本書內容簡介

本書內容大致分為三個部分。前面 12 章的內容是討論 JavaScript 語言，接下來 7 章是介紹網頁瀏覽器以及如何利用 JavaScript 設計出瀏覽器程式，最後 2 章則是專門介紹另一種 JavaScript 程式的開發環境：Node.js。書中共有五個**章節的實作專案**，說明比較大型的程式，帶各位讀者體驗一下真正的程式設計。

第一部分的內容會先利用前 4 個章節的篇幅介紹 JavaScript 語言的基本結構，討論控制結構（例如我們在導讀裡看到的關鍵字 `while`）、函式（自行撰寫的程式區塊）和資料結構。學會這些結構之後，各位就能寫出基本的程式。接下來的第 5、第 6 章則會介紹函式和物件的使用技巧，撰寫更**抽象**的程式碼和控制程式複雜度。

完成第一個簡單的宅配機器人專案後，程式語言部分的章節內容會繼續介紹錯誤處理和除錯、正規表達式（處理文字的重要工具）、模組化（另一種控制複雜度的防禦手段）以及非同步程式設計（處理需要時間的事件）。第二個專案章節會實作一個程式語言，作為本書第一部分內容的總結。

本書第二部分從第 13 章到 19 章的內容是介紹 Javascript 在瀏覽器上可以使用的工具，我們將學習如何將內容顯示在螢幕上（第 14 章和第 17 章）、回應使用者輸入的內容（第 15 章）以及網路通訊（第 18 章）。這部分也包含兩個專案章節，分別是開發平台遊戲和小畫家程式。

看完前面兩大部分之後，第 20 章會介紹另外一個開發環境——Node.js，第 21 章則是使用這項工具建置一個小型的網站。最後我們會在第 22 章討論如何提升 JavaScript 程式的執行速度，也就是 JavaScript 程式最佳化時，應該提出哪些考量。

7

實作專案：宅配機器人

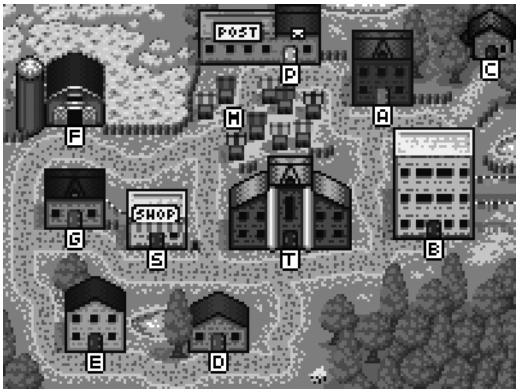
在各個「實作專案」章節裡，不會介紹讓各位讀者崩潰的新理論，先讓大家稍微喘口氣，一起應用之前學過的理論來實作程式。學習程式設計一定會需要理論，但閱讀和理解實際程式兩者同樣重要。

本章的實作專案是開發一個小程式，目的是在虛擬世界中自動執行任務。我們撰寫的程式會是一個能自動收發包裹的宅配機器人。

綠野村

綠野村是一個小村莊，整個村莊有 11 個地點和 14 條路，以下面這個陣列來表示綠野村的道路：

```
const roads = [
  "Alice's House-Bob's House",   "Alice's House-Cabin",
  "Alice's House-Post Office",   "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```



整個村莊的道路會形成一個**路網圖**（如上所示），由點（村莊裡各個地點）和線（連接個地點之間的聯絡道路）組成。本專案裡的宅配機器人將在這個地圖世界中穿梭。

表示道路名稱的字串陣列不太方便操作，因為我們感興趣的是機器人如何從指定地點到達目的地，所以我們要先將道路列表轉換成資料結構，告訴我們從每個地點可以直接到達的所有地點。

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (graph[from] == null) {
      graph[from] = [to];
    } else {
      graph[from].push(to);
    }
  }
  for (let [from, to] of edges.map(r => r.split("-"))) {
    addEdge(from, to);
    addEdge(to, from);
  }
  return graph;
}

const roadGraph = buildGraph(roads);
```

