第一章

客戶端搜尋引擎

應用程式特點：

- 效率化搜尋引擎
- 各式搜尋演算法
- 搜尋結果排序和分頁
- 紀錄筆數不限
- 相容於 JavaScript 1.0

JavaScript 實務

- 串接字串可含多項資料
- 巢狀 for 迴圈
- 善用 document.write()
- 條件運算式

任何網站皆可支援搜尋引擎，但何苦讓伺服器承攬所有的查詢業務？客戶端搜尋引擎即為此而生，讓訪客的搜尋查問全部在客戶端完成作業。這是因為訪客發出查詢請求之後，下載回來的網頁內箝了必需的資料庫，這一點和以往把查詢字串送往遠端伺服器，交由應用程式進一步比對資料庫之後，再由伺服器予以回應的做法，是全然不同的設計概念。這樣如影隨形的假資料庫其實是用 JavaScript 的陣列模擬出來的；資料庫中的每筆紀錄（record）都是陣列的一個元素。

這種做法顯然有許多不可磨滅的優點，最主要的好處是降低伺服器的工作量，提昇回應訪客請求服務的效率。聽起來不錯，但請謹記在心，應用程式仍然受限於客戶端本身能提供的系統資源，其中以 CPU 處理速度和可用記憶體空間影響最為深遠。無論如何，客戶端搜尋引擎對網站而言，絕對是不可多得的好幫手。本章提列的程式碼儲放在 ch01 資料夾，讀者可自取並測試之。圖 1-1 正是此搜尋引擎的網頁入口。

圖 1-1：搜尋引擎入口

客戶端搜尋引擎支援兩種布林搜尋法則：AND 和 OR。搜尋時輸入的關鍵詞不外乎文件標題、說明文字或者文件的 URL。從程式功能的觀點來看，實在沒什麼好多說的了。訪客只要輸入想查詢的資料項（term），再按下 <Enter> 鍵，搜尋的結果倏忽呈現眼前。下面是搜尋選項的分析：

- 輸入資料項之間若以空格分開，則任何一筆紀錄只要含有輸入資料項的其中一項，就會被當成查詢結果傳送回來（布林 OR）。
- 對前一條法則而言，若於最左端資料項的前端加上加號（+），則資料庫的紀錄必須含有所有輸入的資料項，才能符合搜尋條件的要求（布林 AND）。
- 輸入的資料項若為 url：URL（完整或不完整），則傳回的紀錄，將含有指定的 URL。

別忘了序文中提到的原始碼 zip 壓縮檔！本書所有的程式碼都可從 O'Reilly 的網站下載：http://www.oreilly.com/catalog/jscook/index.html。

圖 1-2 是做了簡單查詢之後，得到的回應網頁（result page）。查詢所用的關鍵詞是 javascript，搜尋法則採用預設法則布林 OR，關鍵詞前不綴加號（+），這一點值得注意。每一次搜尋都會給出回應網頁，羅列出符合搜尋條件的紀錄資料，每一頁網頁的右上方都有一個 Help 的鏈結，作為操作指引之用。

能夠以 URL 來搜尋資料也是很貼心的設計。圖 1-3 正是用了 url: 搜尋得出的視窗畫面，指示搜尋引擎只要針對 URL 求取資料即可。對此而言，html 正是輸入的關鍵詞，因此搜尋引擎把含有 html 字串的文件統統列出來。文件的說明文字仍然秀了出來，就在 URL 之後。URL 搜尋法則限定在單一比對條件之下，如同預設法則那般。這當然不是什麼缺點，畢竟，會去研究複雜搜尋法則的人不多啊。

搜尋引擎每頁顯示的資料筆數可加以限定，然後設計按鈕，讓訪客得以前後瀏覽查詢結果。如果不加分頁，讓一大疊資料魚貫排列在同一張網頁，容易眼花的訪客恐怕難逃自掘墳墓的結局。至於每張網頁顯示的筆數應該在多少上下，則依個人喜好而定，一般約在 10 筆左右。

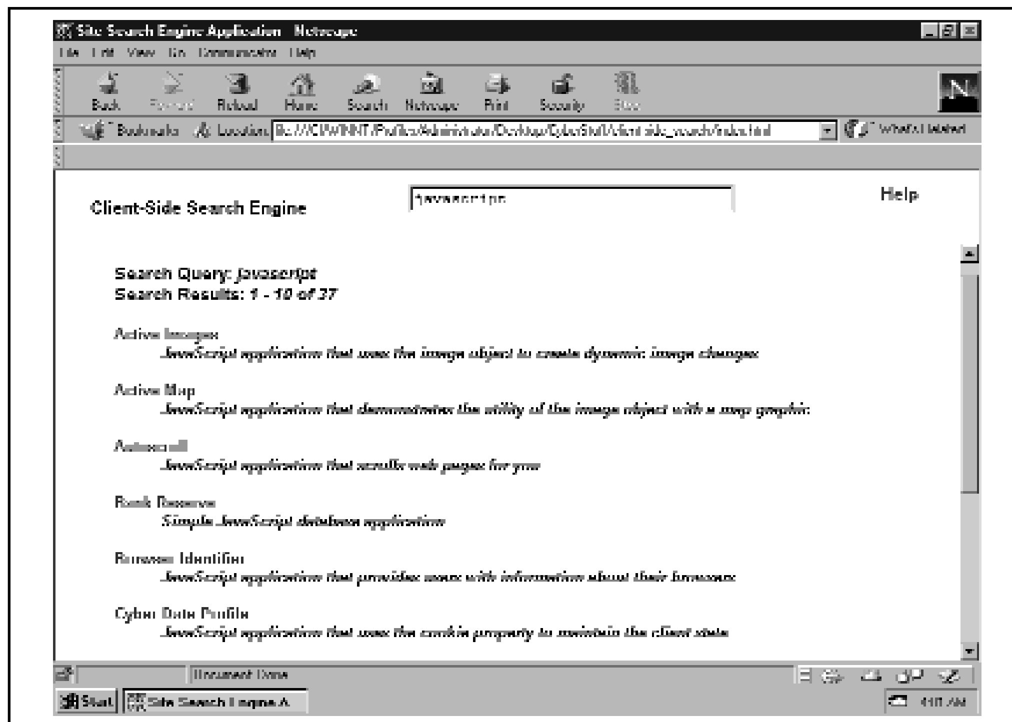

圖 1-2：典型的搜尋結果

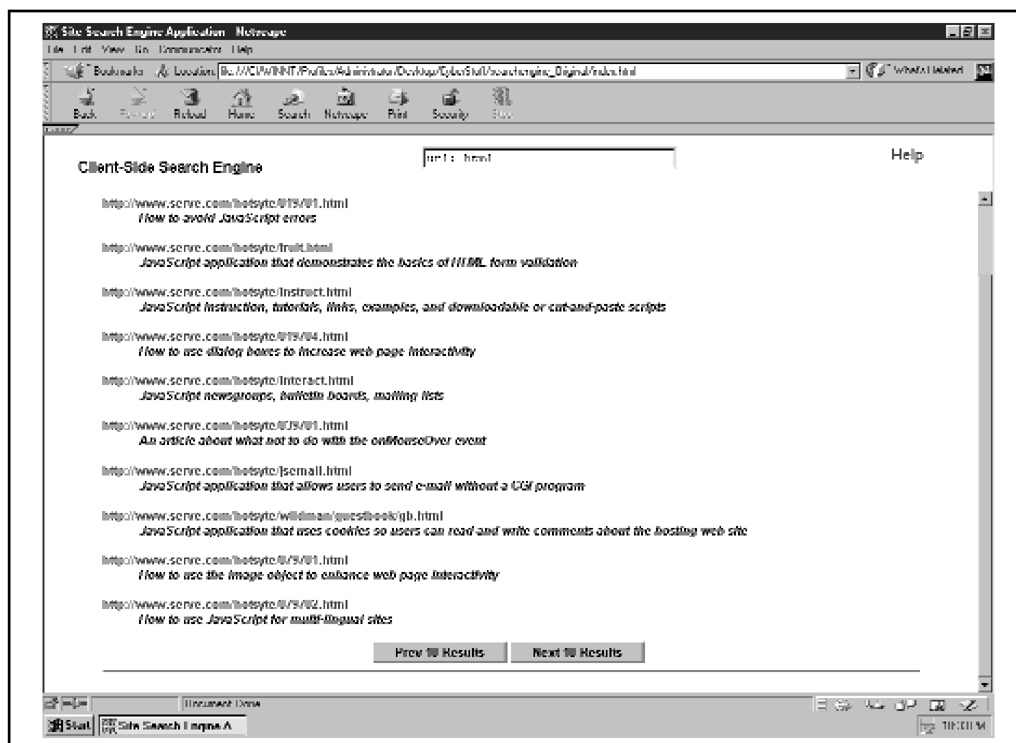

圖 1-3：URL 的搜尋結果

系統需求

瀏覽器必需支援 JavaScript 1.1 才能執行此應用程式。NN 3~4 版和 MSIE 4~5 版都適用，但 MSIE 3 可就回天乏術了。如果打算發展向下相容的程式碼，讓那些還在用 MSIE 3 的訪客也能夠享用客戶端搜尋引擎帶來的優點，請參考本章「成品加工」一節的說明。

