附錄 — 開發工具簡介

喬逸偉 (Yiwei Chiao)

1 導論

程式開發過程裡一件最重要的事就是知道目前電腦裡發生了什麼事，程式設計師才能由此找出程式的錯誤或改變程式行為。

現代的網頁瀏覽器，因為網頁應用程式的普及，也都內建了讓程式設計師了解網頁內部行為的工具，以協助網頁程式設計師完成工作。這裡以 Google Chrome 為例，一窺瀏覽器在這個面向可以給我們什麼幫助。

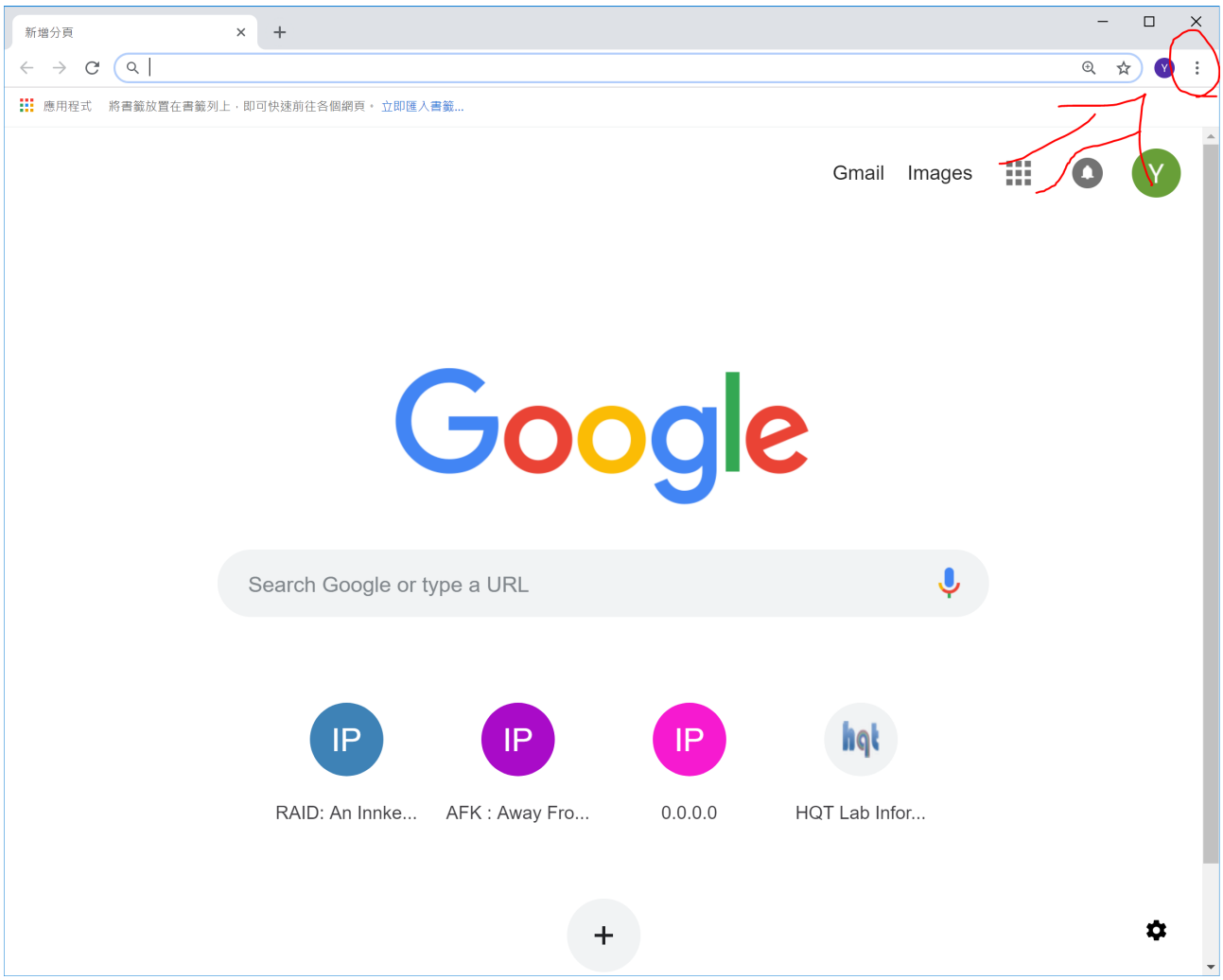

Figure 1: Chrome 畫面

1.1 專案準備

為了了解 Chrome 的開發人員工具，請先準備好下面的檔案：

- `index.html`: 放在 `breakit/htdocs` 資料夾下。
- `styles.css`: 放在 `breakit/htdocs/asssets/css` 資料夾下。
- `breakit.js`: 放在 `breakit/htdocs/js` 資料夾下。

這三個檔案的作用：

- `index.html`: 使用者瀏覽/網路爬蟲爬梳時，看到的網頁頁面。記錄了基本的網站資訊，如文字編碼，主題資訊等。也作為通知瀏覽器，後續 Web 資源，如 `.js`，`.css` 等檔案的 `url` 資訊。
- `styles.css`: 網頁的設計風格設定檔。網站的視覺風格由 `.css` 檔案決定。一個好的網站設計應該可以利用切換不同的 `.css` 檔作到不同的視覺呈現。
- `breakit.js`: Breakit 專案的客戶端程式。`.html` 提供了頁面的骨架，`.css` 為骨架加上了衣服，而 `.js` 是血肉。有了 `.js`，網頁才真正有了生命。

