推薦序

圖（graph）[譯注 1] 無所不在：資訊不是孤立的，而是互聯的。尤其是在智慧代理系統的時代，要讓你的 LLM 以可信的「脈絡化事實」作為基礎，你必須擁有一套自底層便為高度連結的資訊而組建、同時可靠且高效能的全端資料庫引擎。知識圖譜（knowledge graph）是企業的數位分身，能讓你詢問和回答更全面的問題，並挖掘隱藏在資料深層的見解。GraphRAG（*http://graphrag.com*）提供進階的檢索增強生成（retrieval augmented generation，RAG）模式，使 LLM 的輸出更容易理解、更符合脈絡。

我和圖的緣分始於 1990 年代，我為多人線上文字冒險遊戲開發用戶端工具時，意外重現了 Dijkstra 最短路徑演算法。之後的 2008 年，我在波羅的海的一場技術郵輪會議上，遇見 Neo4j 的共同創辦人 Emil Eifrem。當時我正在開發零售應用程式，一聽到 Neo4j 這個名字，就立刻被它深深吸引，希望深入瞭解圖模型與查詢指令（query）[譯注 2] 在複雜的資料階層之中的應用。我開始為 Neo4j 資料庫開發開源整合方案（當時它只是個小型 Java 函式庫，故名為「4j」），並於 2010 年加入這家小規模的瑞典新創公司，成為第 10 位員工。我參與平台的各個部分，貢獻範圍包含核心、Cypher（全球最優秀的查詢語言），甚至資料匯入與應用開發整合程式庫（如 Spring Data Neo4j 與 GraphQL）。

譯注 1　本書中的 graph 是由邊和節點構成的資料結構，不是幾何形式構成的圖形或圖表，雖然也可以譯為「圖形」，但為了避免歧義，本書皆譯為「圖」或「圖譜」。

譯注 2　query 一般譯為查詢，為了避免與動詞的「查詢」混淆，本書譯為「查詢指令」，有時譯為「指令」，畢竟 query 的功能不是只有查詢。

十五年後，Neo4j 已成為瑞典的「獨角獸」企業之一，服務了全球數千家大型顧客，並作為一個在生成式 AI 革命浪潮中，扮演關鍵角色的圖資料平台。巧合的是，這正是我目前擔任產品創新與開發者體驗副總裁的核心工作。

如果你正在尋找由兩位頂尖專家撰寫、深入探討 Neo4j 的著作，那就是這本書了。作者 Luanne Misquitta 與 Christophe Willemsen 是我的多年好友，他們在 Neo4j 圖領域耕耘已久（幾乎與我同期），曾參與多個開源專案與眾多顧客專案，已經是資深的官方 Neo4j Ninja，回答過上千個問題，並透過部落格、演講、訓練課程推廣圖技術。在 Neo4j 的旅程中，我們在不同層面上並肩前行：我負責開發資料庫平台，Luanne 與 Chris 則身兼顧客、使用者、貢獻者的角色。當我帶領 Developer Relations、Neo4j Labs、以及後來的 Innovation and Developer Experience in Product Management 時，都曾經和他們密切合作，我也是他們的朋友與導師。我一直欣賞他們的好奇心、才華，以及在艱困的正式環境中，拆解複雜問題並加以解決的卓越能力。

他們兩人都熱衷於圖技術，證明了圖無所不在，無論是在人與人的互動之間（無論好壞）、在科學研究裡、在 IT 基礎架構的安全與依賴關係中、在全球經濟的供應鏈中、在數位人文學領域裡，以及在知識儲存庫中，都能看到圖的存在。

我在 2010 年認識 Luanne，當時她從事圖技能管理，並以名為 Flavorwocky 的食譜推薦 app 贏得我們的 Heroku Neo4j 錦標賽。自那以後，Luanne 成為 Neo4j 使用者社群的重要成員，持續撰寫文章、發表演講，與開設培訓課程。

Chris 則是在協助比利時海軍開發軟體時，加入開放原始碼的圖專案。初次見面時，他求知若渴的態度，以及具批判性的深刻回饋讓我印象深刻。他的第一個開源專案是 Neoxygen Graphgen，這個工具利用「文字化的圖模式敘述」來生成圖模型，目的不是用來查詢（querying），而是用來指定資料。之後，他開發並維護 neo4j-PHP 驅動程式多年，並推動 Neo4j 與多種資料技術的整合，包括 Camel、Kafka、Elasticsearch 與資料湖。