客戶端應用程式的表現能力，完全依賴客戶端系統的機器性能而定，這一點對本章提舉的搜尋引擎而言，更是實質的核心問題。然而，我們大可打包票，相信客戶端系統有最低限度的資源足以圓滿執行程式。但是，如果傳送給客戶端的資料庫過於龐大（差不多超出 6000 筆紀錄），恐怕會造成 JavaScript 程式的執行效能曲線陡降，最終迫使客戶端系統陷於疲於奔命的窘境。

我自己用 MSIE 4 和 NN 4 做測試，下載的資料庫約莫近 10000 筆紀錄，測試結果沒什麼問題。有一點在此提出說明，JavsScript 的程式碼單單儲存這些資料，就超過了 1MB。我自己的機器每台配備的 RAM 都在 24MB 到 128MB 之間。若以 NN 3.0 Gold 做測試，結果會出現堆疊溢位（stack overflow）錯誤 —— 陣列的紀錄太多了。

想知道紀錄筆數的最低容許量，只要拿一台 IBM ThinkPad，配備 MSIE 3.02 和 JavaScript 1.0 的規格來檢視，即可得出 215 筆的極限值。不過，千萬別讓這樣低的數字給嚇倒了。ThinkPad 這種中古世紀出品的膝上型電腦已經過時很久了，那顆氣息微弱的 CPU 彷彿是依靠實驗室小白鼠努力推轉迴旋輪，才能發出陣陣的喘息聲。現代網際網路遊俠想必都有更新型、更強力的電腦設備了，這一點無庸置疑。

語法分析

客戶端搜尋引擎包括了三個 HTML 檔案（index.html、nav.html 和 main.html）和一個 JavaScript 原始碼檔案（records.js）。這三個 HTML 檔案定義了一個小小的窗框台（frameset），再由這個窗框台切割出導前窗框（frame）和預設窗框。導前窗框可輸入搜尋資料項，而預設窗框則指出搜尋引擎的操作細則（顯示視窗中的 HELP 鏈結）。

nav.html

客戶端搜尋引擎的心臟地帶是 nav.html。事實上，JavaScript 唯一現身之處，僅在搜尋引擎編製回應網頁時，才看得出端倪。範例 1-1 正是 nav.html 的程式碼。

範例 1-1：nav.html 的原始程式碼

```
1  <HTML>
2  <HEAD>
3  <TITLE>Search Nav Page</TITLE>
4
5  <SCRIPT LANGUAGE="JavaScript1.1" SRC="records.js"></SCRIPT>
```

範例 1-1：nav.html 的原始程式碼（續）

```
 6 <SCRIPT LANGUAGE="JavaScript1.1">
 7 <!--
 8
 9 var SEARCHANY  = 1;
10 var SEARCHALL  = 2;
11 var SEARCHURL  = 4;
12 var searchType = "";
13 var showMatches   = 10;
14 var currentMatch  = 0;
15 var copyArray    = new Array();
16 var docObj  = parent.frames[1].document;
17
18 function validate(entry) {
19   if (entry.charAt(0) == "+") {
20    entry = entry.substring(1,entry.length);
21    searchType = SEARCHALL;
22    }
23   else if (entry.substring(0,4) == "url:") {
24    entry = entry.substring(5,entry.length);
25    searchType = SEARCHURL;
26    }
27   else { searchType = SEARCHANY; }
28   while (entry.charAt(0) == " ") {
29    entry = entry.substring(1,entry.length);
30    document.forms[0].query.value = entry;
31    }
32   while (entry.charAt(entry.length - 1) == " ") {
33    entry = entry.substring(0,entry.length - 1);
34    document.forms[0].query.value = entry;
35    }
36   if (entry.length < 3) {
37    alert("You cannot search strings that small. Elaborate a little.");
38    document.forms[0].query.focus();
39    return;
40    }
41   convertString(entry);
```

```
42   }
43
44 function convertString(reentry) {
45   var searchArray = reentry.split(" ");
46   if (searchType == (SEARCHALL)) { requireAll(searchArray); }
47   else { allowAny(searchArray); }
48   }
49
50 function allowAny(t) {
51   var findings = new Array(0);
52   for (i = 0; i < profiles.length; i++) {
53     var compareElement  = profiles[i].toUpperCase();
54     if(searchType == SEARCHANY) {
55       var refineElement = compareElement.substring(0,
56         compareElement.indexOf('|HTTP'));
57       }
58     else {
59       var refineElement =
60         compareElement.substring(compareElement.indexOf('|HTTP'),
61         compareElement.length);
62       }
63     for (j = 0; j < t.length; j++) {
64       var compareString = t[j].toUpperCase();
65       if (refineElement.indexOf(compareString) != -1) {
66         findings[findings.length] = profiles[i];
67         break;
68         }
69       }
70     }
71   verifyManage(findings);
72   }
73
74 function requireAll(t) {
75   var findings = new Array();
76   for (i = 0; i < profiles.length; i++) {
77     var allConfirmation = true;
78     var allString = profiles[i].toUpperCase();
79     var refineAllString = allString.substring(0,
```

範例 1-1：nav.html 的原始程式碼（續）

```
 80        allString.indexOf('|HTTP'));
 81      for (j = 0; j < t.length; j++) {
 82        var allElement = t[j].toUpperCase();
 83        if (refineAllString.indexOf(allElement) == -1) {
 84          allConfirmation = false;
 85          continue;
 86          }
 87        }
 88      if (allConfirmation) {
 89        findings[findings.length] = profiles[i];
 90        }
 91      }
 92    verifyManage(findings);
 93    }
 94
 95  function verifyManage(resultSet) {
 96    if (resultSet.length == 0) { noMatch(); }
 97    else {
 98      copyArray = resultSet.sort();
 99      formatResults(copyArray, currentMatch, showMatches);
100      }
101    }
102
103  function noMatch() {
104    docObj.open();
105    docObj.writeln('<HTML><HEAD><TITLE>Search Results</TITLE></HEAD>' +
106      '<BODY BGCOLOR=WHITE TEXT=BLACK>' +
107      '<TABLE WIDTH=90% BORDER=0 ALIGN=CENTER><TR><TD VALIGN=TOP>' +
108      '<FONT FACE=Arial><B><DL>' +
109      '<HR NOSHADE WIDTH=100%>"' + document.forms[0].query.value +
110      '" returned no results.<HR NOSHADE WIDTH=100%>' +
111      '</TD></TR></TABLE></BODY></HTML>');
112    docObj.close();
113    document.forms[0].query.select();
114    }
115
```

```
116 function formatResults(results, reference, offset) {
117   var currentRecord = (results.length < reference + offset ?
118     results.length : reference + offset);
119   docObj.open();
120   docObj.writeln('<HTML><HEAD><TITLE>Search Results</TITLE>\n</HEAD>' +
121     '<BODY BGCOLOR=WHITE TEXT=BLACK>' +
122     '<TABLE WIDTH=90% BORDER=0 ALIGN=CENTER CELLPADDING=3><TR><TD>' +
123     '<HR NOSHADE WIDTH=100%></TD></TR><TR><TD VALIGN=TOP>' +
124     '<FONT FACE=Arial><B>Search Query: <I>' +
125     parent.frames[0].document.forms[0].query.value + '</I><BR>\n' +
126     'Search Results: <I>' + (reference + 1) + ' - ' +
127     currentRecord + ' of ' + results.length + '</I><BR><BR></FONT>' +
128     '<FONT FACE=Arial SIZE=-1><B>' +
129     '\n\n<!-- Begin result set //-->\n\n\t<DL>');
130   if (searchType == SEARCHURL) {
131     for (var i = reference; i < currentRecord; i++) {
132       var divide = results[i].split('|');
133       docObj.writeln('\t<DT>' + '<A HREF="' + divide[2] + '">' +
134         divide[2] + '</A>\t<DD><I>' + divide[1] + '</I><P>\n\n');
135       }
136     }
137   else {
138     for (var i = reference; i < currentRecord; i++) {
139       var divide = results[i].split('|');
140       docObj.writeln('\n\n\t<DT>' + '<A HREF="' + divide[2] + '">' +
141         divide[0] + '</A>' + '\t<DD>' + '<I>' + divide[1] + '</I><P>');
142       }
143     }
144   docObj.writeln('\n\t</DL>\n\n<!-- End result set //-->\n\n');
145   prevNextResults(results.length, reference, offset);
146   docObj.writeln('<HR NOSHADE WIDTH=100%>' +
147     '</TD>\n</TR>\n</TABLE>\n</BODY>\n</HTML>');
148   docObj.close();
149   document.forms[0].query.select();
150   }
151
152 function prevNextResults(ceiling, reference, offset) {
153   docObj.writeln('<CENTER><FORM>');
```

