第二章

# 透過效能分析來找出瓶頸

**看完這一章之後，你可以回答這些問題**

- 如何找出程式碼的速度與 RAM 瓶頸？
- 如何分析 CPU 與記憶體的使用量？
- 該分析得多深？
- 如何分析長期運行的 app ？
- CPython 的底層是什麼情況？
- 如何在調整效能的同時，讓程式碼維持正確？

**效能分析**（*profiling*）可讓我們找出瓶頸，進而用最少的工作量提升最多效能。雖然我們都希望用最少的工作量來大幅提升速度，並降低資源使用量，但實際上，你的目標是讓程式碼跑得「夠快」而且「夠精簡」，以滿足需求。效能分析可讓你用最少的整體勞力做出最務實的決策。

任何一種可測量的資源都能夠加以分析（而不是只有 CPU！）。本章將探討 CPU 時間與記憶體的使用。你也可以用類似的技術來評估網路頻寬與磁碟 I/O。

如果程式跑得太慢，或使用太多 RAM，你就要修正罪魁禍首。當然，你也可以跳過效能分析，修正你所認為的問題所在，但若是如此，你要特別小心，因為你會經常「修正」錯誤的東西。與其依靠直覺，先建立假設並分析效能，再修改程式碼結構，才是更明智的做法。

有時懶惰反而是好事。先做效能分析可以快速找出有待解決的瓶頸，接著只要適度優化其中一部分，就能獲得所需的效能。如果你跳過效能分析就直接做優化，最終很可能會做更多白費力氣的工作。務必根據分析的結果採取行動。

## 有效率地分析

分析的首要目的是測試一個具代表性的系統，以找出緩慢的部分（或使用太多 RAM，或造成太多磁碟 I/O 或網路 I/O）。效能分析通常有額外開銷（典型的情況是變慢 10 倍至 100 倍），但你仍然希望在盡可能接近真實情境之下執行程式碼，所以你要提取測試案例，並隔離需要測試的部分。最好將它寫在它自己的模組裡面。

本章的第一部分介紹的基本技術包括 IPython 神奇的 `%timeit`、`time.time()`，以及計時用的 decorator。你可以用這些技術來瞭解陳述式與函式的行為。

接著我們會探討 `cProfile`（第 42 頁的「使用 cProfile 模組」），說明如何使用這種內建工具來瞭解哪些函式的執行時間最久。透過它可以用更高的視野來看問題，進而將注意力放在關鍵的函式上。

接下來，我們要看 `line_profiler`（第 49 頁的「使用 line_profiler 來逐行測量」），它可以逐行分析你選擇的函式。它輸出的結果包含每一行被呼叫的次數，以及各行花掉的時間百分比。這正是瞭解哪些部分跑得慢及原因的資訊。

有了 `line_profiler` 的分析結果之後，你就掌握進一步使用編譯器（第 8 章）所需的資訊了 —— 如果沒有先做效能分析就嘗試編譯，你可能只是讓某段非瓶頸程式變快而已，最終仍無法改善整體效能。

第 6 章會教你如何使用 `perf stat` 來瞭解 CPU 最終執行的指令數量，以及 CPU 的快取的使用效率。它們可讓你用進階的方式調整矩陣運算。看完這一章之後，你可以看一下範例 6-8。因為這個範例探討的是 `numpy` 陣列的使用情境，所以留到之後再介紹。