三個檔案準備好了以後，啟動 `breakit/httpd` 下的網頁伺服，可以準備來看看 Chrome 的開發工具。

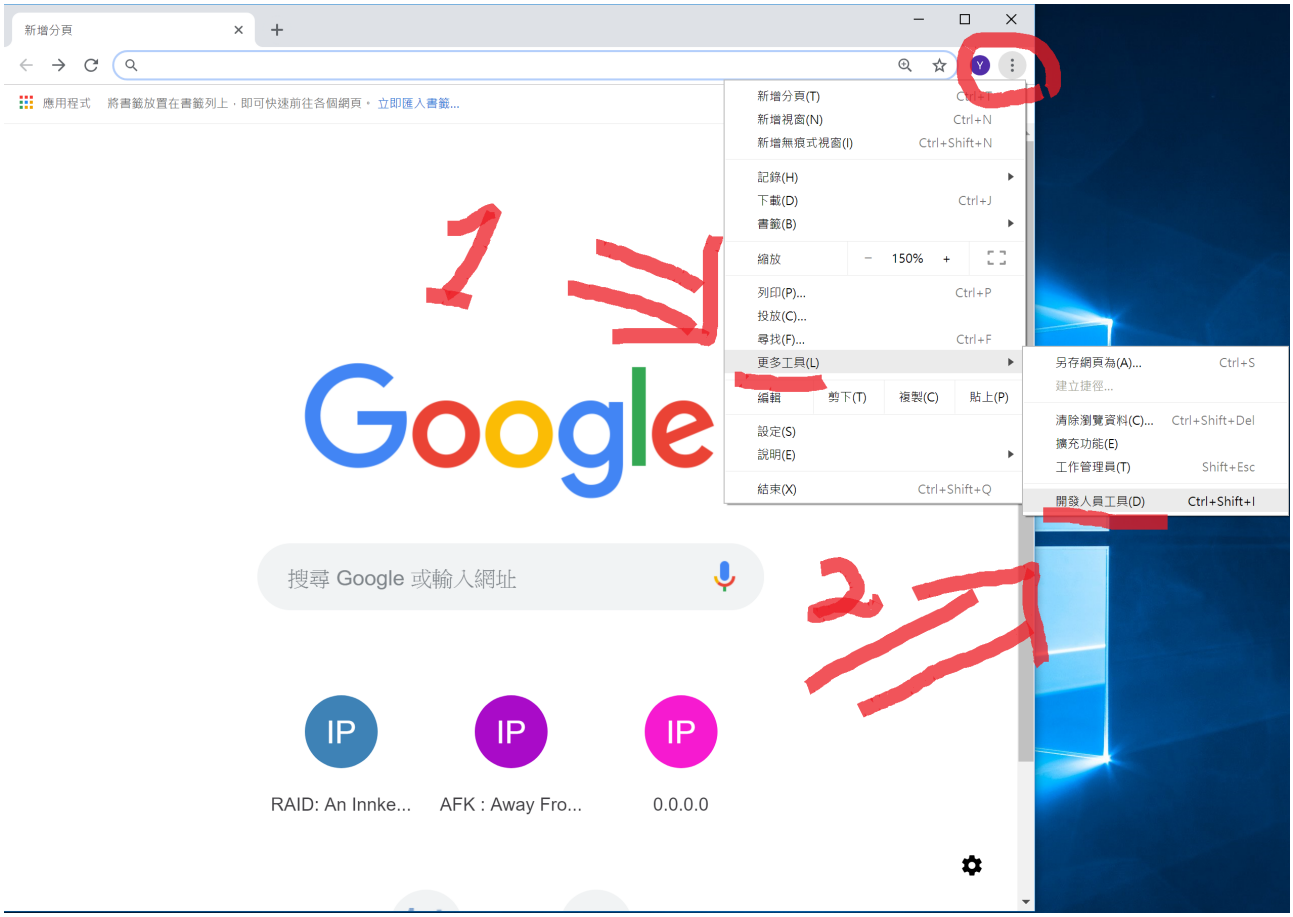

Figure 2: Chrome 開發人員工具

1.2 Chrome 開發人員工具

啟動 Chrome 瀏覽器，如圖 Figure 1，注意畫面右上角的按紐。那裡是開啓 Chrome (Firefox 也是) 瀏覽器設定的地方。

打開後，如圖 Figure 2，找到 [開發人員工具]，開啟它。開啟後，瀏覽器的畫面應該很類似圖 Figure 3 的樣子。先如圖 Figure 3 所示，找到 [network] 標籤下的 [Disable Cache] 將 Chrome 的**快取** (*cache*) 保持 **關閉** (*disabled*)，以確保網頁開發過程中，瀏覽器執行的確定是最新修定的版本。

同樣如圖 Figure 3 所示，[network] 標籤下，可以看到瀏覽器和伺服器間的資料傳輸網路延遲等資訊，那些在開發大型網站應用作優化時是很重要的資訊。不過目前知道有它存在就好，暫時可以不用管它。