範例 1-1：nav.html 的原始程式碼（續）

```
154   if(reference > 0) {
155     docObj.writeln('<INPUT TYPE=BUTTON VALUE="Prev ' + offset +
156       ' Results" ' +
157  'onClick="parent.frames[0].formatResults(parent.frames[0].copyArray, ' +
158       (reference - offset) + ', ' + offset + ')">');
159     }
160   if(reference >= 0 && reference + offset < ceiling) {
161     var trueTop = ((ceiling - (offset + reference) < offset) ?
162       ceiling - (reference + offset) : offset);
163     var howMany = (trueTop > 1 ? "s" : "");
164     docObj.writeln('<INPUT TYPE=BUTTON VALUE="Next ' + trueTop +
165       ' Result' + howMany + '" ' +
166  'onClick="parent.frames[0].formatResults(parent.frames[0].copyArray, ' +
167       (reference + offset) + ', ' + offset + ')">');
168     }
169   docObj.writeln('</CENTER>');
170   }
171
172 //-->
173 </SCRIPT>
174 </HEAD>
175 <BODY BGCOLOR="WHITE">
176 <TABLE WIDTH="95%" BORDER="0" ALIGN="CENTER">
177 <TR>
178   <TD VALIGN=MIDDLE>
179   <FONT FACE="Arial">
180   <B>Client-Side Search Engine</B>
181   </TD>
182
183   <TD VALIGN=ABSMIDDLE>
184   <FORM NAME="search"
185     onsubmit="validate(document.forms[0].query.value); return false;">
186   <INPUT TYPE=TEXT NAME="query" SIZE="33">
187   <INPUT TYPE=HIDDEN NAME="standin" VALUE="">
188   </FORM>
```

```
189    </TD>
190
191    <TD VALIGN=ABSMIDDLE>
192    <FONT FACE="Arial">
193    <B><A HREF="main.html" TARGET="main">Help</A></B>
194    </TD>
195  </TR>
196  </TABLE>
197  </BODY>
198  </HTML>
```

程式碼可真不少。若想深度瞭解這支程式在做些什麼，除了從頭到尾研究一番，實在是別無它法了。所幸，程式碼的編排正好和執行流程的先後次序相當。

我們按下列次序逐步檢視之：

- 資料庫：records.js
- 廣域變數
- 函式
- HTML

records.js

第一個值得探究的就是 JavaScript 原始碼檔案 records.js。第 5 列的 <SCRIPT> 標籤正是其出場之處。

records.js 存有一個長度非常冗長的陣列，名為 profiles。書中篇幅有限，沒有把 records.js 的內容詳列於此，讀者如果打開 records.js 來看，也會發現文字內容似乎過於擁擠雜亂，不適合出現在印刷品中。相信讀者已從 O'Reilly 的網站把本書範例程式的壓縮檔抓了下來；解開 zip 檔之後，請以文書編輯程式開啟 ch01/records.js。記住，這就是你的資料庫了。每個元素（即每一筆紀錄）可再細分成三小段字串。下面是其中一例：

```
"http://www.serve.com/hotsyte|HotSyte-The JavaScript Resource|The " +
  "HotSyte home page featuring links, tutorials, free scripts, and more"
```

一筆紀錄以管道字元（|）分隔成三小段。這組小引擎搜尋資料庫時，若發現符合比對條件的紀錄資料進而印至視窗螢幕，那這些介乎其間的管道字元就發揮了功效。第二段字串是文件的標題（這?的標題和 HTML 語法單元的 TITLE 標籤毫無關聯）；而第三段字串則是文件說明；開頭第一段為文件的 URL。

嗯，有一件事要提出來澄清，用不用管道字元（|）來區隔紀錄的子字串，並非明文規定的律法，但是，擇取的分隔字元必須是訪客在做查詢時不太可能會輸入的字元才行，如 &^ 或 ~[% 等等。倒斜線字元（\）不要混進來用；JavaScript 看見倒斜線字元會特別解釋成跳脫字元，傳回來的查詢網頁只有訪客仰天長嘯的份，或者把應用程式搞成一團亂也未可定。

為什麼所有的資料都要擺在 JavaScript 原始碼檔案中？這?有兩個理由：調節和簡潔。假設網站有上百張網頁，可能需要寫一支伺服端程式，由其產生程式碼，把資料庫的紀錄儲存其中。客戶端搜尋引擎的做法，就是把資料庫儲存在 JavsScript 的原始碼檔案內，這樣的配置比較有組織脈絡可尋。

當然，資料庫的資料也可以放在別支應用程式?，只需把 records.js 的內容箝進去即可。順此帶上一筆，我個人很不喜歡把所有的程式碼都擺進同一個 HTML 檔案，讓 JavaScript 的程式碼跟著四處流浪見人，似乎並非明智之舉。

廣域變數

第 9 ~ 16 列的程式碼宣告了廣域變數（global variable），並指定了初值。

```
var SEARCHANY    = 1;
var SEARCHALL    = 2;
var SEARCHURL    = 4;
var searchType   = '';
var showMatches    = 10;
var currentMatch  = 0;
var copyArray   = new Array();
var docObj    = parent.frames[1].document;
```

以下闡釋各變數的用途：

JavaScript 實務：串接字串可含多項資料

此應用程式的賣點在於搜尋資訊，有點類似資料庫的功能。為了模擬搜尋資料庫的行為舉止，陣列儲存的資料必需具備同樣的型式，才能交由 JavaScript 進行分析（搜尋）。

一個陣列元素對應一項資料（如 URL、網頁標題等），似乎是天經地義的常識。這樣設計沒有錯，程式碼也可以跑，但卻可能陷自己於萬劫不復的境地。

資料字串可串接（concatenate）起來，彼此間以分隔字元（delimiter，例如管道字元 |）隔開即可。串接起來的字串仍然是字串，你可藉此大量減少位屬廣域變數範圍的陣列元素。進行資料庫分析（parse）時，String 物件的 split() 方法（method）可以替每一個元素另建一個陣列。換句話說，像這樣的廣域陣列其實不大用得到：

```
var records = new Array("The Good", "The Bad", "and The JavaScript
                        Programmer"),
```

因為，你可以在函式中用區域陣列。例如：

```
var records = "The Good|TheBad|and The JavaScript
              Programmer".split('|');
```

讀者可能會覺得這根本是換湯不換藥。其實，差別在於，廣域陣列宣告了三個廣域陣列元素，除非將其佔有的記憶體空間釋出，否則這些陣列元素是不會善罷干休的。區域陣列就不一樣了，函式只宣告一個廣域元素。這三個元素只會在搜尋期間臨時由 split('|') 方法做出來，這是因為 records 是函式的區域變數。

JavaScript 會在搜尋函式執行時宣告定義 records 變數。如此一來，記憶體空間的使用情形便較有彈性可言。串接字串內籍分隔字元而形成多項資料堆擠一格的做法，可以有效減少程式碼的長度。對我個人而言，第二種選擇才是最好的。談到程式碼進行分析作業時，我們會再度回到此一概念。

SEARCHANY

以輸入資料項中的任何一項進行搜尋。

SEARCHALL

輸入資料項必須全部作為搜尋之用。

SEARCHURL

僅搜尋 URL（以輸入資料項中的任何一項搜尋）。

searchType

指出搜尋方式（設為 SEARCHANY、SEARCHALL 或 SEARCHURL）。

showMatches

指定回應網頁每頁顯示的紀錄筆數。

currentMatch

回應網頁第一筆要列出來的紀錄為何。

copyArray

陣列的複本（copy），儲存符合比對條件的紀錄，用於顯示下一組或上一組的搜尋結果。

docObj

指涉子窗框文件物件的變數。docObj 對整套程式而言並不重要，但有助於管理程式碼，因為在列出搜尋結果時，必須多次存取 parent.frames[1].document 物件。docObj 指涉（refer to）的物件就是 parent.frames[1].document，這樣可以減少程式碼的長度，任何風吹草動，都可由 docObj 做為中心點輻射而出。

函式

接下來的討論幾個主要的函式（function）：

validate()

第 18 ~ 42 列是 validate() 函式的所在位置，當訪客按下 <Enter> 鍵時，就由 validate() 函式負責瞭解訪客的查詢需求和查詢方式。回想一下前面提到的三種查詢選擇：

- 搜尋文件標題和說明文字，只要其中一資料項吻合即可
- 搜尋文件標題和說明文字，所有資料項都必須吻合才行
- 搜尋文件的 URL（或路徑），只要其中一資料項吻合即可

validate() 接收訪客輸入的字串之後，會進一步研究一下字串的前幾個字元，以確認訪客想搜尋的東西為何以及改採取怎樣的搜尋法則。搜尋法則儲存在那？？就是 searchType 變數。如果訪客希望搜尋條件嚴格一點，輸入的資料項必須能完全比對成功才可列為搜尋結果，searchType 就設為 SEARCHALL。如果訪客想搜尋 URL，searchType 就設為 SEARCHURL。如果只想單純的看一看文件標題和說明，則將 searchType 設為 SEARCHANY（預設值）。以下是這整個過程的詳細經過：

第 19 列出現的 charAt() 是 String 物件的方法，負責搜尋字串的第一個字元，並透過 if 判斷式的比較，檢查 + 號是否存在。如果找得到 + 號，搜尋方式就設為 SEARCHALL，也就是 2。

```
if (entry.charAt(0) == "+") {
 entry = entry.substring(1,entry.length);
 searchType = SEARCHALL;
 }
```

第 23 列出現的 substring() 是 String 物件的方法，負責搜尋 " url: "。如果找到了，searchType 就據此而設定。

```
if (entry.substring(0,4) == "url:") {
 entry = entry.substring(5,entry.length);
 searchType = SEARCHURL;
 }
```

第 20 和 24 列的 substring() 在做什麼？validate() 得知搜尋標的物和搜尋方式之後，當然得將這些標示搜尋方式的指示字元（indicator，+ 與 url:）從原來的字串中拿掉抽取出來，因為指示字元並非搜尋標的物，純粹為了指定搜尋法則而生。因此，validate() 必須把這些餘贅的字元移除掉，substring() 方法的用途即在於此。

如果字串開頭找不到 + 號，也找不到 url:，validate() 就會把 searchType 設為 SEARCHANY。到此，if 判斷式算是告一段落了。緊接而來的是一小段的清除動作。由下面那兩個 while 迴圈（第 28 與 32 列）的測試條件可知，目的是把字串開頭和尾端的空白字元抽取掉。

