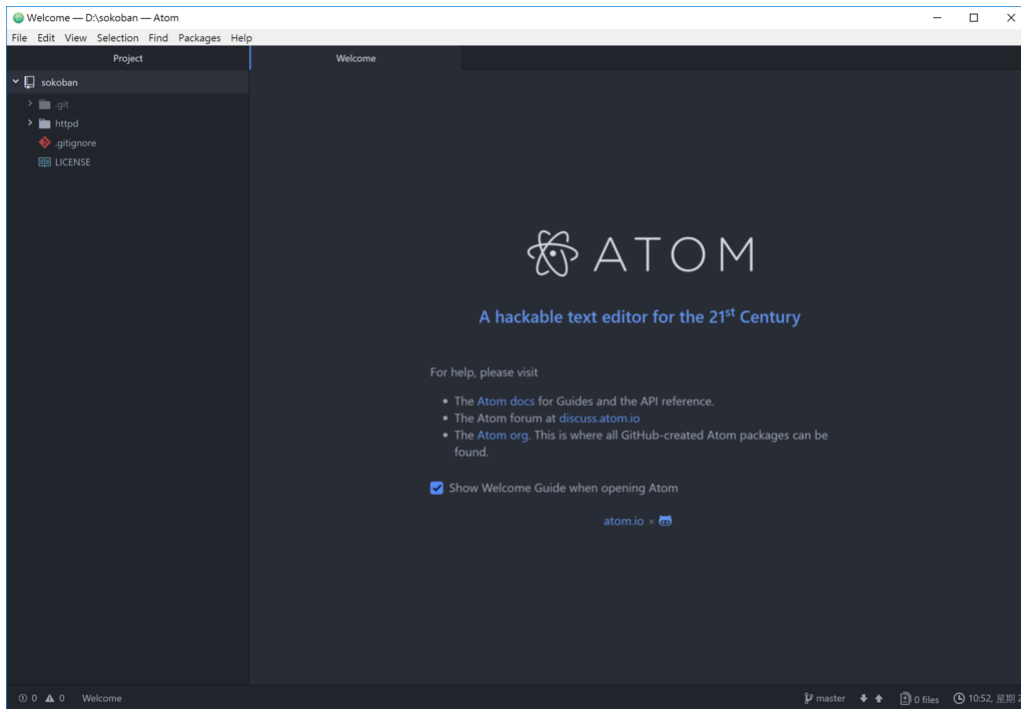


Figure 13: atom: project

5.2 New file

如果專案的目錄結構已建好，可以在畫面右上角的 httpd 資料夾點右鍵，選擇 Figure 14 裡的紅圈，

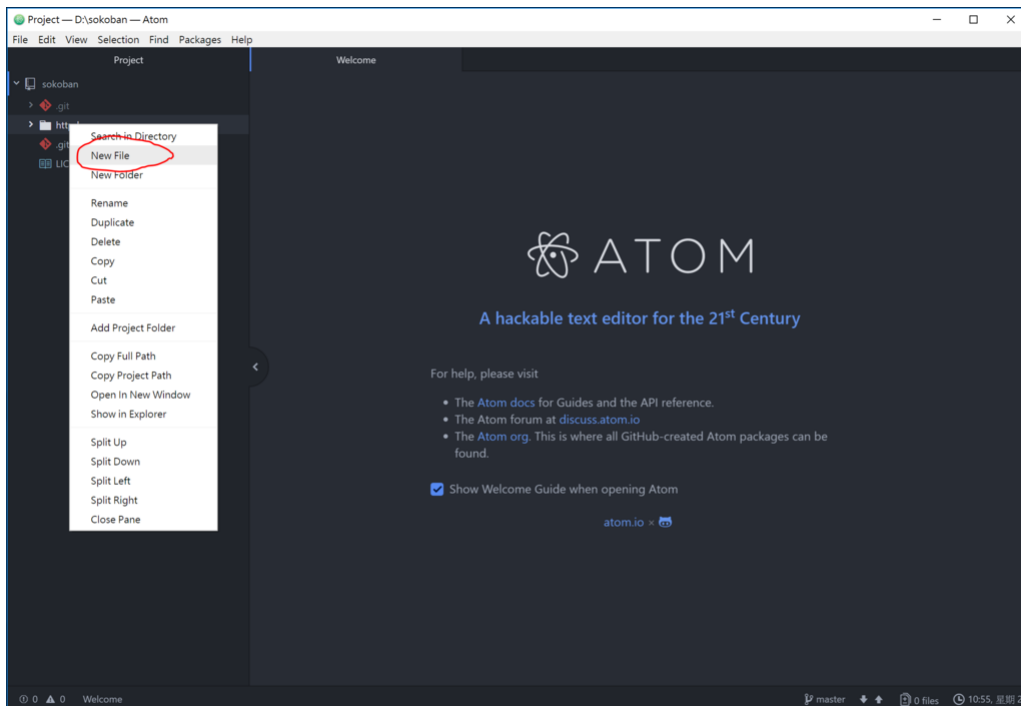


Figure 14: atom: project

然後在 15 裡輸入 index.js

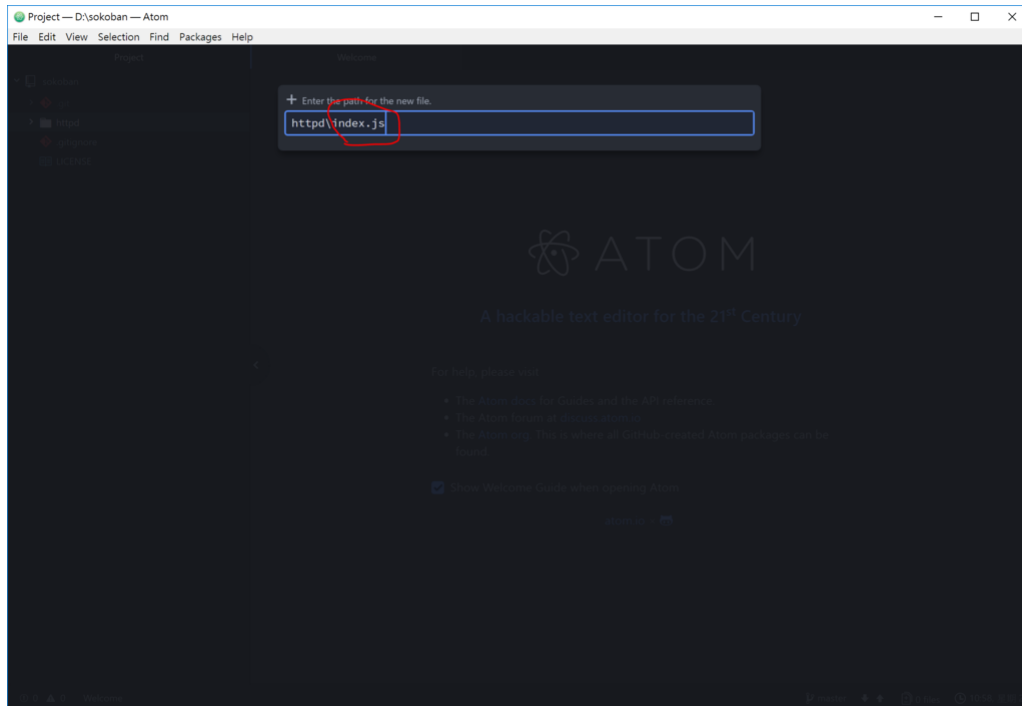


Figure 15: atom: project

就可以快樂的開始編輯 index.js 檔案了。

5.2.0.1 Happy Coding!