推薦序

您現在拿著的書很有趣。這是一本 JavaScript 的書，其中包含了用 C 編寫的範例，討論了如何使用外顯式的單執行緒程式語言來進行多執行緒，也提供了很好的範例來說明如何以及何時來刻意的阻止事件迴圈，即使專家多年來一直告訴您永遠不要做這樣的事。最後以一串很好的理由和警告，來說明為何您實際上可能並不會想使用本書所描述的機制。更重要的是，無論您的程式碼將在哪裡部署和執行，我都會認為這是一本任何的 JavaScript 開發人員都必須閱讀的書。

當我和一些公司合作來幫助他們建構具有更高效率、更高效能的 Node.js 和 JavaScript 應用程式時，我經常不得不先往後退一步，花時間討論一下開發人員對程式語言的許多常見誤解。例如，曾經有一位在 Java 和 .NET 開發方面有著長久經驗的工程師認為，在 JavaScript 中建立一個新的 promise，很像在 Java 中建立一個新執行緒（其實並不是），並且 promise 會允許 JavaScript 平行執行（其實它們不會）。在某次對話中提到，有人建立了一個 Node.js 應用程式，該應用程式產生了 1,000 多個同時運作的執行緒，但他不確定為什麼在只有 8 個邏輯 CPU 核心的機器上進行測試時，卻沒有看到預期的效能改進。從這些對話中我們得到的教訓很清楚：對於很多 JavaScript 開發人員來說，多執行緒、並行性和平行性仍然是非常陌生和困難的話題。

處理這些誤解直接促使我（與我的同事，也是 Node.js 技術指導委員會成員 Matteo Collina 合作）發展了 Broken Promises 研討會，我們在其中奠定 JavaScript 非同步程式設計的基礎——教導工程團隊如何更有效的對他們程式碼的執行順序和各種事件出現的時機進行推理。它還直接導致了 Piscina 開源專案（與 Node.js 核心貢獻者 Anna Henningsen 一起）的開發，該專案提供了基於 Node.js worker 執行緒的 worker 池模型的最佳實務實作。但這些只有助於解決一部分的挑戰。

在本書中，Bryan 和 Thomas 專業地概括了多執行緒開發的基礎，並巧妙地說明了各種 JavaScript 執行時期（runtime）（如 web 瀏覽器和 Node.js）如何透過一種程式語言來支援平行運算，而該語言並沒有內建支援平行運算的機制。因為提供多執行緒支援的責任落在了執行時期上，而且由於這些執行時期之間存在著許多差異，因此瀏覽器和 Node.js 等平台各以不同的方式來實作多執行緒。儘管它們共享相似的 API，但 Node.js 中的 worker 執行緒和 web 瀏覽器中的 web worker 執行緒實際上並不是同一回事。幾乎所有的瀏覽器都支援共享 worker、web worker 和 service worker，而且 worker 執行緒在 Node.js 中已經存在好幾年了，但它們對於 JavaScript 開發人員來說仍然是一個相對較新的概念。無論您的 JavaScript 在何處執行，本書都將提供重要的見解和資訊。然而，最重要的是，作者花時間確切地解釋了為什麼您應該關心 JavaScript 應用程式中的多執行緒。

— James Snell
Node.js 技術指導委員會成員

前言

Bryan 和我（Thomas）在接受日本行動遊戲開發公司 DeNA 舊金山分公司的採訪時第一次碰面。即使大多數高層管理人員很明顯都會拒絕，但那天晚上我們兩個人在 Node.js 聚會上閒逛之後，Bryan 去說服他們給我一個工作機會。

在 DeNA 期間，Bryan 和我致力於編寫可重用的 Node.js 模組，以便遊戲團隊可以建構他們的遊戲伺服器，並根據他們的遊戲需求組合適當的元件。我們一直在衡量效能，而指導遊戲團隊有關效能的一切就是工作的一部分；我們的伺服器不斷受到來自於仰賴 C++ 的行業的開發人員的審視。

我們兩人也會以其他身份合作。另一種這樣的角色是在一家名為 Intrinsic 的小型安全新創公司，在那裡我們專注於在既完整又精細的層級上強化 Node.js 應用程式，我懷疑世界上不會再看到類似的產品。效能調校也是該產品的一大關注點，因為客戶不想影響他們的產能。我們花了很多時間執行基準測試、研究火焰圖（flamegraph）並挖掘 Node.js 內部程式碼。如果 worker 執行緒模組在我們使用者請求的所有 Node.js 版本中都可用，毫無疑問我們會將它整合到產品中。

我們也在工作之外的場合上合作過。NodeSchool SF（*https://oreil.ly/TNS5w*）就是一個這樣的例子，我們都自願教導其他人如何使用 JavaScript 和建立 Node.js 程式。我們還在許多相同的會議和聚會上發言。

您的兩位作者都對 JavaScript 和 Node.js 充滿熱情，並熱衷於將它們教給別人並消除其中的誤解。當我們意識到有關建構多執行緒 JavaScript 應用程式文件說明極其缺乏時，我們知道我們必須要做什麼。這本書的誕生源自於我們不只希望向他人介紹 JavaScript 的能力，而且還希望有助於證明像 Node.js 這樣的平台，在建構能善用硬體的高性能服務這方面，與其他平台一樣強大。

目標讀者

本書的理想讀者是已經編寫了幾年 JavaScript 的工程師，但不一定要有編寫多執行緒應用程式的經驗，甚至不需要有使用 C++ 或 Java 等更傳統的多執行緒語言的經驗。我們確實包含了一些 C 的範例應用程式碼，因為把它視為一種多執行緒通用語言，但這並不是我們期望讀者熟悉甚至理解的東西。