然而，光是這樣還不夠好。程式碼會進一步檢查，整理過的輸入字串如果太短，不足 3 個字元，validate() 會將搜尋請求視為無效。搜尋的關鍵字元太短，彰顯不出搜尋結果的特點，得到的只是一堆主題不相關的紀錄罷了。但是，這種限制的標準隨人而異，讀者可按照自己的喜好做修改：

```
if (entry.length < 3) {
 alert("You cannot search strings that small. Elaborate a little.");
  document.forms[0].query.focus();
  return;
  }
```

完成上述步驟之後，validate() 就可呼叫 convertString() 函式了。此時傳遞給 convertString() 的引數，乃是經處理後（去除指示字元及空白字元）的字串物件 entry。

convertString()

convertString() 執行兩道運算。首先，接到 validate() 傳遞來的查詢字串後，convertString() 就會把這個字串切割成片，轉放到陣列的各個元素中；然後，再呼叫合宜的搜尋函式。String 物件的 split() 方法負責把訪客輸入的字串用空白格分隔開來，然後把切割的結果置入 searchArray 陣列。前述過程發生在第 45 列，如下所示：

```
var searchArray = reentry.split(" ");
```

舉個例子，如果訪客在搜尋欄位中輸入了“client-side JavaScript development”這樣的字串，searchArray 最終的內容會含有 client-side、JavaScript、development 這三個分割出來的字串元素，分別佔用了 searchArray 陣列的索引值 0、1、2。切出字串之後，convertString() 就會根據 searchType 的數值內容呼叫適合的搜尋函式。請詳參第 46 ~ 47 列的程式碼：

```
if (searchType == (SEARCHALL)) { requireAll(searchArray); }
else { allowAny(searchArray); }
```

如你所見，兩個函式定有一個會被呼叫。這兩個函式的內涵其實大同小異，但讀者需細心查訪差異點何在。下面來瞭解一下這兩個函式：allowAny() 和 requireAll()。

allowAny()

函式的名稱暗示了函式的用途（allowAny：任何一個皆可），滿足搜尋條件的最低限度，只需資料庫的紀錄吻合 searchArray 陣列儲存的其中一個字串元素即可。allowAny() 函式的程式碼示於第 50 ~ 68 列：

```
function allowAny(t) {
  var findings = new Array(0);
  for (i = 0; i < profiles.length; i++) {
    var compareElement  = profiles[i].toUpperCase();
    if(searchType == SEARCHANY) {
      var refineElement  =
        compareElement.substring(0,compareElement.indexOf('|HTTP'));
      }
    else {
      var refineElement =
        compareElement.substring(compareElement.indexOf('|HTTP'),
        compareElement.length);
      }
    for (j = 0; j < t.length; j++) {
      var compareString = t[j].toUpperCase();
      if (refineElement.indexOf(compareString) != -1) {
        findings[findings.length] = profiles[i];
        break;
        }
```

搜尋函式的重點在於利用了巢狀的 for 迴圈來比較字串。請參考「JavaScript 實務：巢狀 for 迴圈」方塊文章的說明。for 迴圈分別在第 52 列和第 63 列發揮效用。靠外的 for 迴圈負責讓 profiles 陣列的每一個元素都不被錯過；靠內的 for 迴圈則負責檢視每一個 profiles 陣列元素，企圖找出其中是否有片段的字串合於 searchArray 陣列的某個元素。

訪客輸入的字串可能夾雜了大小寫的字母，為了確保搜尋的廣度，第 53 列和第 64 列的程式碼分別宣告了區域變數 compareElement 和 compareString，把每個陣列元素的字元不論其大小寫統統轉換成大寫的字母，然後再交付查詢。如此一來，就不用擔心訪客輸入的字串是“javaScript”、“javascript”或者“jAvasCRIpt”了。

allowAny() 仍然得決定要由文件標題和說明文字來搜尋，亦或由 URL 來搜尋。第 55 列和第 59 列定義了區域變數 refineElement，refineElement 就是要和查詢字串比較的子字串，它的內容會依據 searchType 的值而有所不同。如果 searchType 為 SEARCHANY，就把紀錄的文件標題和說明文字指定給 refineElement；若非如此，則 searchType 必定為 SEARCHALL，所以 refineElement 為文件的 URL。

還記得 | 這個符號嗎？它是 JavaScript 程式用以判別紀錄的組成部份的符號。因此，若 searchType 為 SEARCHANY，substring() 方法傳回的子字串便是從 compareElement 字串的開頭字元開始抓起，直到遇見第一個“|HTTP”為止。如果 searchType 等於 SEARCHALL，substring() 方法傳回的子字串則是從 compareElement 字串中第一個“|HTTP”開始抓起，直到 compareElement 字串結束為止。前置作業都做好之後，緊接著的就是去比較訪客輸入的字串了。程式碼在第 65 列：

```
if (refineElement.indexOf(compareString) != -1) {
  findings[findings.length] = profiles[i];
  break;
  }
```

如果從 refineElement 字串中找得到 compareString，就等於得到一筆吻合比對條件的紀錄。進行比較的是切割處理過的字串，而我們要留下的則是完整的紀錄，因此，第 66 列的程式碼是把原來的紀錄資料，也就是 profiles 陣列的元素指定給 findings 陣列。我們以 findings.length 做為 findings 陣列的索引計值器（indexer），如此可連續不斷指定元素給 findings 陣列；findings 陣列宣告於第 52 列，宣告時的 findings.length 為 0。

一旦紀錄符合比對條件，自然沒有理由再拿這筆紀錄和其它的查詢子字串比較了。此時，第 67 列的 break 敘述就會中止 for 迴圈的比較敘述。這樣的考量並非絕對必須，但可以減少不必要的運算。

全部的紀錄都經過查詢字串的比對之後，allowAny() 會把儲存在 findings 陣列的的紀錄傳遞給 verifyManage() 函式，程式碼出現在第 95 ~ 101 列。如果搜尋成功(resultSet.length 大於 0)，就會呼叫 formatResults() 函式，把搜尋結果顯示出來。

反之，如果搜尋沒有成功（resultSet.length 等於 0），則有 noMatch() 函式負責提示訪客，說明訪客輸入的查詢字串和搜尋法則無法在資料庫中找到符合條件的紀錄。formatResults() 函式和 noMatch() 函式後續會討論到。接著，讓我們往下一個搜尋函式前進。

requireAll()

搜尋資料項前面若綴上 + 號，進行搜尋時，就會改為呼叫函式 requireAll()。requireAll() 和 allowAny() 幾乎沒什麼兩樣，差別僅在於呼叫 requireAll() 函式之後，搜尋條件變得更嚴苛，必須搜尋資料項的每一項都符合的紀錄才能納入輸出結果之列。對 allowAny() 而言，紀錄的文字資料中，搜尋資料項只要配對出一只，便認定這筆紀錄應囊括在訪客的搜查範圍中，而予以輸出列示。然而，倘若由 requireAll() 作主，則比對過程中，勢必難逃等待的命運，直到所有搜尋資料項都出現在某一筆紀錄資料時，才能把這筆紀錄加到輸出結果之列。第 74 列是 requireAll() 函式定義的開頭：

```
function requireAll(t) {
  var findings = new Array();
  for (i = 0; i < profiles.length; i++) {
    var allConfirmation = true;
    var allString = profiles[i].toUpperCase();
    var refineAllString = allString.substring(0,
      allString.indexOf('|HTTP'));
    for (j = 0; j < t.length; j++) {
      var allElement = t[j].toUpperCase();
      if (refineAllString.indexOf(allElement) == -1) {
        allConfirmation = false;
        continue;
        }
      }
    if (allConfirmation) {
      findings[findings.length] = profiles[i];
      }
    }
  verifyManage(findings);
  }
```

乍看之下，和 allowAny() 頗為相似。有巢狀的 for 迴圈，也有大寫字母的轉換敘述，統統都在那兒。然而，第 79 ~ 80 列的程式碼就起了變化：

```
var refineAllString = allString.substring(0,allString.indexOf('|HTTP'));
```

讀者可能已經發現，requireAll() 並沒有再針對 searchType 的設定值做檢測，進而找出究竟是紀錄的那些部份要拿出來做搜尋比對之用。allowAny() 在第 50 列的程式碼正在做出決策，但對 requireAll() 而言根本多此一舉，故而略去。這是因為我們在第 46 列的程式碼就已經限定只有在 searchType 為 SEARCHALL 之時，才有資格呼叫 requireAll() 函式進行搜尋之舉。URL 搜尋法並不含布林 AND 法則，因此，每筆紀錄拿出來在此比對的資料將會是文件標題和說明文字。

requireAll() 稍微難了點，不容易看出門道。由於訪客輸入的所有搜尋資料項都必須出現在紀錄拿出來比對的字串中，因此搜尋過程的邏輯演算會比 allowAny() 的要嚴密許多。請參見第 83 ~ 86 列的程式碼：

```
if (refineAllString.indexOf(allElement) == -1) {
  allConfirmation = false;
  continue;
  }
```

refineAllString 正是每筆紀錄交付搜尋的子字串，而 allElement 列於內圈 for 迴圈的主體敘述中，和 refineAllString 比較之前先全數換裝大寫字母，再交由 if 判斷式去研判，一旦發現 allElement 不在 refineAllString 中，即刻研判條件不符，把 allConfirmation 設為 false，再以跳躍敘述 continue 通知外圈的 for 迴圈直接放棄這一筆紀錄，繼續往下一筆紀錄搜尋。下這樣的決斷，總比等到全部的搜尋資料項比對完了之後再宣判結果來得有效率許多。