看完 `line_profiler` 之後，如果你正在處理長期運行的系統，你可以使用 `py-spy` 來瞭解正在運行的 Python 程序。

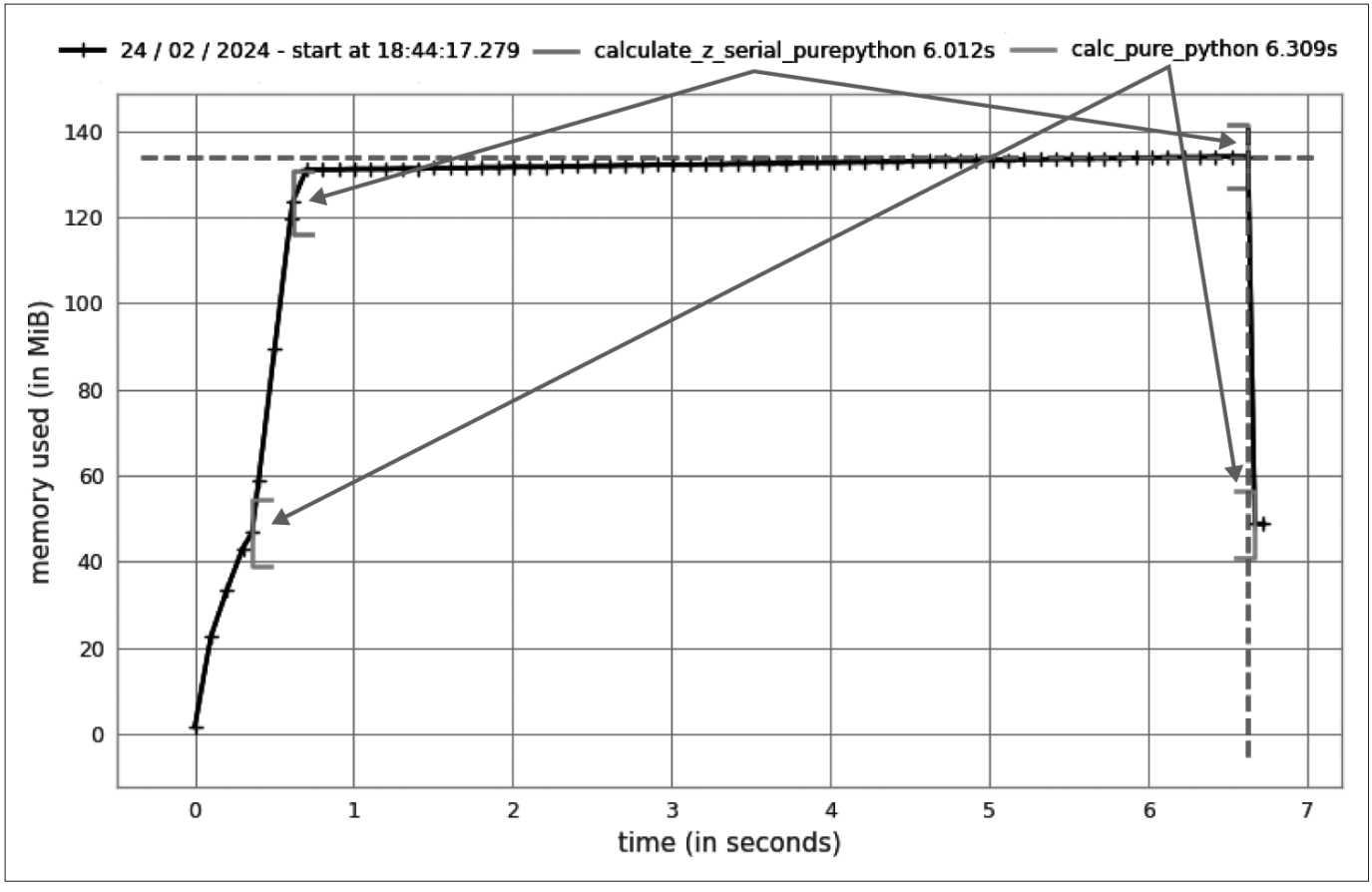


圖 2-6　使用 mprof 來繪製 memory_profiler 報告

第八章

# 編譯為 C

**看完這一章之後，你可以回答這些問題**

- 如何讓 Python 程式以更低階的程式碼執行？
- JIT 編譯器與 AOT 編譯器有什麼不同？
- 編譯過的 Python 程式處理哪些任務的速度比原生 Python 更快？
- 為什麼型態註記（type annotations）可讓編譯過的 Python 程式碼更快？
- 如何使用 C 或 Fortran 來為 Python 編寫模組？
- 如何在 Python 中使用 C 或 Fortran 的程式庫？

要讓程式跑得更快，最簡單的方法就是減少它的工作量。如果你已經選擇很好的演算法，並且降低需要處理的資料量了，執行較少指令最簡單的方法是將程式編譯成機器碼。

Python 為此提供了多種選項，包括以純 C 為基礎的編譯方式（如 Cython）、利用 Numba 的 LLVM 編譯技術，以及替代虛擬機器 PyPy（它內建即時編譯器，JIT）。當你從中選擇時，應兼顧程式碼的可調整性與團隊開發速度。

這些工具都會在你的工具鏈加入新的依賴項目。其中 Cython 會讓你使用新的語言類型（Python 與 C 的混合），也就是說，你必須學習新技能。Cython 的新語言可能會減緩團隊的開發速度，因為不瞭解 C 的成員可能難以維護這種程式碼，但在實務上，這可能只是小問題，因為我們通常只會在仔細挑選的小範圍內使用 Cython。

值得注意的是，分析 CPU 與記憶體的效能往往會促使你開始思考能不能使用更高階的演算法優化。這些演算法層面的改變（例如加入額外邏輯以避免重複運算，或使用快取以避免重新計算）有助於避免在程式中執行不必要的工作，而 Python 的高表達力，可幫助你發現這類演算法層面上的優化機會。Radim Řehůřek 在「Making Deep Learning Fly with RadimRehurek.com (2014)」的第 477 頁中探討了為什麼 Python 的實作有時能勝過純 C 的實作。