指定一個節線陣列（edges），`buildGraph` 函式會建立一個地圖物件，針對每一個節點，將與其有連結的節點儲存為一個陣列。字串陣列中表示道路字串的格式為「起點 - 終點」，現在我們要利用 `split` 方法，將這個陣列轉換成二維陣列，讓新陣列包含**起點和終點**的單獨字串。

宅配任務

我們的機器人會在村莊四處移動，包裹分散在村莊裡的各個地點，每個包裹都要宅配到某個地方。機器人還會到各個地點去收包裹，再將包裹配送到目的地。

機器人必須在每一個點自動決定下一步要去哪裡，當所有包裹配送完畢後，才算是完成任務。

為了模擬這個過程，我們必須定義一個可以描述這個情況的虛擬世界。這個模型要能告訴我們機器人和包裹目前所在的位置。當機器人決定移動到某個地點時，就需要更新模型以反映新的情況。

各位如果正想從物件導向程式設計的角度思考，第一個衝動可能是開始為這個虛擬世界的各個元素定義物件：為機器人建立一個類別、為包裹建立一個類別，可能還需要為地點也建立一個類別。再利用這些類別來儲存屬性，描述當前的狀態，例如某個地點堆放的包裹，我們可以在更新世界狀態時更改這些屬性。

但這樣的想法有誤，至少，許多人經常會產生這樣的誤解。某些事物感覺上像是物件，但不表示可以自動帶入為程式中的物件。反射性地為應用程式中的每個概念都撰寫一個類別，往往會留下一組相互關聯的物件，每個物件內部各自有不斷變化的狀態。這一類的程式通常最後都難以理解，而且很容易發生程式中斷的問題。

我們反而應該定義最小的集合值，將村莊的狀態濃縮在這個集合裡。這個集合包括機器人當前的位置和所有尚未配送的包裹，每個包裹底下都有目前所在位置和配送目的地的地址，大致的想法就是如此。

既然要簡化，就順便改成：機器人移動時不改變這個狀態，而是等到機器人移動後再計算出新的狀態。

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }

  move(destination) {
    if (!roadGraph[this.place].includes(destination)) {
      return this;
```

由於這個機器人不需要記住任何資料，所以忽略第二個參數（請記住，JavaScript 能以額外的參數呼叫函式，而且不會產生不好的效果），而且在回傳物件時省略 memory 屬性。

為了讓這個複雜的機器人運作，首先需要建立一個新狀態，包含某些包裹資料。靜態方法適合實作這個功能（此處範例中是直接將屬性新增到建構函式）。

```
VillageState.random = function(parcelCount = 5) {
  let parcels = [];
  for (let i = 0; i < parcelCount; i++) {
    let address = randomPick(Object.keys(roadGraph));
    let place;
    do {
      place = randomPick(Object.keys(roadGraph));
    } while (place == address);
    parcels.push({place, address});
  }
  return new VillageState("Post Office", parcels);
};
```

我們不希望有任何一個包裹出現寄出地和配送地相同的情況。基於這個理由，do 迴圈遇到和配送地址相同的地點時，會重新挑選新地點。

讓我們開始啟動這個虛擬世界。