當內圈的 for 迴圈走完一圈之後，等於所有的搜尋資料項都和這一筆紀錄比較過了，如果此時的區域變數 allConfirmation 仍然是當初的 true，那麼比對就成功了，咱們的客戶端搜尋引擎替訪客找到了一筆符合其要求的紀錄。隨後這筆比對成功的紀錄會加到 findings 陣列中，如第 89 列程式碼所示。這樣的搜尋條件顯然嚴苛不少，但可讓搜尋結果更能取信於訪客。

當資料庫的每筆紀錄統統受過 requireAll() 函式這般的嚴刑酷打之後，findings 陣列就會把逼供過程中招供出來的紀錄連同自身轉交給 verifyManage()，由 verifyManage() 進一步決定最終的輸出畫面。如果的確找到了條件吻合的紀錄，便輪到 formatResults() 上場；否則，verifyManage() 會改呼叫 noMatch()，通知訪客這令人惋惜的消息。

JavaScript 實務：巢狀 for 迴圈

allowAny() 和 requireAll() 這兩個搜尋函式都用到了巢狀的 for 迴圈。一維陣列對一層 for 迴圈，多維陣列對多層 for 迴圈，各階元素才能輪迭替換，不致走漏。就技術層面而言，JavaScript 的 Array 物件其實是一維陣列，然而，JavaScript 可藉由一維陣列的深層發展，而模擬出多維陣列。請先看下列的一維陣列，內含 5 個元素：

```
var numbers = ("one", "two", "three", "four", "five");
```

如果你想拿一個字串和陣列中的字串元素相比對，採用單層 for 迴圈（或 while 迴圈）就已足夠。範例如下所示：

```
for (var i = 0; i < numbers.length; i++) {
  if (myString == numbers[i]) { alert("That's the number");
  break;
  }
```

沒什麼特別的，就這麼簡單。說到多維陣列，只要想到陣列的陣列就行了，陣列的元素可以是另一個陣列。例如：

```
var numbers = new Array(
new Array("one", "two", "three", "four", "five"),
new Array("uno", "dos", "tres", "cuatro", "cinco"),
new Array("won", "too", "tree", "for", "fife")
);
```

單層的 for 迴圈是沒有辦法劃分出各個元素的。我們還需要更細的劃分。剛剛的 numbers 陣列是一維陣列（1×5）。下面的 numbers 陣列搖身一變成了多維陣列（3×5）。為了在這 15 個元素間走透透，則意謂著還需要另一層 for 迴圈隨伺在旁：

```
for (var i = 0; i < numbers.length; i++) {         // 1...
  for (var j = 0; j < numbers[i].length; j++) {    // and 2.
    if (myString == numbers[i][j]) {
      alert("Finally found it.");
```

```
        break;
        }
      }
    }
```

這就是取得每個二維陣列元素的做法了。讀者需做第三項修鍊，才能化於涅盤之境。現在，我們打算秀出總共 216 色的安全色彩（safe color），列成表格式的調色盤，每一格都是一小塊的方塊色。此時的 for 巢狀迴圈便派上用場了。不過，這一次我們將以一維陣列來應對。

安全色彩以十六進位制的數字表示，共有 6 組數字，每 2 組組成色彩的基色，各代表 RGB 這 3 種基色，如 FFFFFF、336699、99AACC。安全色彩賴以成色的數對為：33、66、99、AA、CC 和 FF。下面就是這個一維的數對陣列：

```
var hexPairs = new Array("33","66","99","AA","CC","FF");
```

或曰：「怎麼只有一個一維陣列？騙錢書，還錢來！」

先別急著跑到書店退書。實際而論，這張未露面的調色盤是三維空間的，只不過每維空間的元素都以同一個一維陣列巢狀發展下去罷了。下面是可行的程式碼：

```
var str = ' ';

// Strike up a table
document.writeln('<H2>Web Safe Colors</H2>' +
  '<TABLE BORDER=1 CELLSPACING=0>');
for (var i = 0; i < hexPairs.length; i++) {
  // Create a row
  document.writeln('<TR>');
  for (var j = 0; j < hexPairs.length; j++) {
    for (var k = 0; k < hexPairs.length; k++) {
    // Create a string of data cells for the row with whitespace in each
    // Notice each background color is made with three hexPairs elements
    str += '<TD BGCOLOR="' + hexPairs[i]  + hexPairs[j] + hexPairs[k] +
    '">   </TD>';
```

```
        }
      // Write the row of data cells and reset str
      document.writeln(str);
      str ='';
      }
    // End the row
    document.writeln('</TR>');
    }
  // End the table
  document.writeln('</TABLE>');
```

請把這段程式碼帶到網站文件中（\ch01\websafe.html 有現成的調色盤），執行之後，讀者可得到一張 6 ×36 的色彩表，總共有 216 色的小色塊（6 ×6 ×6），這就是所謂的 216 色安全色彩。程式碼用到了三層 for 迴圈，建構出三維空間陣列。當然，你可以對此調色盤做任何的組合和修改；然而，醉翁之意不在酒，讓你懂得巢狀 for 迴圈的妙用，才是此方塊文章的最終目的。

verifyManage()

讀者也許已經領悟到 verifyManage() 函式的用途了。verifyManage() 負責確認訪客的搜尋是否有任何符合其條件的紀錄存在，並根據有無紀錄的條件，來呼叫輸出函式把結果列出來。函式的首列定義始於第 95 列：

```
function verifyManage(resultSet) {
if (resultSet.length == 0) {
     noMatch();
      return;
      }
    copyArray = resultSet.sort();
    formatResults(copyArray, currentMatch, showMatches);
    }
```

allowAny() 和 requireAll() 做完比對工作之後，都會呼叫 verifymanage()，並把 findings 陣列當成引數傳遞給 verifyManage()。第 96 列的程式碼指出，如果陣列 resultSet（findings 陣列的複本）不含有任何一筆紀錄，則呼叫 noMatch() 函式。

只要 resultSet 陣列中含有符合搜尋條件的紀錄，那怕只有一筆，整個 resultSet 陣列就會交由 sort() 方法排序，指定給廣域變數 copyArray。排序並非必然之舉，但這樣做的好處，的確增進了輸出結果的次序感，當添增 profiles 陣列元素時，也就不用擔心紀錄的前後順序，可盡情的添加紀錄了。因為我們在最後輸出的關頭，先做了排序的整理。

說到這兒，為什麼已經有了 findings 陣列了，還需要再做一個 copyArray 陣列？記住一點，findings 是區域變數，因此，findings 是一個暫時的陣列。一旦搜尋結束（亦即應用程式執行了其中一個搜尋函式之後），findings 就宣告死亡，而其所配置之記憶體也隨之消散，另供它用。這樣的設計模式很好。然而，我們仍然得繼續存取這些搜尋出來的紀錄，因此，有了 resultSet 複本，排序後，再交給 copyArray 陣列。

由於之前的程式碼設定每頁顯示 10 筆紀錄，因此，訪客眼前看到的結果很有可能只是符合條件的紀錄筆數中的一小部份。copyArray 是廣域陣列變數；陣列 resultSet 排序之後再指定給 copyArray，copyArray 自然留有所有吻合的紀錄筆數。訪客得以一次觀看 10 筆、15 筆紀錄，或者更多筆，全在於應用程式的設定。這個 copyArray 會一直留有全部搜尋出來的紀錄，直到訪客下達另一次的查詢為止。

函式 verifyManage() 最後做的事情是呼叫 formatResults()。呼叫時，陣列 copyArray 與變數 currentMatch、showMatches 會傳遞給 formatResults()。currentMatch 指出要從哪一筆紀錄開始列出，showMatches 則指出每頁列出多少筆紀錄。currentMatch 和 showMatches 兩者都是廣域變數，函式執行完之後，currentMatch 和 showMatches 仍然會存留於記憶體中，這兩個變數和應用程式的生命周期相當，其它地方仍然用得到。

noMatch()

顧其名，思其義，noMatch() 就是“No Match”。如果查詢沒有得出任何結果，這個函式就負責帶給訪客這道壞消息。noMatch() 的程式碼很短很可愛，編製出一頁通知訪客查詢沒有結果的網頁。程式碼始於第 103 列：

```
function noMatch() {
  docObj.open();
  docObj.writeln('<HTML><HEAD><TITLE>Search Results</TITLE></HEAD>' +
    '<BODY BGCOLOR=WHITE TEXT=BLACK>' +
    '<TABLE WIDTH=90% BORDER=0 ALIGN=CENTER><TR><TD VALIGN=TOP>' +
    '<FONT FACE=Arial><B><DL>' +
    '<HR NOSHADE WIDTH=100%>"' + document.forms[0].query.value +
    '" returned no results.<HR NOSHADE WIDTH=100%>' +
    '</TD></TR></TABLE></BODY></HTML>');
  docObj.close();
  document.forms[0].query.select();
  }
```

formatResults()

這個函式的工作，是顯示符合搜尋條件的紀錄資料給訪客瀏覽。函式的寫法不難，只不過讀者需要堅強的 JavaScript 基礎才能看得出門道。以下是秀出漂亮的回應網頁的要素：