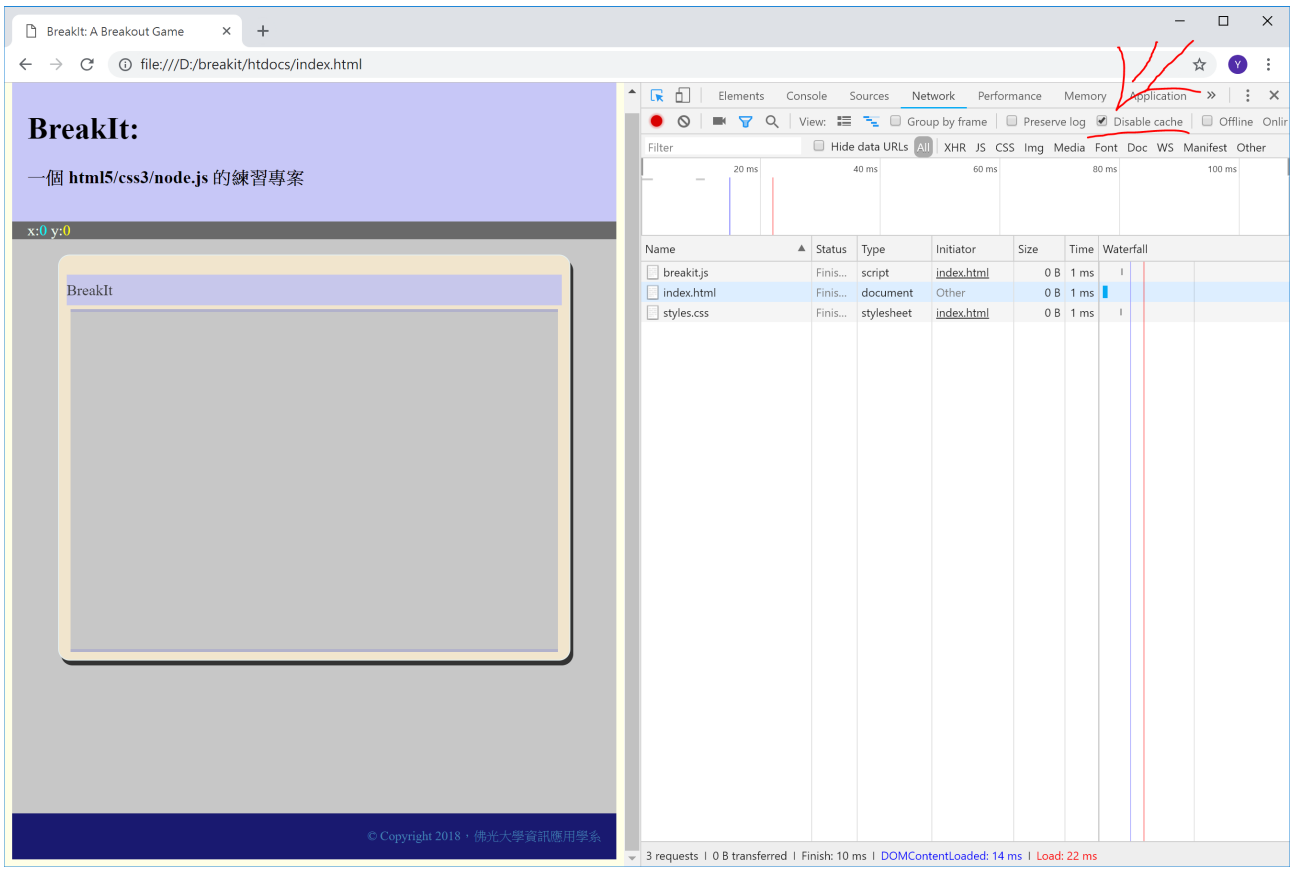

Figure 3: Chrome 開發人員工具設定

將快取關閉後，就可以回頭來看最常接觸的兩個標籤：[Elements] 和 [Console] 了。

1.3 Chrome [Elements] 標籤

[Elemtns] 指得是 HTML 和 CSS。在這個頁籤，可以看到 [開發人員工具] 上半部的畫面，顯示的是 HTML 的內容；而下半部則是 CSS 的樣子。