如果您確實有使用此類語言的經驗，那就太好了，這本書將幫助您理解 JavaScript，它與您可能熟悉的任何語言具有同等的功能。另一方面，如果您只使用過 JavaScript 來編寫程式碼，那麼這本書也適合您。我們包含了橫跨多個學習層次的資訊；其中包括低階 API 參照（reference）、高階樣式，以及介於兩者之間的大量切入技術，用以填補其間的任何空白。

目標

本書最大的目標可能是讓社群能夠瞭解，使用 JavaScript 來建構多執行緒應用程式是可能的。傳統上，JavaScript 程式碼僅限用於單核心，並且的確有許多 Twitter 推文和論壇貼文如此描述這種語言。有了像*多執行緒 JavaScript* 這樣的標題，我們希望能徹底消除 JavaScript 應用程式僅限用於單核心這樣的觀念。

在更具體的層次上，本書的目標是教您（讀者）有關編寫多執行緒 JavaScript 應用程式的幾個層面。讀完本書後，您將瞭解瀏覽器中提供的各種 web worker API、它們的優缺點以及該何時使用。就 Node.js 而言，您將瞭解 worker 執行緒模組並比較它的 API 和瀏覽器中的 API。

本書聚焦於介紹建構多執行緒應用程式的兩種方法：一種使用訊息傳遞（message passing），另一種使用共享記憶體（shared memory）。透過閱讀本書，您將能瞭解用於實作每種方法的 API、何時該使用其中的一種方法或另一種方法，以及在哪些情況下可以將它們組合起來——您甚至會接觸到一些建立在這些方法之上的高階樣式。

第一章

簡介

電腦在過去要比現在單純多了。這並不是說使用它們或為它們編寫程式碼很容易，但從概念上講，要操作的東西少得多了。1980 年代的 PC 通常只有一個 8 位元 CPU 核心跟少量記憶體，您通常一次只能執行一個程式。我們現在認為的作業系統甚至不能與使用者正在互動的程式同時執行。

最終，人們會想要一次執行多個程式，於是多工（multitasking）處理誕生了。這允許作業系統透過在多個程式之間進行切換來同時執行它們。程式可以透過將執行權交給作業系統，來決定何時是執行另一個程式的適當時間。這種方法稱為**合作多工**（*cooperative multitasking*）。

在合作多工環境中，當程式因為任何原因無法執行時，其他程式都無法繼續執行。其他程式的這種中斷並不是我們想要的，因此作業系統最後轉成**搶奪式多工**（*preemptive multitasking*）。在此模型中，作業系統將使用自己的排程原則，來確定哪個程式將在何時在 CPU 上執行，而不是讓程式本身成為該何時切換執行的唯一決定者。直到今天，幾乎每個作業系統都是使用這種方法，即使在多核心系統上也是如此，因為我們一般會執行的程式數量都比 CPU 核心還多。

一次執行多個任務對程式設計師和使用者來說都是非常有用的。在執行緒出現之前，單一程式（也就是單個**程序**（*process*））不能同時執行多個任務。相反的，希望能夠並行的（concurrently）執行任務的程式設計師，若不是必須將任務分成較小的區塊並在程序中進行排程，不然就是要在分別的程序中執行分別的任務，並讓它們互相通訊。

即使在今天，在一些高階語言中，同時執行多個任務的合適方法，就是執行額外的程序。在某些語言例如 Ruby 和 Python 中，有一個**全域解譯鎖**（*global interpreter lock, GIL*），這意味著在某個給定時間只能執行一個執行緒。雖然這可以使記憶體管理更加務實，但它使得多執行緒程式設計喪失了對程式設計師的吸引力，取而代之的是採用了多個程序的作法。

直到最近，JavaScript 唯一可用的多工處理機制是將任務拆分成小塊，並對它們進行排程以供後續執行，且在使用 Node.js 的情況下，需執行額外的程序。我們通常會使用回呼（callback）或 promise 將程式碼分解為非同步單元。以這種方式編寫的典型程式碼塊可能類似於範例 1-1，透過回呼或 `await` 來分解運算。

範例 *1-1* 一段典型的非同步 *JavaScript* 程式碼，使用兩種不同的樣式

```
readFile(filename, (data) => {
  doSomethingWithData(data, (modifiedData) => {
    writeFile(modifiedData, () => {
      console.log('done');
    });
  });
});

// 或者

const data = await readFile(filename);
const modifiedData = await doSomethingWithData(data);
await writeFile(filename);
console.log('done');
```

今天，在所有主要的 JavaScript 環境中，我們都可以存取執行緒，和 Ruby 和 Python 不同的是，我們並沒有 GIL，這使得它們在執行 CPU 密集型（CPU-intensive）任務時實際上毫無用處。相反的，它進行了其他取捨，例如不能跨執行緒共享 JavaScript 物件（至少不是直接共享）。儘管如此，對於 JavaScript 開發人員來說，執行緒對於管制 CPU 密集型任務很有用。在瀏覽器中，還有一些特殊用途的執行緒，它們具有與主執行緒不同的功能集合。我們如何做到這一點的細節是後面章節的主題，但為了讓您先有一個概念，在瀏覽器中建立一個新執行緒來處理訊息，可以像範例 1-2 一樣簡單。

範例 *1-2.* 建立一個瀏覽器執行緒

```
const worker = new Worker('worker.js');
worker.postMessage('Hello, world');

// worker.js
self.onmessage = (msg) => console.log(msg.data);
```