- **標頭、標題和主體程式碼**
- **文件標題、說明文字和連結之 URL**
- **Previous 和 Next 按鈕，用於觀看前後之紀錄**

HTML 標頭和標題

第 116 ~ 129 列的程式碼負責把標頭（head）、標題（title）和主體內容的起始部份印出來。如下：

```
function formatResults(results, reference, offset) {
  var currentRecord = (results.length < reference + offset ?
    results.length : reference + offset);
  docObj.open();
  docObj.writeln('<HTML><HEAD><TITLE>Search Results</TITLE>\n</HEAD>' +
    '<BODY BGCOLOR=WHITE TEXT=BLACK>' +
    '<TABLE WIDTH=90% BORDER=0 ALIGN=CENTER CELLPADDING=3><TR><TD>' +
    '<HR NOSHADE WIDTH=100%></TD></TR><TR><TD VALIGN=TOP>' +
```

```
'<FONT FACE=Arial><B>Search Query: <I>' +
parent.frames[0].document.forms[0].query.value + '</I><BR>\n' +
'Search Results: <I>' + (reference + 1) + ' - ' + currentRecord +
' of ' + results.length + '</I><BR><BR></FONT>' +
'<FONT FACE=Arial SIZE=-1><B>' +
'\n\n<!- Begin result set //-->\n\n\t<DL>');
```

印出標頭和標題之前，讓我們先找出要從那一筆紀錄開始印起。由 verifymanage() 可知，這一筆起頭的紀錄應由 results[reference] 決定。總共應該印出 offset 筆紀錄，除非 reference + offset 已經超出紀錄的總筆數。為了確定 reference 和 offset 之和是否超出 results.length，我們以條件運算式測試之，以找出真正應列出的紀錄筆數。第 117 列的 currentRecord 變數即表示此一列出筆數值；currentRecord 馬上就會用到。

現在，formatResults() 印出了 HTML 文件的標頭和標題了。主體部份以一個置中的表格和一條水平線起頭。在第 125 列，我們又把訪客輸入的查詢字串秀出來，提示給訪客參考之用。查詢字串的引用取自表單欄位的內容：

```
parent.frames[0].document.forms[0].query.value
```

到第 126 列，事態更加的明朗。由這?印出來的資訊，指出搜尋的總筆數，並點明當前印製出來的紀錄範圍為何，例如：

```
Search Results:  1 - 10 of 38
```

我們需要三個數字才能反應現況：第一筆要印出來的紀錄、要印出來的紀錄筆數、copyArray 陣列的長度。我們一步一步來看這幾個數字。記住一點，這幾個數字並非程式用於顯示紀錄的邏輯運算。這?的邏輯概念無非是想讓訪客知道搜尋出來的紀錄筆數有多少，以及目前是從那一筆紀錄開始印起的。以下是事情的經過：

1. 把 currentMatch 指定給 reference，也就是當前紀錄的編號。
2. 把 showMatches 指定給 offset，指出每頁要印出多少筆紀錄（此例為 10 筆）。
3. 如果 reference + offset 大於紀錄總筆數，則改為指定總筆數。否則的話，就指定 reference + offset 的和（究竟該用怎樣的數字已經反應在 currentRecord 了）。
4. 印出比對符合的總筆數。

步驟 1 和步驟 2 相當簡單。回想一下 verifyManage() 的程式碼，第 99 列指出：

```
formatResult(copyArray, currentMatch, showMatches);
```

區域變數 results 是 copyArray 的複本。變數 reference 是 currentMatch 之值，因此，reference + offset 的和就是 currentMatch + showMatch 的和。第 13 ~ 14 列的程式碼指出，showMatches 設為 10，而 currentMatch 設為 0。因此，reference 自當從 0 開始，而 reference + offset 之和也恰巧從 10 起跑。步驟 1 和步驟 2 是函式定義之初即完成的運算。

到了步驟 3，我們以條件運算式（第 117 ~ 118 列）確定 reference + offset 是否大於紀錄的總筆數。易言之，offset 加到 reference 之後，兩者之和會大於紀錄總筆數嗎？如果 reference 是 20，總紀錄筆數是 38，reference 加上 offset 之後得到 30。則顯示出來的文字列應如下所示：

```
Search Results:  20 - 30 of 38
```

然而，如果 reference 是 30，總筆數是 38，reference 加上 offset 之後，我們得到 40。則顯示出來的文字列應如下所示：

```
Search Results:  30 - 40 of 38
```

這是不對的。搜尋引擎無法印出第 39 筆和第 40 筆的紀錄，最多只能找到第 38 筆紀錄而已。這代表紀錄筆數已經印到尾端了，因此，應該要印出來的是總筆數而非 reference + offset 之和。最終結果應如下所示：

```
Search Results:  30 - 38 of 38
```

函式 formatResults() 隨處可見 \n 和 \t 這一類的特殊字元。\n 是新列字元，就好像你用文書編輯程式寫程式碼時，按下 <Enter> 鍵的效果，而 \t 同等於按下跳格鍵。這些特殊字元是為了讓程式碼做出來的 HTML 回應網頁，外觀看起來整齊一點。我特地把特殊字元加進程式碼，讓你體驗一下不同的視覺感受。這些特殊字元並非必要，然而，也不致於影響程式碼的書寫，這一點請讀者謹記在心。如果覺得這些特殊字元讓你有如坐針氈之感，就不要用了。本書用到特殊字元的地方著實有限。

秀出文件標題、說明文字以及連結之 URL

現在，該印那幾筆紀錄已經決定出來了，接著就是實際製成網頁了。第 130 ~ 143 列的程式碼如下：

```
if (searchType == SEARCHURL) {
  for (var i = reference; i < currentRecord; i++) {
    var divide = results[i].split('|');
    docObj.writeln('\t<DT>' + '<A HREF="' + divide[2] + '">' +
      divide[2] + '</A>' + '\t<DD>' + '<I>' + divide[1] + '</I><P>\n\n');
    }
  }
else {
  for (var i = reference; i < currentRecord; i++) {
    var divide = results[i].split('|');
    docObj.writeln('\n\n\t<DT>' + '<A HREF="' + divide[2] + '">' +
      divide[0] + '</A>' + '\t<DD>' + '<I>' + divide[1] + '</I><P>');
    }
  }
```

第 131 列和 138 列分別是 for 迴圈，兩者都以 currentRecord 做為計數用的參數，執行相同的運算，差別僅在於列出的項目次序不同而已。searchType 再度出現。如果 searchType 為 SEARCHURL，則 URL 會被視為連結文字列出來。除了 SEARCHURL 之外，searchType 就只有 SEARCHANY 或 SEARCHALL 的可能了；不論何者，文件標題都會被視為連結文字而列出來。

搜尋法則已經知道了，問題是該如何顯示紀錄資料才能讓畫面看起來更好看？我們只需要一個迴圈，把紀錄的標題、說明文件和 URL 分隔開來，以我們最期待的方式置放到網頁上去就行了。以下是 for 迴圈的開頭：

```
for (var i = reference; i < lastRecord; i++) {
```

再來請看紀錄的部份。回想 records.js 檔案。profiles 的每一個元素都是一串字串，由 | 分隔出子字串。以下是我們把子字串抽離出來的做法：

```
var divide = results[i].split('|');
```

對每一元素而言，區域變數 divide 就設為一個陣列，內含由 | 分隔出來的元素（子字串）。第一個元素 divide[0] 是 URL，第二個元素 divide[1] 是文件標題，而第三個元素 divide[2] 則為文件的說明文字。每一個元素都配有 HTML 的標籤（範例中用的是 <DL>、<DT>、<DD>），印製成 HTML 的網頁送回給訪客。如果訪客以 URL 來搜尋，URL 就會顯示成連結文字，如果不是 URL 來搜尋，則文件標題會成為連結文字。

加上 Previous 和 Next 按鈕

剩下來的事是添加按鈕，這樣訪客才能前後換頁觀看搜尋出來的紀錄資料。前後換頁的功能寫在 prevNextResults() 函式，下一個討論的函式就是 prevNextResults()。下面是 formatResults() 函式最後的幾列程式碼：

```
docObj.writeln('\n\t</DL>\n\n<!- End result set //-->\n\n');
prevNextResults(results.length, reference, offset);
docObj.writeln('<HR NOSHADE WIDTH=100%>' +
  '</TD>\n</TR>\n</TABLE>\n</BODY>\n</HTML>');
docObj.close();
}
```

這一小部份的程式碼用來呼叫 prevNextResults()，加上最後的 HTML。

prevNextResults()

如果讀者讀到現在一句怨言也沒有，想必這個函式也難不倒你了。prevNextResults() 如下所示，從第 152 列開始：

```
function prevNextResults(ceiling, reference, offset) {
  docObj.writeln('<CENTER><FORM>');
  if(reference > 0) {
    docObj.writeln('<INPUT TYPE=BUTTON VALUE="Prev ' + offset +
      ' Results" ' +
      'onClick="parent.frames[0].formatResults(parent.frames[0].copyArray, ' +
      (reference - offset) + ', ' + offset + ')">');
    }
  if(reference >= 0 && reference + offset < ceiling) {
```

```
    var trueTop = ((ceiling - (offset + reference) < offset) ?
      ceiling - (reference + offset) : offset);
    var howMany = (trueTop > 1 ? "s" : "");
    docObj.writeln('<INPUT TYPE=BUTTON VALUE="Next ' + trueTop +
      ' Result' + howMany  + '" onClick="' +
      parent.frames[0].formatResults(parent.frames[0].copyArray, ' +
      (reference + offset) + ', ' + offset + ')">');
    }
  docObj.writeln('</CENTER>');
  }
```