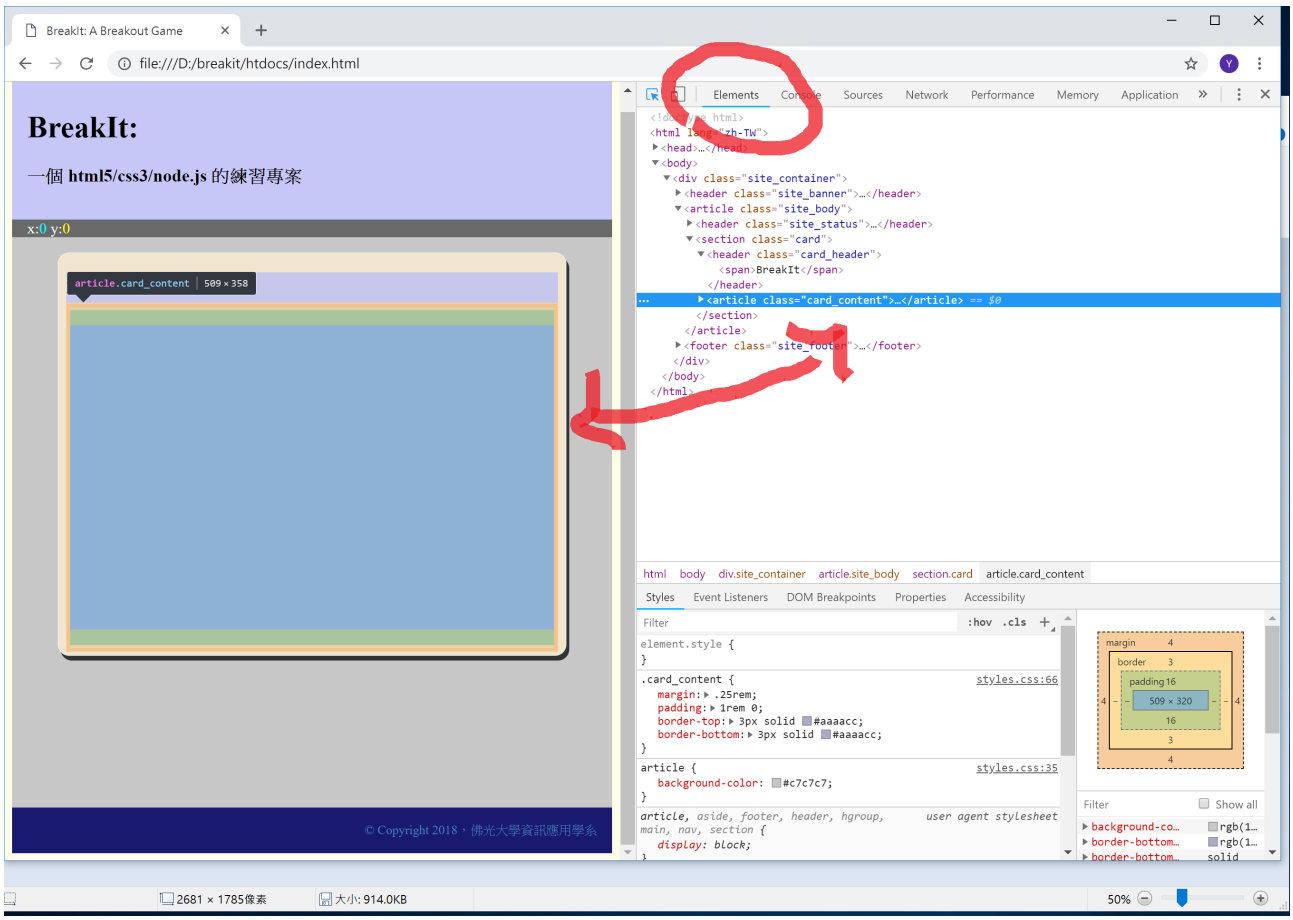

Figure 4: Chrome Elements

如圖 Figure 4 所示，試著在 [Elements] 顯示的 HTML 標籤上移動滑鼠，可以注意到畫面左邊也有視覺變化的效果。那是 Chrome 在標示滑鼠所在的 HTML 標籤，在網頁上呈現的效果和範圍大小。

所以，有這個頁面協助，設定 HTML 與 CSS 就不用再憑空想像，而可以實時看到效果。

1.4 Chrome [Console] 標籤

在寫 Node.js 程式時，可以利用 `console.log(...)` 在螢幕上輸出訊息以理解程式內部實際發生的事情；同樣的 `console.log(...)` 在瀏覽器裡，就是輸出到這個 [Console] 頁籤。

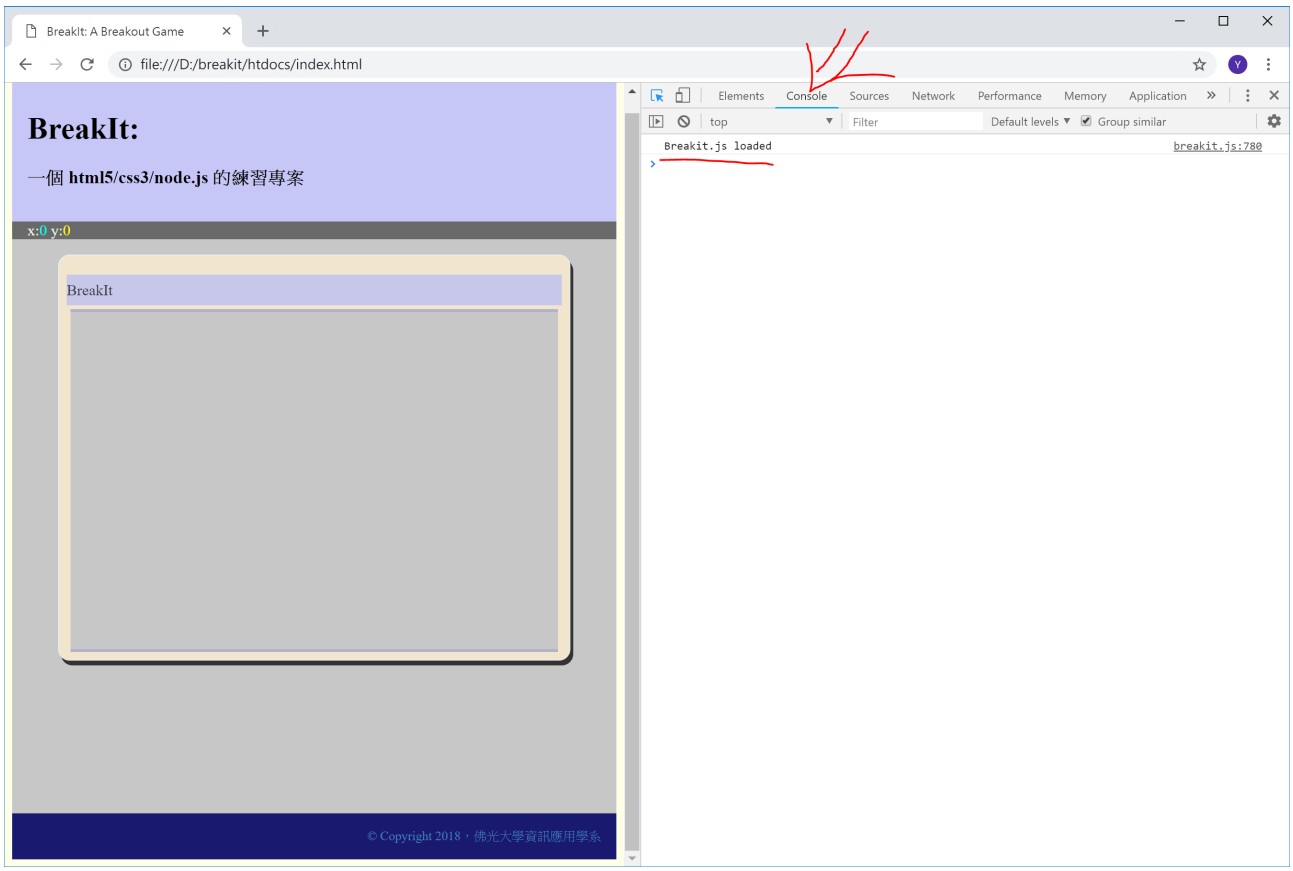

Figure 5: Chrome Console

可以打開 `breakit/htdocs/js/index.js` 看到程式一開始就有一行 `console.log(...)`，和圖 Figure 5 裡顯示的相同。

```
window.addEventListener('load', () => {
  console.log('Breakit.js loaded');
};
```