本書的目的是探索和解釋 JavaScript 執行緒來作為一種程式設計的概念和工具。您將學習到如何使用它們，更重要的是，何時使用它們。不是每個問題都需要用執行緒來解決。甚至不是每個 CPU 密集型問題都需要用執行緒來解決。軟體開發人員的工作是評估問題和工具來確定最合適的解決方案。此處的目的是為您提供另一種工具和足夠的知識，以讓您瞭解何時使用它以及如何使用它。

什麼是執行緒？

在所有現代作業系統中，核心程式（kernel）之外的所有執行單元，都被組織成程序和執行緒。開發人員可以使用程序和執行緒以及它們之間的通訊，來為專案添加並行性 (concurrency)。在具有多個 CPU 核心的系統上，這也意味著增加平行性（parallelism）。

當您執行一個程式（例如 Node.js 或程式碼編輯器）時，您會啟動一個程序。這意味著程式碼會被載入到該程序獨有的記憶體空間中，而且程式在沒有向核心程式請求更多記憶體或映射至不同的記憶體空間之前是無法存取其他的記憶體空間的。沒有添加執行緒或額外的程序時，一次只能執行一個**指令**（*instruction*），而且會按照程式碼所規定的適當順序執行。如果您對此還不熟悉，您可以將指令視為一個程式碼單位，就像一行程式碼。（實際上，一個指令通常對應到處理器的組合語言（assembly）程式碼中的一行！）

一個程式可能會產生額外的程序，而這些程序都有自己的記憶體空間。這些程序並不會共享記憶體（除非它是透過額外的系統呼叫映射進來的）並且有自己的指令指標，這意味著每個程序可以在同一時間執行不同的指令。如果程序在同一個核心上執行，處理器可能會在程序之間來回切換，當某一個程序執行時，會暫時停止另一程序的執行。

一個程序也可以產生執行緒，而不是一直增大程序。執行緒就像一個程序，只不過它會和它所屬的程序共享記憶體空間。一個程序可以有多個執行緒，每個執行緒都有自己的指令指標。程序執行時的所有相關屬性也適用於執行緒。因為它們共享記憶體空間，所以很容易在執行緒間共享程式碼和其他的值，這使得它們在為程式添加並行性這方面比程序更有價值，但要付出的代價是程式設計的一些複雜性，我們將在本書的後面介紹這部份。

利用執行緒的一種典型方法，是將 CPU 密集型工作（如數學運算）卸載到附加的執行緒或執行緒池（pool）上，而主執行緒可以透過在一無窮迴圈中檢查新的互動，來自由的與使用者或其他程式進行外部互動。許多經典的 web 伺服器程式，例如 Apache，都使用這樣的系統來處理大量的 HTTP 請求（request）。這最終可能看起來類似於圖 1-1。在這個模型中，HTTP 請求資料被傳遞給一個 worker 執行緒進行處理，當回應（response）就緒時，它會被傳遞回主執行緒以傳回給使用者代理人（agent）。

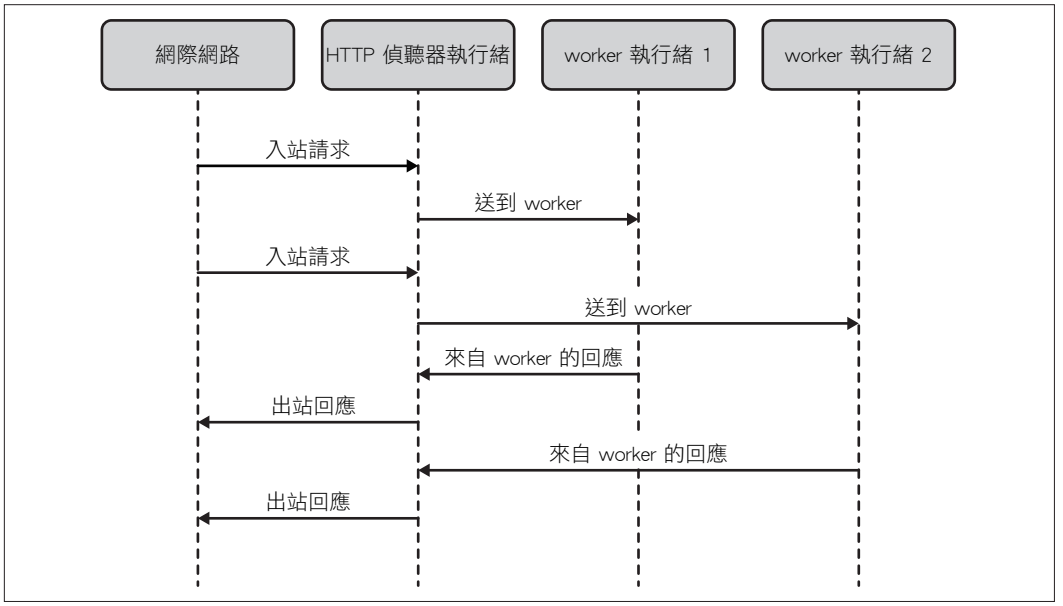

圖 1-1　可能在 HTTP 伺服務器中使用的 worker 執行緒

執行緒要能有用，需要能夠互相協調。這意味著它們必須能夠做一些事情，像是等待在其他執行緒上發生的事情，還有從它們那裡獲取資料。正如我們所討論過的，在執行緒之間有一個共享的記憶體空間，並且使用其他一些基本原語（primitive），我們可以建構能夠在執行緒之間傳遞訊息的系統。在許多案例中，會在語言或平台層級上提供這些類型的構造。

並行性 vs. 平行性

區分並行性和平行性是很重要的，因為在以多執行緒方式進行程式設計時，它們會經常出現。兩者是密切相關的術語，在某些情況下，它們可能會意指非常相似的事物。讓我們從一些定義開始。