這個函式印了一個置中的 HTML 表單和按鈕，放在回應網頁的底部。圖 1-3 是一頁回應網頁，底端有 Prev 和 Next 這兩個按鈕。按鈕的組合有下列三種可能性：

- 只有 Next 按鈕 —— 對第一頁回應網頁而言，此時並無前頁。
- Prev 和 Next 按鈕皆有 —— 界於第一頁回應網頁和最後一頁回應網頁之間的網頁，就應該有這兩個按鈕。此時，既可看前頁，也能看後頁。
- 只有 Prev 按鈕 —— 對最後一頁網頁而言，再也沒有後頁了。

三種組合，兩個按鈕。這意謂著應用程式必須知道何時該印出那一種組合。下列詳述每一種組合在那一種情況下會發生。

只有 Next 按鈕

什麼時候只能放置 Next 按鈕？答案：除了最後一頁之外，其餘的頁數都要有 Next 按鈕。換句話說，只要 reference + offset 小於總紀錄筆數時，就要配上 Next 按鈕。

那麼，何時該把 Prev 按鈕排除在外？答案：在第一頁的時候。易言之，當 reference 等於 0 時，就要拿掉 Prev 按鈕。

Prev 和 Next 按鈕皆有

什麼時候應該兩個按鈕都放？ Next 按鈕應該放在任何的網頁中，除了最後一頁網頁。而 Prev 按鈕也應該放在任何的網頁中，除了第一頁網頁。因此，當 reference 不再等於 0 時，就需要 Prev 按鈕，而當 reference + offset 仍然小於總紀錄筆數時，就應當設有 Next 按鈕。

只有 Prev 按鈕

讀者是否已知道何時可以引入 Prev 按鈕，而在什麼條件下又應當把 Next 按鈕排除在外？答案：在最後一頁時。換句話說，當 reference + offset 大於或等於總紀錄筆數時，就只能有 Prev 按鈕現身而已。

講了這麼多，讀者也許還有斷簡殘篇的感覺，但至少我們知道何時該配那一種按鈕組合了。第 154 列和第 160 列的 if 敘述，就是負責判斷按鈕組合的地方。這些敘述會根據當前的瀏覽情形，來搭配 Prev 按鈕和 Next 按鈕。

JavaScript 實務：善用 document.write()

再來看一看 formatResults()。讀者大概會發現寫到網頁的 HTML 都是以 document.write() 或 document.writeln() 書寫的。傳遞給這些方法的字串通常都很長，得要分成好幾列由 + 號串接起來才行。讀者可能覺得如果一列寫不完，乾脆就用 document.writeln() 一列一列寫，這樣一來，程式碼也會比較好讀。我不這麼做是有原因的，這?要說個明白。以下是 formatResults() 的程式片段：

```
function formatResults(results, reference, offset) {
    docObj.open();
    docObj.writeln('<HTML>\n<HEAD>\n<TITLE>Search Results</TITLE>\n
      </HEAD>' +
    '<BODY BGCOLOR=WHITE TEXT=BLACK>' +
    '<TABLE WIDTH=90% BORDER=0 ALIGN=CENTER CELLPADDING=3><TR><TD>' +
    '<HR NOSHADE WIDTH=100%></TD></TR><TR><TD VALIGN=TOP>' +
    '<FONT FACE=Arial><B>Search Query: <I>' +
    parent.frames[0].document.forms[0].query.value + '</I><BR>\n' +
    'Search Results: <I>' + (reference + 1) + ' - ' +
    (reference + offset > results.length ? results.length :
      reference + offset) +
    ' of ' + results.length + '</I><BR><BR></FONT>' +
    '<FONT FACE=Arial SIZE=-1><B>' +
    '\n\n<!- Begin result set //-->\n\n\t<DL>');
```

這?只呼叫了一個 document.writeln() 方法，負責把文字寫到網頁?去。看來很不理想。下列的方式可能會比較整齊一點，每一列各寫一道 document.writeln()：

```
function formatResults(results, reference, offset) {
  docObj.open();
  docObj.writeln('<HTML><HEAD><TITLE>Search Results</TITLE>\n</HEAD>');
  docObj.writeln('<BODY BGCOLOR=WHITE TEXT=BLACK>');
  docObj.writeln('<TABLE WIDTH=90% BORDER=0 ALIGN=CENTER ' +
    'CELLPADDING=3><TR><TD>');
  docObj.writeln('<HR NOSHADE WIDTH=100%></TD></TR><TR><TD VALIGN=TOP ');
  docObj.writeln('<FONT FACE=Arial><B>' +  'Search Query: <I>' +
    parent.frames[0].document.forms[0].query.value + '</I><BR>\n');
  docObj.writeln('Search Results: <I>' + (reference + 1) + ' - ' );
  docObj.writeln( (reference + offset > results.length ?
    results.length : reference + offset) +
    ' of ' + results.length + '</I><BR><BR></FONT>' +
    '<FONT FACE=Arial SIZE=-1><B>');
  docObj.writeln('\n\n<!- Begin result set //-->\n\n\t<DL>');
```

這樣寫看起來可能比較有組織，但是，每呼叫一次 document.writeln() 就意謂著 JavaScript 搜尋引擎的工作量愈多。仔細想一想。你會分五次來來回回的去超商買東西？或者一次就把東西買齊了呢？文字字串長一點沒關係，用 + 串接起來就行了，這樣做沒什麼不好的。

當訪客按下按鈕時，不論按下的是 Prev 按鈕或是 Next 按鈕，按鈕都會呼叫 formatResults() 函式。唯一的差別在於按鈕傳遞的引數，分別指向紀錄資料中不同的段落。兩個按鈕很像，但看來不同，因為 VALUE 屬性不同。第 155 ~ 156 列是 Prev 按鈕的開端：

```
docObj.writeln('<INPUT TYPE=BUTTON VALUE="Prev ' + offset + ' Results" ' +
```

第 164 ~ 165 列是 Next 按鈕的開端：

```
docObj.writeln('<INPUT TYPE=BUTTON VALUE="Next ' + trueTop +
  ' Result' + howMany + '" ' +
```

兩者都含有表單按鈕的 TYPE 屬性和 VALUE 屬性，外加一個數目字，指出前頁筆數或後頁筆數。由於前頁筆數都一樣（offset），Prev 按鈕顯示出來的數字都是“Prev 10 Results”。然而，後頁筆數的數字就有可能變化了。後頁筆數正常是 offset，但如果最後一頁筆數少於 offset，則應以剩餘的筆數為顯示之數字。因此，Prev 按鈕的數字就不再是 offset，而以 trueTop 替換。

注意到一點，Prev 按鈕總是含有“Results”這個字。這很有道理。showMatches 從都到尾都沒有改過，一直都是 10。訪客總是可以看到 10 筆前頁的紀錄。然而，對 Next 按鈕來講，事情就不會那麼順利了。假設最後一頁只有一筆紀錄。訪客會看到按鈕出現“Next 1 Results”這樣的句子，顯然不合語法要求。單數“1”和複數“Results”是不能擺在一起的【譯註】。為了解決這樣的問題，prevNextResults() 內含了一個區域變數，名為 howMany，用於條件運算式中。請看第 163 列的程式碼：

```
var howMany = (trueTop > 1 ? "s" : "");
```

如果 trueTop 大於 1，howMany 就設為字串 s。如果 trueTop 等於 1，howMany 就設為空字串。如讀者在第 165 列所見，howMany 會在“Result”之後接著印出來。如果網頁中只剩一筆紀錄，“Result”就會出現，不會有任何改變。然而，如果筆數不只一筆，訪客會看見“Results”。

最後一件工作是告訴按鈕，當它們被按下時，該做些什麼事。稍早我曾提過兩個按鈕的 onClick 事件都會呼叫 formatResults() 函式。第 157 ~ 158 列和第 166 ~ 167 列的程式碼會自動寫出 formatResults() 函式的呼叫程序，加進 onClick 事件處理器（event handler）中。以下是第一組（document.writeln() 的後半部）：

```
'onClick="' + parent.frames[0].formatResults(parent.frames[0].copyArray, ' +
 (reference - offset) + ', ' + offset + ')">');
```

注意到傳遞的引數有三個，分別是 copyArray、reference - offset 和 offset。Prev 按鈕會取得這三個引數。此外，注意 formatResults() 和 copyArray 的寫法：

```
parent.frames[0].formatResults(...);
```

譯註　當然，中文就沒有這個問題了！

以及：

```
parent.frames[0].copyArray
```

乍看之下很奇怪，但請記住，此時 formatResults() 的呼叫並非來自於 nav.html (parent.frames[0])。formatResults() 的呼叫是從回應網頁 parent.frames[1] 而來的，在此窗框中並沒有 formatResults() 函式，也沒有 copyArray 陣列。因此，parent.frames[1] 若要呼叫屬於 parent.frames[0] 的 formatResults() 函式和 copyArray 陣列，就需要綴上 parent.frames[0]，以明確指涉其來源地。

Next 按鈕的 onClick 事件處理器呼叫類似 Prev 按鈕的。等一下；難道這？不用面對最後剩餘筆數小於 offset 的可能性嗎？看來不用。因為這部份已交由 formatResults() 來做。我們要做的就是把 reference 加到 offset，然後傳送給 formatResults()。請看一下第 166 ~ 167 列的程式碼，document.writeln() 方法的後半部如下：

```
'onClick="parent.frames[0].formatResults(parent.frames[0].copyArray, ' +
  (reference + offset) + ', ' + offset + ')">');
```