而如果網頁程式執行有**錯誤**發生，Chrome 的 [console] 頁籤會如圖 Figure ?? 所顯示。右上角會有紅色的數字顯示程式中止前的錯誤個數；而 [Console] 視窗則會顯示出錯的程式碼和它的 `.js` 檔名與行號。

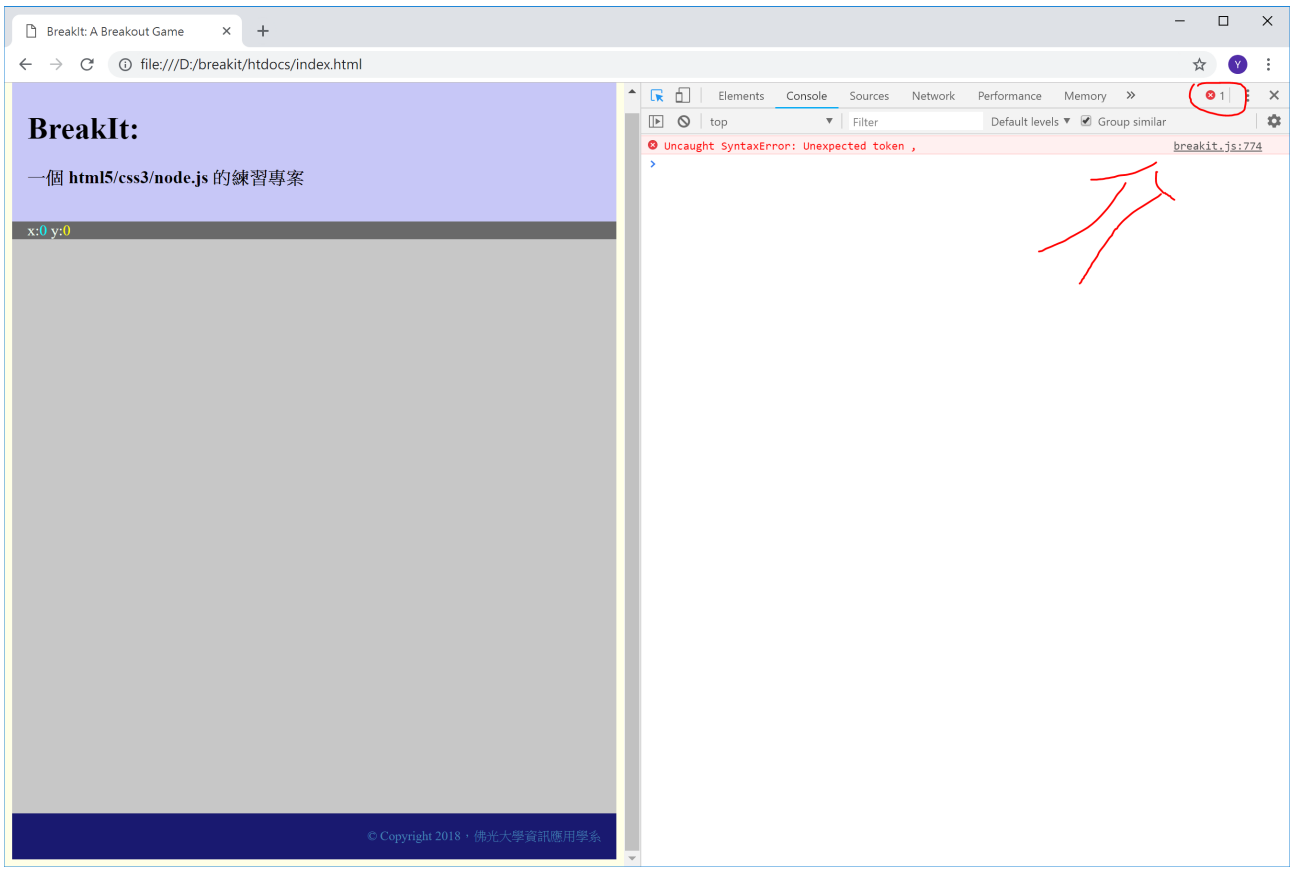

Figure 6: Chrome Console 錯誤視

1.5 問題與練習

1. 在 [Elements] 裡移動滑鼠游標，嘗試理解它的顯示和作用。
2. 利用文字編輯器 (如 `atom`)，打開 `htdocs/index.html` 比較它的內容和 [Elements] 顯示的內容。好像有些不大一樣？將 `htdocs/js/index.js` 裡 `console.log(...)` 後的內容都註解掉，再比對看看內容是否相同？研究一下？

2 DOM (Data Object Model) 背景

要寫作網頁應用程式 (WebApp)，在客戶端 (網頁瀏覽器) 有三大支柱：* HTML：負責**文件** (*Document*) 結構 * CSS：負責**文件**排版 * JavaScript：負責**文作**操作 (*manipulate*) 一個有趣的問題就出現了。HTML 和 CSS 都是簡單的文字檔案。JavaScript (或任何其它程式語言) 當然可以將它們當作 **文字** (*text*)，或說**字串** (*string*) 來處理。事實上，在伺服端 (網頁伺服器)，所謂的**後台** (或稱**後端**，*backend*) 程式，如 PHP，JSP，Python，Ruby 等開發工具就是這麼作的。甚至還開發了專門的**樣本語言** ([*template engine*][wikiTemplateEngine]) 來作這件事。例如，給 Node.js 用的 Jade，Python 的 Jinja2，Ruby 內建的 ERB，PHP 的 Twig 等。

但是，現在是在**前端** (或稱 **前台**，*frontend*)；無論是 HTML 或 CSS 都已經 (也必需) 分析轉換成瀏覽器的 **內部** (*internal*) 表示型式。所以在前端最好的方式應該是直接和瀏覽器溝通。直接操作解譯過的 HTML 物件。