並行性

任務在重疊的時間內執行。

平行性

任務在完全相同的時間執行。

雖然它們似乎指同一件事，但請考慮將任務分解為更小的部分然後交錯執行。在這種情況下，我們可以在不具有平行性的情況下達成並行，因為任務執行的時間範圍可以重疊。對於平行執行的任務而言，它們必須在**完全相同的時間**執行。一般而言，意思是它們必須同時在不同的 CPU 核心上執行。

請看一下圖 1-2。在此圖中，我們有兩個平行和並行執行的任務。在並行的情況下，在給定時間上只有一個任務正在執行，但在整個期間，會在兩個任務之間切換執行。這代表著它們是在重疊的時間內執行，因此符合並行的定義；在平行的情況下，兩個任務會同時執行，因此它們是平行執行的。由於它們也在重疊的時間段內執行，因此它們也是並行執行。平行是並行的子集合。

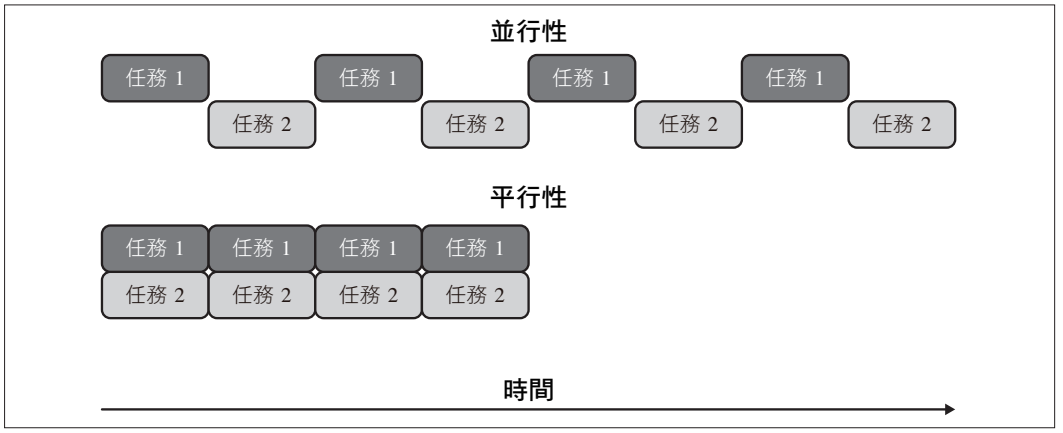

圖 1-2　並行 vs. 平行

執行緒不會自動提供平行性。系統硬體必須具有多個 CPU 核心才能實現這一點，而且作業系統排程器也必須決定要在不同的 CPU 核心上執行這些執行緒才行。在單核心系統上，或正在執行的執行緒數量多於 CPU 核心的系統上，多個執行緒可以透過在適當的時間，在它們之間進行切換來在單 CPU 上並行執行。此外，在 Ruby 和 Python 等具有 GIL 的語言中，執行緒被明確禁止提供平行性，因為在整個執行時間內，一次只能執行一個指令。

在考慮時間時這一點也很重要，因為在程式中添加執行緒通常是為了提高效能。如果您的系統是因為只有一個 CPU 核心可用，或已經載入了其他任務而只允許並行的話，那麼使用額外的執行緒可能不會有明顯的好處。事實上，執行緒之間同步（synchronization）和內容交換（context-switching）的額外負擔最終可能會使程式的效能變得更差。永遠要在應用程式所預期執行的條件下測量應用程式的效能，這樣您就可以驗證多執行緒程式設計模型是否真的對您有益。

單執行緒 JavaScript

歷史上，執行 JavaScript 的平台不提供任何執行緒的支援，因此該語言被認為是單執行緒的。當你聽到有人說 JavaScript 是單執行緒的，他們指的是這個歷史背景和它自然而然的程式設計風格。確實，儘管本書的標題是如此，但該語言本身並沒有任何內建功能來建立執行緒。這並不足為奇，因為它也沒有任何內建功能來與網路、裝置或檔案系統進行互動，或進行任何系統呼叫。事實上，即使像 `setTimeout()` 這樣的基本函數實際上也不是 JavaScript 的功能。相反的，虛擬機器（virtual machine, VM）嵌入的環境（例如 Node.js 或瀏覽器），會透過特定於環境的 API 來提供這些功能。

大多數 JavaScript 程式碼並不以執行緒作為並行原語（primitive），而是以運作在單執行緒上的事件導向（event-oriented）方式編寫。當使用者互動或 I/O 等各種事件發生時，它們會引發那些在之前設定為會在這些事件上執行的函數的執行。這些函數通常稱為**回呼**（*callback*），是 Node.js 和瀏覽器中非同步程式設計的核心。即使在 promise 或 `async/await` 語法中，回呼也是其底層原語。重要的是要認知到回呼並不是平行執行的，也不是與任何其他程式碼一起執行的；當回呼中的程式碼正在執行時，這會是當前正在執行的唯一程式碼。換句話說，在任何給定時間上只有一個呼叫堆疊（call stack）是活動的。

我們通常很容易認為運算是平行的，然而實際上它們是並行的。例如，假設您要開啟三個包含數字的檔案，分別命名為 *1.txt*、*2.txt* 和 *3.txt*，然後將結果相加並列印出來。在 Node.js 中，您可能會執行類似於範例 1-3 的操作。

範例 *1-3* 在 *Node.js* 中同時讀取檔案

```
import fs from 'fs/promises';

async function getNum(filename) {
  return parseInt(await fs.readFile(filename, 'utf8'), 10);
}

try {
  const numberPromises = [1, 2, 3].map(i =>=>getNum(`${i}.txt`));
  const numbers = await Promise.all(numberPromises);
  console.log(numbers[0] + numbers[1] + numbers[2]);
} catch (err) {
  console.error('Something went wrong:');
  console.error(err);
}
```