```
runRobot(VillageState.random(), randomRobot);
// → 移動到 Marketplace（市集）
// → 移動到 Town Hall（村公所）
// → ...
// → 完成配送任務：63 趟
```

由於這個做法並沒有事先做好適當的規劃，因此，機器人要繞來繞去才能將包裹配送到目的地，後續內容很快就會介紹解決這個問題的方法。

郵務車路線

我們應該能夠找出比隨機機器人更好的方法，所以，我們從現實世界的郵件遞送工作中得到線索，找出一個簡單的改善方法。如果我們能找出一條路線，可以經過村莊裡的所有地點，機器人只要在這條路線上來回兩次，保證一定能完成任務。這條路線如以下所示（從郵局出發）：

22

提升 JavaScript 效能的技巧

在機器上執行電腦程式，需要在程式語言和機器本身的指令格式間架起一座橋樑，跨越兩者之間的差異。雖然可以跟第 11 章的做法一樣，撰寫某個程式來轉譯其他程式，但通常會透過編譯（翻譯）將程式轉換成機器碼。

像 C 和 Rust 這類的程式語言，其設計目的是大致表達出機器擅長的事，所以很容易提升程式語言編譯的效率。JavaScript 則採用了截然不同的設計方式，其設計焦點是著重於程式語言的簡單性和易用性，所以 JavaScript 提供的操作幾乎都沒有直接對應到機器的功能，因而造成 JavaScript 程式在編譯上變得更加困難。

然而，現代 JavaScript 引擎（負責編譯和執行 JavaScript 的程式）執行程式的速度，確實令人印象深刻。以 JavaScript 撰寫的程式，其執行速度可能只比效果相同的 C 或 Rust 程式慢幾倍而已，聽起來或許還有一大段差距，但舊版的 JavaScript 引擎通常比 C 語言慢上近百倍，其他採用類似設計的程式語言（例如 Python 和 Ruby），即使發展到現代實作版本，其效能依舊比 C 語言慢很多。和這些語言相比，現代 JavaScript 的執行速度快得驚人，快到你幾乎不會因為效能問題，而被迫改用另一種程式語言。

不過，我們可能還是需要調整程式碼，避免使用 JavaScript 語言中某些執行效能較慢的特性。本章會以一個迫切需要執行速度的程式為例，示範如何提升運行速度，並且透過這個過程來探討 JavaScript 引擎編譯程式的方式。

分段式編譯

首先，各位必須了解 JavaScript 編譯器跟傳統編譯器的做法不同，前者不會只編譯一次程式碼，而是在程式執行過程中，根據程式需要反覆編譯程式碼。

多數程式語言的做法是編譯一個大型程式，所以需要花一些時間編譯程式。由於這些程式會提前編譯完成，而且是以編譯後的形式發布程式，所以通常可以接受較長的編譯時間。

JavaScript 的情況則有所不同，網站可能包含大量以文字形式載入的程式碼，而且每次開啟網站時都必須重新編譯。如果每次進行編譯流程都要花五分鐘，使用者一定不樂見這個情況。因此，JavaScript 編譯器幾乎是在編譯的同時就要立即開始執行程式，即使是面對大型程式也是如此。

為了達成這個目標，JavaScript 編譯器採用了多種編譯策略。首次開啟網站時，會先以低成本的方式快速簡單地編譯程式腳本。這種編譯方式雖然不會產生很快的執行速度，但能讓程式腳本快速啟動，所以許多函式在首次呼叫前可能都不會進行編譯。

在一般程式中，絕大多數的程式碼只會執行幾次，有些甚至完全沒有出場的機會，處理這些部分的程式碼時，採取低成本的編譯策略就已經足夠，反正這些程式碼也不會佔用太多執行時間。然而，面對那些經常呼叫或是包含大量運算迴圈的程式碼，就要採取不同的編譯方式。執行程式期間，JavaScript 引擎會觀察每段程式碼的執行頻率，如果發現某段程式碼可能會耗費大量時間（通常稱為「**常用程式碼**」），則引擎會使用更進階但編譯速度較慢的編譯器重新進行編譯。這種編譯器的效能**更佳**，可以產出執行速度更快的程式碼。有些編譯器甚至會採取兩種以上的編譯策略，針對**特別**常用的程式碼，還會採用編譯成本更高的最佳化策略。

由於執行和編譯程式碼之間會交錯進行，意味著等到智慧型編譯器開始處理某段程式碼時，其實這段程式碼已經執行過好幾次了，讓編譯器能**觀察**程式碼的執行狀況，並且蒐集程式碼相關資訊。本章稍後會談到這個部分，了解編譯器如何利用這些資訊來建立更有效率的程式碼。

圖形配置

本章要再討論一個跟圖形問題有關的例子。圖形非常適合用來描述道路系統、網路、電腦程式的控制流方式等等，以下圖形表示南美洲各國和領地之間的邊界關係圖，連線代表這兩個國家之間有共享邊界：

像以上這樣從某個圖的定義資料中推導出視覺化圖形的過程，就稱為**圖形配置**（graph layout）。這個推導的過程包括為每個節點指定適當的位置，讓鄰近節點彼此盡量靠近，同時又不會互相擠在一起。相同圖形若採用隨機配置，會大幅提升圖形闡述的難度。