這個讓外部程式可以直接操縱瀏覽器解譯後的 HTML，CSS 物件的標準，就是 DOM 應用程式介面 (api)。

2.1 DOM 簡介

DOM 全稱是 *Data Object Model^ (**資料物件模型**)；設計用來處理和表示 HTML，SVG 和 XML 文件的 Web 公開標準。

DOM 背後的骨幹概念很簡單而直覺。DOM 將文件視為一棵**樹** (*tree*)，文件內的結構則視作這棵樹的**分支** (*branch*)，最後的內容，自然是**樹葉** (*leaf*)，稱作**節點** (*node*)。因為 DOM 將文件視為一棵樹，所以後面會用 **DOM 樹**或 *DOM tree* 來稱呼某個 HTML 文件的 DOM 型式。

2.2 DOM 和 HTML 文件

一個簡單的例子，考慮下面這個簡單的 `.html` 檔案:

```
<html>
  <body>
    Hello World!
  </body>
</html>
```

以 DOM 模型來表示，大概長成這樣：

```
window.document
  |
  + body
  |   |
  .   TextNode
  .
```

和原來的 HTML 對照，應該可以看到明確的一一對應。而上面列表中的 `window.document` 就是 JavaScript 在處理網頁文件時的**根** (*root*) 物件。其中的 `window` 代表的是瀏覽器視窗 (viewport)；是真正的**瀏覽器物件**；也就是說，`window` **不是** HTML 物件的一部份，它的存在是作為一個容器，將瀏覽器和外來的 HTML 文件結合在一起，就是 `window.document` 這個**屬性**裡存放的物件才是真正 HTML 文件。一般在 JavaScript 裡，可以直接寫 `document` 來存取它的方法。

在 body 下面的 TextNode 就對應到 HTML 裡的 Hello World! 因為 Hello World! 不是個 HTML 的**標籤** (*tag*)，而是普通的文字內容，所以 DOM 模型設計了一個 **TextNode** 節點物件來存放它。

2.3 DOM 實作練習

來試試 DOM 的實際操作。先在專案裡建立一個**測試** (*test*) 資料夾，如圖 7：

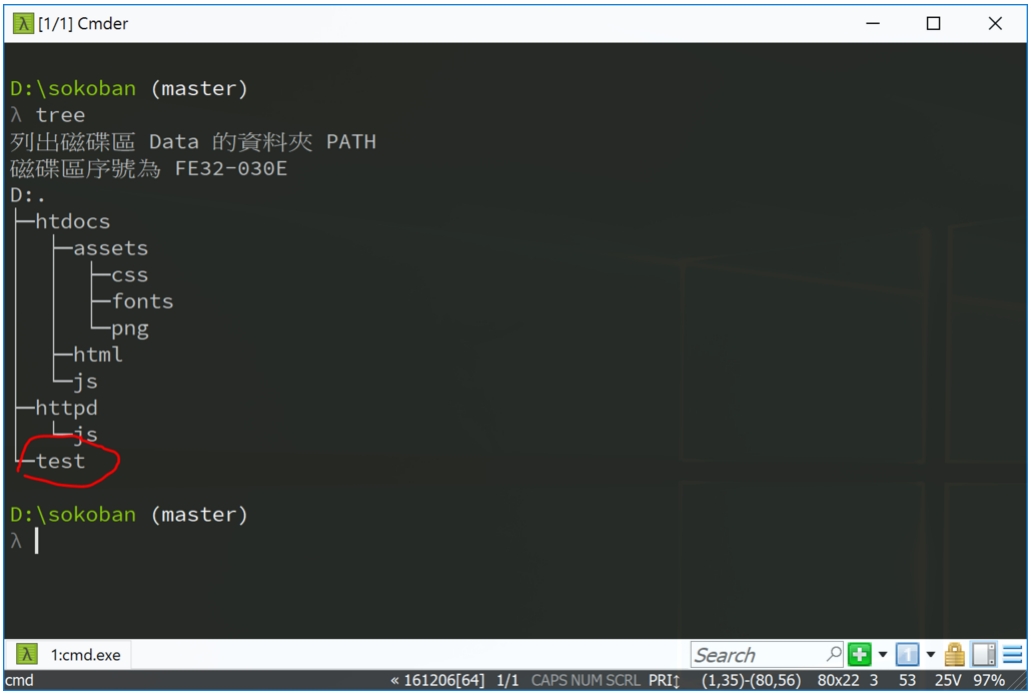

Figure 7: 專案資料夾

將下面的程式碼放到 test/index.html：

```
1. <html lang="zh-TW">
2.   <head>
3.     <meta charset="utf-8">
4.     <script src="index.js"></script>
5.   </head>
6.   <body>
7.   </body>
8. </html>
```

再將下面的程式碼放到 test/index.js：

```
1. 'use strict';
2.
3. window.addEventListener('load', () => {
4.   console.log("index.js loaded");
```

```
 5.
 6.    let h1 = document.createElement('h1');
 7.    let msg = document.createTextNode(' 這是 <h1> 的文字訊息');
 8.
 9.    h1.appendChild(msg);
10.
11.    document.body.appendChild(h1);
12. });
```

利用 Chrome (或 Firefox) 打開 `file:///d:/breakit/test/index.html` (記得將前面的網址修改成適合當下電腦配置。) 應該會看到類似圖 8 的畫面。