在這一章，我們將討論以下主題：

- Cython，最常被用來編譯成 C 的工具，涵蓋 `numpy` 與一般 Python 程式碼（需要稍微懂 C）
- Numba，一種專門處理 `numpy` 程式的編譯器
- PyPy，一種穩定的即時編譯器，一般用於非 `numpy` 程式碼，是標準 Python 執行檔的替代方案

在本章稍後，我們將介紹外部函式介面，它們可將 C 程式編譯成 Python 的擴展模組。Python 的原生 API 可以搭配 `ctypes`，或 `cffi`（來自 PyPy 的作者），以及 `f2py` 這個 Fortran 轉 Python 的轉換器。

## 可能提升多少速度？

如果你的問題適合走編譯路線，它的效能完全有機會提升一個數量級甚至更多。接下來要介紹在單一核心來加速一至兩個數量級的各種方法，以及如何透過 OpenMP 在多核心上自動分配工作並同步結果，這幾乎不需要開發者額外付出太多心力。

在編譯後可以跑得更快的 Python 程式碼多半是數學運算型的，而且有許多重複執行相同操作的迴圈。那些迴圈很可能產生大量臨時物件。

而呼叫外部程式庫的程式碼（例如正規表達式、字串操作，以及呼叫資料庫程式庫）不太可能在編譯後提升任何速度。I/O-bound 的程式也不太可能會有明顯的加速。

如果 Python 程式主要呼叫向量化的 numpy 程式，在編譯之後應該也不會跑得更快，除非被編譯的主要是 Python 程式碼（而且大多數是迴圈程序）。我們已經在第 6 章看過 numpy 操作了，由於中間物件不多，編譯沒有太大幫助。

整體而言，編譯過的程式碼幾乎不可能比手寫的 C 程式更快，但它也不太可能慢很多。用 Python 產生的 C 程式碼很有可能跑得和手寫的 C 程式一樣快，除非那位 C 程式開發者特別擅長根據電腦的架構微調 C 程式碼。

對於以數學為主的程式來說，手寫的 Fortran 或許可以勝過等效的 C 程式，但同樣地，這可能需要專家級的知識。整體而言，編譯後的結果（很可能使用 Cython）會相當接近手寫的 C 程式碼，這對大多數程式開發者來說已經夠用了。

當你分析與調整演算法時，請記住圖 8-1。花一點心力藉著分析效能來瞭解程式碼，應該可以讓你在演算法層面上做出更聰明的選擇。在這之後，再花一些心力使用編譯器，就能額外提升效能。或許你還會持續調整演算法，但是當你投入越來越多精力，換來的改進卻越來越少時，請不要感到奇怪。你必須要知道何時額外的努力已經沒有價值了。