要執行此程式碼，請將其儲存在名為 *reader.js* 的檔案中，確保您有名稱為 *1.txt*、*2.txt* 和 *3.txt* 的文本檔案，而且每個檔案都包含了整數，然後使用 `node reader.js` 來執行程式。

由於我們使用了 `Promise.all()`，我們會等待全部的三個檔案都被讀取和解析。如果您稍微瞇一下眼，它甚至可能看起來像是本章稍後 C 範例中的 `pthread_join()`。然而，只因 promise 被一起建立並且一起等待，並不代表解決它們的程式碼會同時執行，而只是意味著它們的時間框是重疊的——仍然只有一個指令指標，而且一次只有一個指令正在執行。

在沒有執行緒的情況下，只有一個 JavaScript 環境可以使用。這意味著只有一個 VM 實例、一個指令指標和一個垃圾收集器（garbage collector）實例。只有一個指令指標的意思是 JavaScript 解譯器在任何給定時間上只能執行一個指令。但這並不意味著我們受限於只有一個全域物件。在瀏覽器和 Node.js 中，我們都可以使用領域（realm）（*https://oreil.ly/uy7E2*）。

領域可以被認為是提供給 JavaScript 程式碼的 JavaScript 環境實例，意指每個領域都有自己的全域物件，以及全域物件的所有相關屬性，例如 `Date` 等內建類別和 `Math` 等其他物件。全域物件在 Node.js 中稱為 `global`，在瀏覽器中稱為 `window`，但在兩者的現代版本中，您可以將全域物件稱為 `globalThis`。

在瀏覽器中，網頁中的每個框架（frame）都有一個適用於裏面所有 JavaScript 的領域。因為每個框架裏面都有自己的 `Object` 副本和其他的原語，您會注意到它們有自己的繼承樹，而且 `instanceof` 在對來自不同領域的物件進行操作時，可能不會像您所期望的那樣工作。範例 1-4 對此進行了展示。

範例 1-4 來自瀏覽器中不同框架的物件

```
const iframe = document.createElement('iframe');
document.body.appendChild(iframe);
const FrameObject = iframe.contentWindow.Object; ❶

console.log(Object === FrameObject); ❷
console.log(new Object() instanceof FrameObject); ❸
console.log(FrameObject.name); ❹
```

❶ `iframe` 內的全域物件可以透過 `contentWindow` 屬性來存取。

❷ 這會傳回 false，所以框架內的 `Object` 與主框架內的物件不同。

❸ `instanceof` 的賦值結果為 `false`，正如我們所預期的一樣，因為它們不是同一個 `Object`。

❹ 儘管如此，建構子函數（constructor）擁有相同的 `name` 屬性。

在 Node.js 中，可以使用 `vm.createContext()` 函數來建構領域，如範例 1-5 所示。在 Node.js 中，領域被稱為 Context。適用於瀏覽器框架的同樣規則和屬性也適用於 Context，但在 Context 中，您無權存取任何全域屬性或任何其他可能在 Node.js 檔案範疇（scope）內的內容；如果要使用這些功能，則需要手動將它們傳入 Context。

範例 1-5 Node.js 中來自新 Context 的物件

```
const vm = require('vm');
const ContextObject = vm.runInNewContext('Object'); ❶

console.log(Object === ContextObject); ❷
console.log(new Object() instanceof ContextObject); ❸
console.log(ContextObject.name); ❹
```

❶ 我們可以使用 `runInNewContext` 從新的 context 中獲取物件。

❷ 這將傳回 false，因為對於瀏覽器的 iframe 來說，Context 中的物件與主 Context 中的物件不同。

❸ 同樣的，`instanceof` 的賦值結果為 `false`。

❹ 再一次，建構子函數具有相同的 `name` 屬性。

在任何一種領域的情況下，注意到我們仍然只有一個指令指標是很重要的，並且一次只有一個領域的程式碼在執行，因為我們目前仍然只在討論單執行緒的執行。

隱藏執行緒

雖然您的 JavaScript 程式碼，至少在預設情況下，可能會在單執行緒環境中執行，但這並不代表執行您的程式碼的程序也會是單執行緒的。事實上，可能會使用許多執行緒來使該程式碼平穩且有效率的執行。常常有人誤會 Node.js 是一個單執行緒程序。

像 V8 這樣的現代 JavaScript 引擎使用分別的執行緒來處理垃圾收集，和其他不需要在 JavaScript 執行中發生的功能。此外，平台執行環境本身，也可能會使用額外的執行緒來提供其他功能。

在 Node.js 中，`libuv` 被用作是與 OS 無關的非同步 I/O 介面，此外由於並非所有由系統提供的 I/O 介面都是非同步的，因此它使用一個 worker 執行緒池，來避免在使用其他阻擋式（blocking）API 時阻擋了程式碼，例如檔案系統 API。預設情況下，會生成四個執行緒，但此數量可透過 `UV_THREADPOOL_SIZE` 環境變數進行配置，最多可達 1,024。

在 Linux 系統上，您可以透過在給定程序上使用 `top -H` 來查看這些額外的執行緒。在範例 1-6 中，啟動了一個簡單的 Node.js web 伺服器，並記錄了 PID 並將其傳遞給 `top`。您可以看到各種 V8 和 `libuv` 執行緒加總後，有多達七個執行緒，包括執行 JavaScript 程式碼的執行緒。您可以用自己的 Node.js 程式試試看，甚至可以嘗試改變 `UV_THREADPOOL_SIZE` 環境變數來查看執行緒的數量變化。