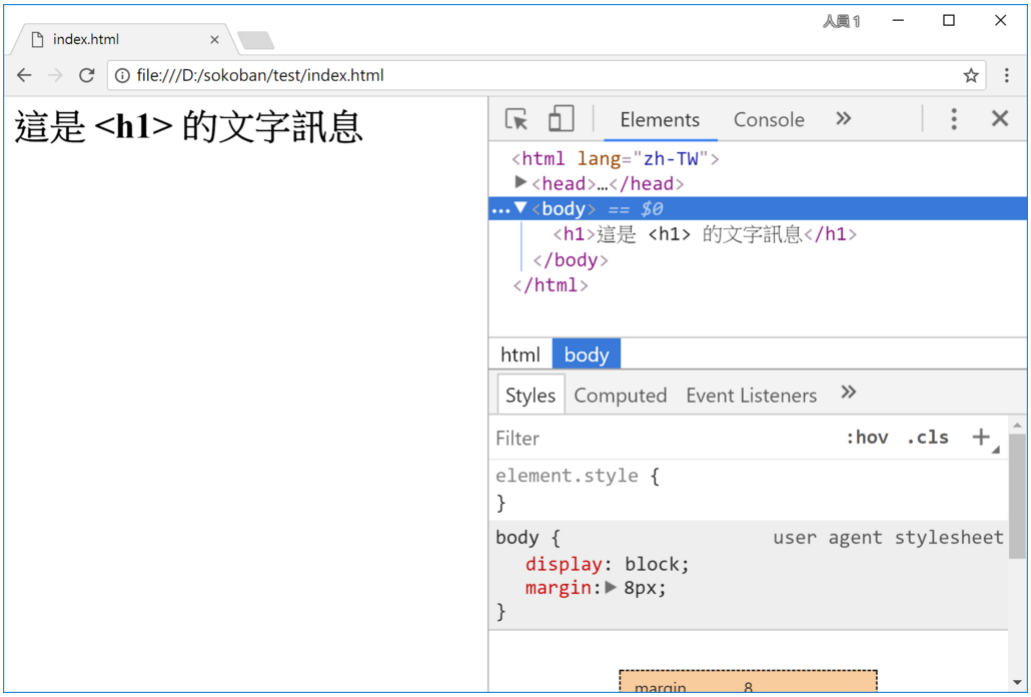

Figure 8: test.index/test.js

回頭看 `index.html` 的源碼，應該可以看到它原來應該是一個空白的網頁；或者，可以將 `index.html` 的第四 (4) 行 (載入 `.js` 檔案那行) 改成如下的型式：

```
 4.    <!--script src="index.js"></script-->
```

也就將第四 (4) 行**註解** (*comment*) 掉。再載入一次，看到的應該是空白畫面。因為畫面上的訊息是由 `index.js` 裡的 [JavaScript][mdnJavaScirpt] 直接操作 DOM tree 產生的。所以如果將載入 `index.js` 的源碼註解掉，程式沒載入，畫面自然回到空白的狀態。

2.4 程式說明

因為這是我們和 JavaScript 的第一次接觸，讓我們停下來仔細看一下這兩個檔案：`index.html` 和 `index.js`。

2.4.1 index.html

首先是 index.html。由上面的程式碼可以看到 index.html 裡，

```
6.    <body>
7.    </body>
```

第六行 <body> 和第七行 </body> 標籤裡是**沒有**任何內容的。造理說，網頁應該是空白的。如前面建議的，如果將第四行

```
4.        <script src="index.js"></script>
```

註解掉

```
4.        <!--script src="index.js"></script-->
```

網頁也真的如預期的變成了空白 [^cmtTag]。 HTML 裡，<!-- ... --> 稱為**註解** (*comment tag*) 標籤，夾在 <!-- 和 --> 間的內容不會被瀏覽器 (browsers) 處理和顯示。

明顯的，關鍵就是第四行：

```
4.        <script src="index.js"></script>
```

2.4.2 <script> tag

在 HTML 裡 <script> 是用來放置 scripting 程式碼的標籤；以目前來說就是放置 JavaScript 程式碼的地方。隨著 WebApp 的發展與成熟，現在的建議 (best practice) 是不直接將程式碼放在這個標籤裡；而是利用 src 這個**屬性**，通知瀏覽器去另外下載指定的程式碼檔案來執行。

例如在這個例子裡，就利用 src=index.js 這個屬性設定，通知瀏覽器去取得 index.js 檔案來執行。

2.4.3 index.js

index.js 是我們看到的第一個 JavaScript 程式，含空白行，只有 12 行。所以可以在下一節，一行行的說明它的行為。

2.5 index.js 說明

2.5.1 JavaScript 程式起點

程式開始執行需要一個起點；C/C++/C#/Java 系列語言都有一個 main() 函數或 Main() 方法當作程式的進入點，開始執行的地方。

但是像 JavaScript 這樣的**劇本式語言** (*scripting languages*，其它如 Python, Ruby 等) 就不仰賴，也無法仰賴在特定的方法/函數上；它們是簡單的遵循程式碼的安排，由**第一行**開始，**循序執行**。

2.5.2 JavaScript `strict` 模式

`index.js` 的第一行，宣告了一個字串：

```
1. 'use strict';
```

在 JavaScript 裡' (單) 和" (雙) 引號都可以用來指定**字串** (*string*)。當然，這些標明字串的引號**必需**是成對出現的。

`'use strict';` 後面的分號 `;` 標明一行的結束；JavaScript 其實並**不**要求這個分號 `;`；因為絕大部份時間，JavaScript 執行引擎會自行推斷出每一行指令的結束。不過為了和寫 C/C++/C#/Java 系列程式的習慣相容，我們還是會放上分號 `;`。

2.5.2.1 strict mode

`'use strict';` 其實也不是一個 JavaScript 的指令/陳述。它其實是一個和 JavaScript **執行環境**溝通用的 **標記字串**。

如同之前提過的，JavaScript 這些年迎來了重大的標準化進展；和任何重大變動一樣，這無可避免的造成語言新舊版本的相容問題。為了降低衝擊，JavaScript 引入了所謂的 strict 模式；使用新版 JavaScript 語言標準的程式自動遵循 strict 模式；而其它使用舊版標準；或混合式 (某些部份使用新版，有些用舊版) 的程式則可以利用 `'use strict'` 這個字串來通知 JavaScript 後續程式碼使用新版語言規範。