後來，他們終於全職投入自己熱愛的圖技術，Chris 加入 Neo4j 顧問公司 GraphAware 擔任 CTO、Luanne 則擔任工程副總。他們不僅每天都與嚴格的顧客及專案合作，還設計、建構並營運了 Hume。Hume 是一種完整的圖匯入、探索與調查平台，具備許多令人印象深刻的新功能，包括複雜圖分析、虛擬模式、用來匯入大量資料的模型 mapping，以及智慧 AI 整合…等。本書將分享他們多年來累積的實戰技巧與寶貴經驗。

本書將帶領你展開圖之旅，跟隨虛構音樂公司 ElectricHarmony 的團隊，從設計、開發、調校一個圖推薦引擎，一直到讓它正式上線。書中除了介紹圖建模、查詢、匯入真實資料的實際應用外，也展示常見的陷阱與正確的解決方法。隨著團隊開發週期的推進，讀者將逐步瞭解圖技術的強大潛力與應用，並學會在各個開發階段做出關鍵決策。

本書將介紹最新的 Neo4j 5 LTS 版本，作為你在正式部署時的穩固基礎，同時也暗示了目前的開發版即將推出的功能與能力。它也讓你為未來做好準備——包括更大規模的分片圖、無伺服器圖運算、與大型資料平台的緊密整合，以及運用圖的 AI 平台。

希望你能像我一樣喜歡這本書，並成為一位圖愛好者。你將在此學到許多別處學不到的知識。

祝你享受圖世界的樂趣！

— *Michael Hunger*
Neo4j 產品創新副總裁
《*DuckDB in Action*》與《*GraphRAG*：權威指南》作者
德國，德勒斯登，2025 年 4 月

前言

圖資料庫正驅動著全球大量企業的關鍵應用領域，範圍涵蓋推薦引擎、詐欺偵測、供應鏈優化、知識圖譜…等多種場景。Neo4j 作為這個領域的先驅與當今的龍頭圖平台，在這場演變中，扮演著關鍵角色。越來越多組織採用圖來解決問題，從互聯的資料中挖掘更深層的見解。然而，對於許多初次進入圖世界的團隊而言，從理解概念到生產部署的旅程仍充滿種種挑戰。

無論你想提升 Cypher 查詢效能、設計圖模型以支援多樣應用場景，或滿足企業級安全要求，本書都是你邁向正式部署的最佳夥伴。本書收錄了一系列實用的教學精華，幫助你運用 Neo4j 解決真實世界的挑戰。這些內容都基於成功的 Neo4j 部署案例，經過實戰的驗證。

此外，生成式 AI 的快速崛起推動了知識圖譜的廣泛應用，大眾從未如此迫切需要實作指南。我們將成為你的嚮導，從快速的概念驗證開始，協助你穩健前進，最終打造完整的生產系統。我們將帶你探索常見陷阱、討論不同方法間的取捨，並協助你打造能夠滿足現代企業架構需求的穩定解決方案。

看完本書後，你將明白原生圖資料庫的哪些特性讓它們成為最佳技術選擇（相較於多模型資料庫），並瞭解圖建模模式如何影響系統的記憶體使用率、CPU 負載、效能、速度，事業 SLA。你將在概念驗證階段做出帶來實際效益的決策，並在正式上線時，重新檢視與調整這些選擇。你也將掌握如何在企業級規模下執行 Neo4j，包括備份與日誌的設定，以及叢集部署的規劃。

我們為什麼寫這本書

圖資料庫這個領域相對年輕。作為這個領域最早的實踐者之一，我們已經和使用圖技術（特別是 Neo4j）的組織與個人合作長達數十年之久。我們始終保持實作精神，訓練大中小型團隊、評估圖應用場景、設計與審查圖架構、舉辦建模工作坊，並優化查詢指令效能。我們最初提供關於圖的顧問服務，後來成為 GraphAware 團隊的核心成員，參與構思並打造了 Hume 這一個連結資料分析平台。這段經驗彌足珍貴。由於圖的特性，每一個挑戰都不盡相同，總有新知識需要學習。我們認為是時候將這些經驗分享給希望在企業架構中導入 Neo4j，並將之正式上線的所有人。