範例 *1-6* `top` 的輸出，顯示 *Node.js* 程序中的執行緒

```
$ top -H -p 81862
top - 14:18:49 up 1 day, 23:18,  1 user,  load average: 0.59, 0.82, 0.83
Threads:   7 total,   0 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.2 us,  0.0 sy,  0.0 ni, 97.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15455.1 total,   2727.9 free,   5520.4 used,   7206.8 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   8717.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  81862 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.03 node
  81863 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00 node
  81864 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00 node
  81865 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00 node
  81866 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00 node
  81867 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00 node
  81868 bengl     20   0  577084  29272  25064 S   0.0   0.2   0:00.00 node
```

瀏覽器同樣的會在用來執行 JavaScript 的執行緒之外的執行緒中執行許多任務，例如檔案物件模型（Document Object Model, DOM）渲染。像我們對 Node.js 所做的那樣使用 `top -H` 進行的實驗會產生相似的少量執行緒。現代瀏覽器透過使用多個程序以隔離方式添加安全層來進一步實現這一點。

在為您的應用程式進行資源規劃練習時，考慮這些額外的執行緒很重要。您永遠不應該只因為 JavaScript 是單執行緒的，就假設您的 JavaScript 應用程式也只會使用一個執行緒。例如，在產出版本的 Node.js 應用程式中，應該要測量應用程式使用的執行緒數量，並相對應地進行規劃。別忘了 Node.js 生態系統中的許多原生附加功能，也會產生自己的執行緒，所以在逐應用程式（application-by-application）的基礎上完成這個練習很重要。

C 的執行緒：透過 Happycoin 致富

執行緒顯然不是 JavaScript 所獨有的，它們是作業系統等級的一個長期概念，獨立於語言之外。讓我們探討一下執行緒程式在 C 中的樣子。在這裏 C 是一個明顯的選擇，因為 C 語言的執行緒介面是其他多數高階語言執行緒實作的基礎，即使看起來有不同的語意。

讓我們從一個範例開始。想像一個名為 Happycoin 的簡單且不切實際的加密貨幣的工作量證明（proof-of-work）演算法，如下所示：

1. 產生一個隨機的無正負號（unsigned）64 位元整數。

2. 判斷整數是否是快樂的。

3. 如果不快樂，它就不是 Happycoin。

4. 如果不能被 10,000 整除，它就不是 Happycoin。

5. 否則，它是一個 Happycoin。

如果一個數字在用組成它的阿拉伯數字平方和替換它時最終會變為 1，並且循環直到 1 出現或之前看過的數字出現時，那麼它就是快樂的。維基百科對其進行了清楚的定義（*https://oreil.ly/vRr3P*）並指出說，如果任何先前看到的數字出現時，則將出現 4，反之亦然。您可能會注意到我們的演算法其實不用那麼昂貴，因為我們可以在檢查快樂度之前檢查是否可整除。但我們是故意的，因為我們試圖要展現繁重的工作量。

讓我們建構一個簡單的 C 程式，該程式會執行工作量證明演算法 10,000,000 次，並且列印出所找到的任何 Happycoin 以及它們的計數。

在此處的編譯步驟中的 `cc` 可以換成 `gcc` 或 `clang`，取決於您可以使用哪個。在大多數系統上，`cc` 是 `gcc` 或 `clang` 的別名，所以它就是我們會在這裡使用的。

Windows 使用者可能需要在這裡做一些額外的工作才能在 Visual Studio 中執行此操作，並且執行緒範例在 Windows 上無法開箱即用，因為它使用可攜作業系統介面（Portable Operating System Interface, POSIX）執行緒而不是 Windows 執行緒，兩者是不同的。為了簡化在 Windows 上的嘗試，建議使用適用於 Linux 的 Windows 子系統（Windows Subsystem for Linux），以便您擁有與 POSIX 相容的環境。

只有主執行緒

在名為 *ch1-c-threads/* 的目錄中建立一個名為 *happycoin.c* 的檔案。我們將在本節的過程中建構此檔案的內容。首先，添加如範例 1-7 所示的程式碼。

範例 *1-7 ch1-c-threads/happycoin.c*

```
#include<inttypes.h>
#include<stdbool.h>
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

uint64_t random64(uint32_t * seed) {
  uint64_t result;
  uint8_t * result8 = (uint8_t *)&result; ❶
  for (size_t i = 0; i< sizeof(result); i++) {
    result8[i] = rand_r(seed);
  }
  return result;
}
```

❶這一行使用了指標，如果您主要來自 JavaScript 背景，您可能不太熟悉指標。把這裡所發生的事情長話短說不是是：`result8` 是一個由 8 個 8 位元無正負號整數組成的陣列，由與 `result` 相同的記憶體所支援，而 `result` 是一個 64 位元無正負號整數。

我們添加了一堆 `includes`，它們為我們提供了一些方便的東西，例如型別（type）、I/O 函數，以及我們即將用到的時間和亂數函數。由於該演算法需要產生一個隨機的 64 位元無正負號整數（即 `uint64_t`），我們需要 8 個隨機位元組，它們會由 `random64()` 透過呼叫 `rand_r()` 來提供給我們，直到我們有了足夠的位元組。由於 `rand_r()` 還需要對種子的參照，因此我們也會將其傳遞給 `random64()`。