2.5.3 window

```
3. window.addEventListener('load', () => {
...
12. });
```

`window` 是 DOM 模型裡代表瀏覽器視窗的介面物件；整個 JavaScript 程式就是在這個物件代表的環境下執行。

這裡，利用 `window` 提供的 `addEventListener` 介面函數，在 `window` 註冊了一個 `load` 事件的**事件處理函式** (event listener/handler)。

2.5.3.1 `load` 事件

由之前對 HTML 的介紹可以注意到，當瀏覽器開啟一個網頁時，它必需要下載相關的 *.html (頁面文件結構)*, .css (頁面風格設計)，*.js (可能的互動控制) 和其它資源 (字型，圖片 ...) 等。

`load` 事件就是設計來通知 `window` 物件，它使用的資源 (`window` 將要呈現的 HTML 內容，和相關資源) **已經**下載完成。

所以，在 `window` 物件上註冊 `load` 事件的處理程序就是在瀏覽器下載完相關資訊後，開始**接手**內容的處理。

2.5.3.2 `() => {...}`

`() => {...}` 是一個**匿名**函式宣告；也就是我們用來處理 `load` 事件的事件處理程序。

傳統的 JavaScript 寫法也可以寫成

```
function () {...}
```

不過利用新版 JavaScript 的**箭頭函數** (*arrow function*) 寫法，感覺更簡潔。

第 4 到 11 行就是這個函數的內容。

2.5.4 `console`

```
4.    console.log("index.js loaded");
```

第 4 行的 `console` 是 JavaScript 執行環境提供的命令列介面物件；最簡單直接的用途就是利用它在命令列介面輸出一些除錯用的信息；如第 4 行作的。

`console` 的 `log` 方法可以在命令列介面輸出一個訊息；第 4 行這裡輸出一個簡單的訊息字串：`index.js loaded` 指出在第 3 行註冊的 `load` 事件處理函式不但註冊成功，而且已被呼叫。

而這個輸出結果會在瀏覽器開發者工具的 `console` 頁面出現；但**不會**在使用者看到的 window 裡出現。

2.5.5 動態型別與 let 變數宣告

和 `C/C++/C#/Java` 等**靜態型別** (*static typed*) 語言，所有變數宣告都需要**指定**變數**型別** (type) 不同；JavaScript 是**動態型別** (*dynamic typed*) 語言 (參照 Python，Ruby 等) 語言，變數型別會由程式在執行時**自動**辨別，指定和使用。

因此，在 JavaScript，傳統上，變數是**不**需要宣告的。

但是變數除了型別 (type) 外，還有一個**有效範圍 (作用域)** (*scope*) 的問題存在；也就是特定變數在程式的那些區段內可以讀取，使用；又在什麼時後生成，什麼時後摧毀。

傳統 JavaScript 在這塊的處理，以最簡單的說法來說，是令人困惑的。

標準化的 JavaScript 引入了 let 變數宣告來處理這個問題。

2.5.5.1 let

```
 6.    let h1 = document.createElement('h1');
 7.    let msg = document.createTextNode(' 這是 <h1> 的文字訊息');
```

let varName 宣告 varName 是一個變數，而 varName 的**作用域**就是 let varName 所屬的程式區塊 ({}) 在 let varName 這一行以下及延伸的範圍。何謂**延伸的範圍**，我們在之後遇到時再討論。以目前而言就是

. 變數 h1 在第 6 行宣告，它的作用域是第 6 行到第 11 行。

. 變數 msg 在第 7 行宣告，它的作用域是第 7 行到第 11 行。

2.5.6 document

document 是 **DOM** 模型裡代表 **HTML** 文件的物件；也是 DOM 數的根節點。在 Web API (應用程式介面) 的設計裡，可以利用 document 的 createElement(...) 方法，動態產生想要的 DOM 節點 (HTML tag 元素) 或文字節點 (TextNode) 來安放文字內容。

```
 6.    let h1 = document.createElement('h1');
```

產生一個 <h1> 的 HTML 節點，由變數 h1 記錄。

```
 7.    let msg = document.createTextNode(' 這是 <h1> 的文字訊息');
```

產生一個內容是 這是 <h1> 的文字訊息的文字節點 (TextNode)，由變數 msg 記錄。

2.5.6.1 方法 appendChild(...)

DOM 節點產生了，也保留在變數裡了，可是它們都還獨立放 DOM 樹之外；還沒有和 document 產生連結；所以下一步就是要將這些新產生的節點安插到 DOM 樹內。

由之前對 HTML 與 DOM 的介紹，可以知道，這樣的 HTML 程式碼：

```
<h1>這是 <h1> 的文字訊息</h1>
```

對應的是這樣的結構：

```
+ h1
  |
  + TextNode
    |
    這 是 <h1> 的 文 字 訊 息
```

所以變數 msg 代表的 TextNode 應該是變數 h1 代表的 <h1> 節點的**子節點** (*child node*)

DOM 提供了 ParentNode.appendChild(childNode) 這個介面方法，讓我們可以將子節點添加到**親節點** (*ParentNode*) 之下。所以，我們有：

```
 9.    h1.appendChild(msg);
10.
11.    document.body.appendChild(h1);
```

先將 msg 加入 h1 (第 9 行)，構成

```
  + h1
    |
    + TextNode
      |
      這 是 <h1> 的 文 字 訊 息
```

再將 h1 加入 document.body 構成：

```
window.document
  |
  + body
      |
      + h1
        |
        + TextNode
            |
            這 是 <h1> 的 文 字 訊 息
```

2.6 思考與練習

- 對 DOM 的操作有基本概念了，範例程式介紹的是 <h1>；試試再加上幾個，如 <h2> 或 <h6> 的訊息。
- 作上面的練習時，觀察一下瀏覽器除錯視窗 Element 窗口的訊息變化。
- 查一下網路資訊，找找如何由 DOM tree 裡：
  - 移除一個節點
  - 將某個節點由當位置搬到新位置。