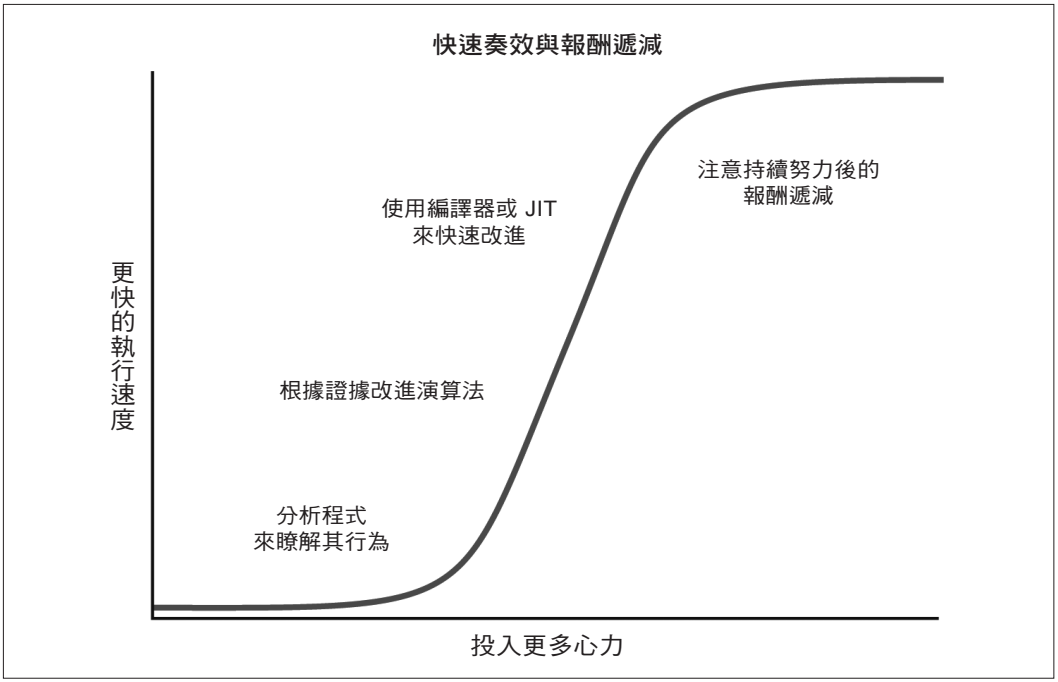


圖 8-1　做一些效能分析與編譯可帶來許多回報，但繼續努力換來的收獲往往開始遞減

雖然這些經驗來自調查新聞領域，但我希望這些原則能讓你應用在對你重要的專案中。

延伸閱讀：

- “Dollars to Megabits, You May Be Paying 400 Times as Much as Your Neighbor for Internet Service”（*https://oreil.ly/MOJmT*）
- “How We Uncovered Disparities in Internet Deals”（*https://oreil.ly/7JkcM*）
- “How We Uncovered Disparities in Internet Deals: GitHub Code and Data Repository”（*https://oreil.ly/ke-ub*）

## 來自網路再保險領域的經驗

***James Poynter***
***linkedin.com/in/james-poynter-0b295375***

James Poynter 是再保險經紀公司 Gallagher Re 的全球資料科學主管。他長期運用資料與技術來解決商業保險與再保險領域的風險管理及承保難題，並累積了豐富的成果。

身為資料科學家，看到高效能時，我會從資料流的角度來思考 —— 我想到一道快速、順暢、高效率的資料流，在機器學習與資料作業線系統中穿梭。每個機器學習（ML）系統都由一系列步驟組成，而每個步驟都包含輸入、轉換與輸出，其中 ML 模型的訓練與推理只是另一個轉換步驟（儘管是複雜的一種）。

若要讓資料科學與機器學習系統真正產生商業價值，就必須將資料傳入，並讓系統輸出高品質且可付諸行動的見解與資訊。在商業脈絡下，高效能的系統可以透過更好的決策與交易，迅速地實現這個具經濟效益的目標。

本節要分享一些來自保險分析領域的經驗，並提供幾個實務案例，展示我個人認為有助於提升效能的原則，它們都是我在（再）保險領域擔任資料科學團隊成員與管理者時的工作經驗。

我挑選了一些與 Python 有關、但大多偏向技術面的經驗，並以我近期在網路再保險領域中，開發預測風險分數與核保工具的工作為例。這些經驗適合許多領域與產業，但對金融領域的資料科學家來說應該特別有幫助。

## 清楚瞭解問題與交付需求

如果你只讀這一段，我最重要的教訓就是：若要追求高效能，首先你必須從第一原理（first principles）出發，徹底瞭解你的需求。無論你正在開發什麼應用程式，需求都必須是為了達成目標而**真正需要**的東西。在你寫下第一行程式之前，務必清楚地知道你需要做什麼，以及為什麼要這麼做。

要勇於發問、挑戰現狀，回到第一原理；能刪就刪，能簡化就簡化，能精簡就精簡——在實務上，高效能往往來自你沒有寫出什麼程式碼，而不是你寫了什麼程式碼。習慣持續深入理解問題，根據目標精煉出核心需求，並記錄、分享、有建設性地挑戰這些需求。

這個第一原理是我自己經常需要重新檢視並強化的。人很容易解錯問題，或做出僅帶來極少價值甚至毫無價值的功能、優化與自動化。把優質的需求視為原則與習慣，在你的工作與專案流程中貫徹實行。

## 不要低估領域知識的重要性

入門的資料科學課程通常把資料科學定義成「數學和統計」、「電腦程式設計」，以及「領域知識」三者的交集。保險業當然不是最吸引資料科學家的產業，我經常發現初階的資料科學家不太願意花時間培養領域知識，而是傾向鑽研演算法與程式設計技巧，因為他們認為這些技能更容易轉移至其他領域。

然而，領域知識可以確保你用正確的方法解決正確的問題。在現實的商業環境中，高效能往往取決於團隊是否掌握某些關鍵資訊。在商業脈絡下，培養領域知識與深入瞭解公司的核心產品很有價值，這能讓這個角色更有深度，經常帶來新的機會，也可以打破溝通障礙。

以下是與後續內容有關的領域知識：

- 再保險是賣給保險公司的保險，能讓保險公司將部分風險轉移給其他再保險公司。再保險經紀人提供風險分析、風險轉移方案設計、市場配置（market placement），以及合約管理服務來促成這一過程。身為一家全球再保險經紀公司 Gallagher Re 的