現在讓我們添加我們的快樂數字計算過程，如範例 1-8 所示。

範例 1-8 ch1-c-threads/happycoin.c

```
uint64_t sum_digits_squared(uint64_t num) {
  uint64_t total = 0;
  while (num >0) {
    uint64_t num_mod_base = num % 10;
    total += num_mod_base * num_mod_base;
    num = num / 10;
  }
  return total;
}

bool is_happy(uint64_t num) {
  while (num != 1 && num != 4) {
    num = sum_digits_squared(num);
  }
  return num == 1;
}

bool is_happycoin(uint64_t num) {
  return is_happy(num) && num % 10000 == 0;
}
```

為了取得 `sum_digits_squared` 中數字的平方和，我們使用 `mod` 運算子 `%` 從右到左來取得每個數字，再將其平方，而後再加到我們的總數中。然後我們在 `is_happy` 中的迴圈裏使用這個函數，當數字為 1 或 4 時停止。我們會在 1 的時候停止，因為這指出數字是快樂的。我們也會在 4 的時候停止，因為這指出我們已經陷入永遠不會停在 1 的無窮迴圈。最後，在 `is_happycoin()` 中，我們會檢查一個數字是否是快樂的並且是否可以被 10,000 整除。

我們把這一切都包含在 `main()` 函數中，如範例 1-9 所示。

範例 1-9 ch1-c-threads/happycoin.c

```
int main() {
  uint32_t seed = time(NULL);
  int count = 0;
  for (int i = 1; i< 10000000; i++) {
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      printf("%" PRIu64 " ", random_num);
      count++;
    }
  }
```

```
  printf("\ncount %d\n", count);
  return 0;
}
```

首先，我們需要一個亂數產生器的種子。目前的時間和任何其他種子都一樣合適，因此我們將透過 `time()` 來使用它。然後，我們將循環 10,000,000 次，首先從 `random64()` 中取得一個亂數，然後檢查它是否是 Happycoin。如果是的話，我們將遞增計數並把數字列印出來。`printf()` 呼叫中奇怪的 `PRIu64` 語法，對於要能正確列印 64 位元無正負號整數是必要的。當迴圈完成時，我們會列印計數並退出程式。

要編譯和執行此程式，請在 *ch1-c-threads* 目錄中使用以下命令。

```
$ cc -o happycoin happycoin.c
$ ./happycoin
```

您將在第一行中得到一份被找到的 Happycoin 列表，並在下一行取得它們的計數。對於某次的程式執行，它可能如下所示：

```
1102354119730451000 0 ...  [ 167 more entries ] ... 770541398378840000
count 169
```

執行這個程式需要很長的時間；在一般電腦上要花上大約 2 秒。在這種情況下，執行緒可以用來加快速度，因為相同的大型數學運算會執行許多次的迭代。

讓我們繼續將此範例轉換為多執行緒程式。

四個 worker 執行緒

我們將設置四個執行緒，每個執行緒將執行迴圈的四分之一迭代，此迴圈會產生一個亂數並測試它是否是 Happycoin。

在 POSIX C 中，執行緒由 `pthread_*` 函數家族來管理。`pthread_create()` 是用於建立執行緒，傳入的函數將在該執行緒上執行。程式流在主執行緒上繼續執行。程式可以透過呼叫 `pthread_join()` 來等待執行緒完成。您可以透過 `pthread_create()` 將參數傳遞給在執行緒上執行的函數，並從 `pthread_join()` 取得傳回值。

在我們的程式中，我們將在名為 `get_happycoins()` 的函數中獨立出 Happycoin 的產生過程，而這就是會在我們的執行緒中執行的東西。我們將建立四個執行緒，然後立即等待它們完成。每當從執行緒得到結果時，我們會將它們輸出並儲存計數值，以便可以在最後列印總數。為了幫忙將結果傳回來，我們將建立一個名為 `happy_result` 的簡單 `struct`。

複製您現有的 *happycoin.c* 並將其命名為 `happycoin-threads.c`。然後在新檔案中，將範例 1-10 中的程式碼插入檔案裏的最後一個 `#include` 之後。

範例 *1-10 ch1-c-threads/happycoin-threads.c*

```
#include<pthread.h>

struct happy_result {
  size_t count;
  uint64_t * nums;
};
```

第一行引入（include） `pthread.h`，它讓可以存取我們需要的各種執行緒函數。然後定義 `struct happy_result`，稍後將使用它作為我們的執行緒函數 `get_happycoins()` 的傳回值。它會儲存一個由找到的 happycoin 所構成的陣列，此處會以一個指標來表達此陣列，也會儲存這些 happycoin 的計數。

現在，去把整個 `main()` 函數刪除吧，因為我們將要換掉它。首先，讓我們在範例 1-11 中添加 `get_happycoins()` 函數，這是將在我們的 worker 執行緒上執行的程式碼。

範例 *1-11 ch1-c-threads/happycoin-threads.c*

```
void * get_happycoins(void * arg) {
  int attempts = *(int *)arg; ❶
  int limit = attempts/10000;
  uint32_t seed = time(NULL);
  uint64_t * nums = malloc(limit * sizeof(uint64_t));
  struct happy_result * result = malloc(sizeof(struct happy_result));
  result->nums = nums;
  result->count = 0;
  for (int i = 1; i< attempts; i++) {
    if (result->count == limit) {
      break;
    }
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      result->nums[result->count++] = random_num;
    }
  }
  return (void *)result;
}
```

❶這個奇怪的指標鑄型（cast）基本上是說「把這個任意指標當作是一個指向 `int` 的指標，然後幫我取得那個 `int` 的值。」

您會注意到這個函數接受一個 `void *` 並傳回一個 `void *`。這就是 `pthread_create()` 所期望的函數簽名（signature），所以在此並無其他選擇。這意味著必須將我們的引數（argument）鑄型為我們希望它們成為的樣子。我們將傳遞嘗試的次數，因此將引數鑄型為 `int`。然後，我們將像在前面的例子中一樣設定種子，但這次它會發生在我們的執行緒函數中，所以每個執行緒都會得到一個不同的種子。