JavaScript 實務：條件運算式

隨著程式碼的推演，讀者想必已發現條件運算式的優點了。條件運算式需要三個運算元，客戶端搜尋引擎善用了條件運算式的簡潔優點，等於是寫在一列的 if/else 敘述。網景提供的 JavaScript 參考手冊指出了條件運算式的語法：

```
(condition) ? val1 : val2
```

若 condition 為真，則條件運算子（即三元運算子 ?）取值為 val1；若 condition 為假，則條件運算式取值為 val2。條件運算式很方便，不僅程式寫來輕鬆，而且程式碼也比較短。如果你有連續的巢狀敘述，用條件運算式來寫會更方便。

然而，條件運算式絕非萬靈丹。如果 condition 為真或為假時，會有許多事情要接連發生，就應該採用 if/else。除此而外，不妨試它一試。

HTML

nav.html 只有少數靜態（static）的 HTML 語言。從第 174 列開始就是了：

```
</HEAD>
<BODY BGCOLOR="WHITE">
<TABLE WIDTH="95%" BORDER="0" ALIGN="CENTER">
<TR>
  <TD VALIGN=MIDDLE>
  <FONT FACE="Arial">
  <B>Client-Side Search Engine</B>
  </TD>

  <TD VALIGN=ABSMIDDLE>
  <FORM NAME="search"
    onsubmit="validate(document.forms[0].query.value); return false;">
  <INPUT TYPE=TEXT NAME="query" SIZE="33">
  <INPUT TYPE=HIDDEN NAME="standin" VALUE="">
  </FORM>
  </TD>

  <TD VALIGN=ABSMIDDLE>
  <FONT FACE="Arial">
  <B><A HREF="main.html" TARGET="main">Help</A></B>
  </TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

沒什麼特殊之處，表格中嵌進了表單，交付表單後，就會執行前面討論過的程式碼。讀者唯一可能有的問題是：「表單沒有按鈕要怎麼交付？」就 HTML 2.0 以後的規範而言，即使只有單一文字欄位的表單，也能夠交付瀏覽器執行（包括 NN 和 MSIE）。

這世界沒有明文規定說一定要這樣做。如果加上按鈕或影像圖案較得你心，就這樣做吧。

建立個人專屬的資料庫

到頭來，你總會把本書提供的紀錄換成別的資料。請以下列三道簡單的步驟來做：

1. 以文書編輯程式打開 records.js。
2. 把 profiles 陣列中儲存的紀錄移除，讓整個檔案只剩下空的陣列定義：

```
var profiles = new Array(


        );
```

3. 請以下列的語法添加你自己的紀錄：

```
"Your_Page_Title|Your_Page_Description|
          http://your_page_url/file_name.html",
```

請在小括號之間填入任意筆數的紀錄。每筆紀錄的尾端請記得加上逗號，但最後一筆除外。另外，注意到文件標題、說明文字和 URL 都是以 | 來分隔的，因此，文件標題、說明文字或 URL 的組成字元中不能含有 | 字元，才不會造成 JavaScript 的錯亂。最後記住，如果想在字串中添加雙引號（"），請務必先以倒斜線字元跳脫方可（\"）；因為字串前後兩端已有了配對的雙引號，前後雙引號間若再有雙引號出現，非跳脫不可。

成品加工

這樣的搜尋引擎當然能發揮功效了。更好的是你能針對自身需求，做進階的改進。以下列舉了幾種未來之路：

- 相容於 JavaScript 1.0
- 讓程式更不容易出錯
- 刊登廣告看板
- 新增搜尋功能
- 研發字群集

相容於 JavaScript 1.0

你知，我也知；NN 和 MSIE 都已出到 4.x ~ 5.x 版了。兩套瀏覽器都能免費取得，但這世上仍然有人停留在 MSIE 3.02 或 NN 2.x 的版本。昇級並非難事，只是為了讓瀏覽器昇級，結果連作業系統也要一併昇級，顯然不符成本效益，無怪乎許多人仍然停留在舊版的水準。

搜尋引擎可以說是網站的核心功能，也許你想擴充這支搜尋程式，把 JavaScript 1.0 的規範容納進來。所幸，讀者唯一要做的，就是回到前面所列的程式碼，一列一列的研讀，找出有那些程式碼用到的特點是 JavaScript 1.0 所沒有的，接著再逐一修正即可。

其實，我已經替你做了，但別忘了，我希望你自己去嘗試一下。實際上，你可以在 /ch01/js1.0 找到修正過的版本。請以瀏覽器打開 index.html 重新檢視一番。這一小節我們很快的看一遍，究竟程式要做什麼修改才能相容於 JavaScript 1.0。下面指出三項修正：

- 沒有 JavaScript 的原始碼檔案（其實是瀏覽器的問題）
- 沒有陣列排序（沒有 sort() 方法）
- 避開 split() 方法

NN 2.x 和 MSIE 3.x 並沒有支援 .js 的原始碼檔案【註】。為了避開這個問題，只好把 profiles 陣列加到 nav.html 檔案中。第二個改變是消掉第 90 列的 resultSet.sort() 的呼叫。易言之，搜尋出來的紀錄將不會再有排序的效果，但你能改由 profiles 來控制。最後的改變是去掉 split() 方法。JavsScript 1.0 並不支援 split()，但避開之後會降低執行效能。

天下沒有白吃的午餐

這是我剛進佛羅里達州立大學當新鮮人時，我的經濟學教授寫在黑板上的一句話：「天下沒有白吃的午餐。」換句話說，這些修正的確能相容於舊版的瀏覽器，但會使得搜尋引擎的機能降低、程式碼管理不易。

註　這是概略的說法，有些 MSIE 3.02 的確有支援 .js 檔案。

沒有了 .js 檔案，你必須把陣列 profiles 丟進 nav.html。這樣一來，nav.html 會變得比較亂，不容易管理。

sort() 方法並非整個搜尋操作的關鍵，但確實是難得的好幫手。訪客總希望看到有次序的查詢結果。當然，你可以自己在陣列中做好字母次序的安排，但這也不是件簡單的事。或者，你可以自行研發排序函式，以供搜尋引擎套用。sort() 並非問題的膠著點 —— JavaScript 1.0 就沒有 sort()，大家還是照用不誤。

讓程式更不容易出錯

訪客輸入的查詢字串中也可以含有管道字元（|）。因此，你可以新增功能，先把查詢字串中不相干的字元拿掉。這樣的話，應用程式就不容易出錯了。

刊登廣告看板

如果網站來客率高，為什麼不藉此招徠一些廣告收入？

怎麼做？試試看這個。假設有五個看板廣告，現在要隨機刊登出來，所謂隨機，就是沒有特定次序的意思。如果你有個陣列，內含多個廣告影像檔的 URL，就可以隨機擇取其一，下載到客戶端的瀏覽器顯示出來。下列是就是這個陣列的內容：

```
var adImages = new Array("pcAd.gif", "modemAd.gif", "webDevAd.gif");
```

接著是隨機加到回應網頁中：

```
document.writeln('<IMG SRC=' + ads[Math.floor(Math.random(ads.length))] +
                 '>');
```

新增搜尋功能

有了這樣的概念，讀者可以沉浸在程式設計的樂園？了。例如，我們可以讓訪客自行選擇要搜尋的陣列。訪客得利於此，就可以縮小搜尋的結果，增加搜尋精度。

我們可以在 nav.html 中的文字欄位下列出一組核取方塊，如下所示：

```
<INPUT TYPE=CHECKBOX NAME="group" VALUE="97">1997 Records<BR>
<INPUT TYPE=CHECKBOX NAME="group" VALUE="98">1998 Records<BR>
<INPUT TYPE=CHECKBOX NAME="group" VALUE="99">1999 Records<BR>
```

利用這一組核取方塊組，就可以讓訪客決定要搜尋的陣列，以此例而言，可定義 profiles97、profiles98、profiles99 這三個陣列。

有許多東西都可以加進搜尋引擎，以增加訪客搜尋的精度和廣度。其中之一是提供大小寫的差別。就目前的程式而言，我們忽略大小寫的不同，但你可以新增核取方塊，讓訪客自行決定是否要理會大小寫的問題。

你也可以擴充布林搜尋法則，以提昇搜尋的精緻度。除了 AND 和 OR 之外，還有 NOT、ONLY，甚至是 LIKE 也成。以下是幾個可能的布林搜尋法則：

AND

紀錄必須含有 AND 兩端的資料項。

OR

紀錄只需含有 OR 兩端的資料項的其中一項即可。

NOT

紀錄不能含有 NOT 右端的資料項。

ONLY

紀錄必須含有此資料項，而且這一筆紀錄必須是唯一的才算數。

LIKE

紀錄含有的資料項若發音和拼法都相近的話，就算是一筆。

這樣的任務非同小可（尤其是 LIKE），但來訪的訪客可能會覺得你的搜尋引擎也非同小可。

研發字群集

另一項受眾人歡迎而且實用的技術是建立字群集（cluster set）。字群集是一組預先定義好的字群，可自動回應預先定義好的搜尋結果。例如，如果訪客在查詢字串中出現了“mutual funds”的字眼，搜尋引擎就可以自動編製回應用的紀錄，內含貴公司的商品說明。這項技術需要多方規劃，但的確值得搜尋引擎研發。