在為我們的陣列和 `struct happy_result` 配置足夠的空間後，繼續進行與單執行緒範例中的 `main()` 相同的迴圈，只是這次將結果放入 `struct` 中而不是列印它們。迴圈完成後，將 `struct` 以指標方式傳回，我們會將其鑄型為 `void *` 以滿足函數簽名。這就是將資訊傳遞回主執行緒的方式，而這將會是有意義的。

這展示了我們無法從程序中取得的執行緒的關鍵屬性之一，也就是共享記憶體空間。例如，如果我們使用程序而不是執行緒，並使用一些**程序間通訊**（*interprocess communication, IPC*）機制將結果傳回的話，我們將無法簡單的將記憶體位址傳回主程序，因為主程序無法存取 worker 程序的記憶體。多虧了虛擬記憶體（virtual memory），記憶體位址可能會參照完全在主程序中的其他內容。我們必須透過 IPC 通道將整個值傳回，而不是只傳遞指標，這會造成額外的效能負擔。但因為我們是使用執行緒而不是程序，所以可以只使用指標，這樣主執行緒就可以同樣使用它。

不過，共享記憶體並非毋需進行取捨。在我們的例子中，worker 執行緒不需要使用它現在傳遞給主執行緒的記憶體。不過執行緒並非總是如此。在很多情況下，我們需要正確的管理執行緒如何透過同步（synchronization）來存取共享記憶體；否則可能會出現一些不可預測的結果。我們將在第四章和第五章詳細介紹它在 JavaScript 中的工作原理。

現在，讓我們用範例 1-12 中的 `main()` 函數來總結一下。

範例 *1-12 ch1-c-threads/happycoin-threads.c*

```
#define THREAD_COUNT 4

int main() {
  pthread_t thread [THREAD_COUNT];

  int attempts = 10000000/THREAD_COUNT;
  int count = 0;
  for (int i = 0; i< THREAD_COUNT; i++) {
    pthread_create(&thread[i], NULL, get_happycoins, &attempts);
  }
  for (int j = 0; j< THREAD_COUNT; j++) {
    struct happy_result * result;
    pthread_join(thread[j], (void **)&result);
```

```
    count += result->count;
    for (int k = 0; k< result->count; k++) {
      printf("%" PRIu64 " ", result->nums[k]);
    }
  }
  printf("\ncount %d\n", count);
  return 0;
}
```

首先，我們將我們的四個執行緒宣告為堆疊上的一個陣列。然後，我們將所需的嘗試次數（10,000,000）除以執行緒數量。這是將作為引數傳遞給 `get_happycoins()` 的內容，我們在第一個迴圈中看到它，此迴圈使用 `pthread_create()` 建立每個執行緒，並將每個執行緒的嘗試次數作為引數傳遞。在下一個迴圈中，我們使用 `pthread_join()` 來等待每個執行緒完成它們的執行。然後我們可以列印所有執行緒的結果和總數，就像我們在單執行緒範例中所做的那樣。

這個程式會洩漏記憶體。C 和其他一些語言中的多執行緒程式設計的一個難處是，很容易忘記記憶體分配的位置和時間，以及應該在何處和何時釋放記憶體。看看您是否可以修改此處的程式碼以確保程式退出時會釋放所有從堆積（heap）配置的記憶體。

更改完成後，您可以在 *ch1-c-threads* 目錄中使用以下命令編譯和執行此程式。

```
$ cc -pthread -o happycoin-threads happycoin-threads.c
$ ./happycoin-threads
```

輸出看起來會像這樣子：

```
2466431682927540000 ... [ 154 more entries ] ... 15764177621931310000
count 156
```

您會注意到與單執行緒範例類似的輸出，[1] 您還會注意到它會更快一些。在一般電腦上，它會在大約 0.8 秒內完成。這並沒有快四倍，因為主執行緒包含了一些初始化的額外負擔，還有列印結果的成本。我們可以在正在執行工作的那個執行緒準備好結果時立即列印結果，但是如果我們這樣做，結果可能會在輸出中相互破壞，因為沒有什麼方法可以阻止兩個執行緒同時列印到輸出流。透過將結果送出到主執行緒，我們可以在那裡協調結果的列印，以使輸出結果不會被弄亂。

[1] 多執行緒範例的總計數與單執行緒範例的不同的這一個事實是無關緊要的，因為計數取決於有多少亂數恰好是 Happycoin。兩次不同的執行結果將會完全不同。

這說明了執行緒程式碼的主要優點和一個缺點。一方面，它對於拆分計算量大的任務很有用，以便它們可以平行執行；另一方面，我們需要確保某些事件能夠正確的同步，以免發生奇怪的錯誤。在使用任何語言在程式碼中添加執行緒時，確保適當的使用方式是值得的。此外，和其他任何想要製作更快程式的練習一樣，總是要進行測量。如果結果證明不會給您帶來任何實際上的好處，那麼您不會希望在您的應用程式中添加了執行緒程式碼的複雜性。

任何支援執行緒的程式語言都會提供一些機制來建立和銷毀執行緒、在執行緒之間傳遞訊息、以及與執行緒之間共享的資料進行互動。這些機制在每種語言中看起來可能並不相同，因為語言及其範式（paradigm）不同，它們的平行程式設計的程式模型也會不同。既然我們已經探索了像 C 這樣的低階語言中的執行緒程式的樣子，讓我們也深入研究 JavaScript。事情看起來會有點不同，但正如您將看到的，原則其實是一樣的。