我們相信，學習圖的理論知識最有效的方法，就是實際開發圖解決方案。我們觀察到，儘管目前已有大量文件可供參考，許多組織仍反覆犯下相同的錯誤和踏入相同的陷阱，所以我們偏好透過試誤法來分享知識，因此，在整本書中，我們會先引導你嘗試看似自然的解法，再解釋為什麼不該那樣做。這種教法可將概念深植心中，讓你真正理解我們的建議背後的原因，不會重蹈覆轍。

祝福你的圖技術之旅一切順利，更重要的是，希望你也能體驗，我們在處理圖時，體驗到的那份興奮與熱情。

這本書是否適合你？

先說一下這本書「不適合」或「還不適合」的人：如果你剛接觸圖技術與 Neo4j，我們建議你先完成初階課程，或實際做過幾個應用案例之後，再回來閱讀本書。GraphAcademy（*https://graphacademy.neo4j.com/*）是絕佳起點。

本書是為中高階的圖資料庫與 Neo4j 使用者而寫的。如果你是資料工程師或資深開發者，你應該已經知道如何將資料匯入圖中，並使用 Cypher 來查詢了。如果你是資料科學家，應該已嘗試過幾種圖資料科學演算法。對架構師、首席工程師與運維工程師而言，你應該曾於非生產或概念驗證環境中，整合圖資料庫，並能夠設定、管理，與監視資料庫。

總之，如果你正在使用 Neo4j，無論你是在正式環境中面臨問題，還是正在邁向正式部署的旅程上，這本書就是為你而寫的。

如何閱讀本書

我們建議從第 1 章開始閱讀，因為這一章將為你建立其餘內容的背景知識，特別是其中的範例。第 3、4、5 章密切相關，適合連續閱讀。其他各章可依序閱讀，逐步在前幾章打下的基礎之上學習新知。這幾章也可以當成參考，單獨查閱。

第 1 章「如何在五天內，從圖獲得價值？」回顧原生圖資料庫的重要性，並教你如何在一週內建立概念驗證。你將接觸各個面向：建模、資料匯入、撰寫 Cypher 查詢，本章也介紹一個簡單但實用的推薦查詢。

第 2 章「匯入更多資料（多很多）」將使用遠多於概念驗證階段的資料集，示範如何大規模匯入資料。你將進一步鞏固關於資料庫交易、記憶體管理，與平行寫入的知識。

第 3 章「檢討建模決策」將分析第 1 章做出來的部分建模決策，帶你瞭解它們的優缺點，以及在建模時應考慮的其他因素。檢討圖模型是經常在實際專案中執行的程序，本章將協助你從概念驗證模型邁向成熟模型。

第 4 章「模型建立與重構模式」整理多種建模模式及其最佳應用情境，並教你如何重構模型與圖。

第 5 章「查詢指令分析與調整」將深入探討該如何規劃查詢指令、瓶頸在哪裡、如何使用查詢分析工具，以及如何大幅提升 Cypher 查詢的效能。

第 6 章「保護你的資料庫」全面介紹在保護 Neo4j 資料庫時，應考慮的各個層面，從身分驗證與授權，一直到防止竄改資料與權限，本章提供一份可實際操作的完整安全檢查清單。

第 7 章「搜尋」教你如何儲存與查詢文字資料，以獲得相關的搜尋結果。

第 8 章「進階圖模式」整理在使用 Neo4j 一段時間之後，必然會遇到的模式與應用場景。本章將介紹子查詢、整併實體並為之建模、帶有數量條件的路徑模式，並協助你將安全性資料建模提升到更高層次

第 9 章「備份與還原」介紹在正式環境中操作 Neo4j 的必備技能。你將學習各種備份、還原的方式，以及如何設計有效的備份策略。

第 10 章「叢集與分片」為你和不斷成長的圖系統做好擴展的準備。你將瞭解為了提升妥善性而使用的叢集機制，以及用來執行分片（sharding）與聯邦式查詢的解決方案——複合資料庫。

第 11 章「觀察機制」帶你透過日誌與監視功能來建立穩健的策略，確保你的圖系統健康運作，並且可以支援關鍵任務。

第 12 章「實用圖資料科學」介紹圖資料科學程式庫，以及如何運用裡面的演算法從資料中挖掘知見。

第 13 章「生成式 AI 與圖的未來發展」闡述知識圖譜與 LLM 之間的共生關係，並說明為何知識圖譜既是現在與未來的核心。

本書編排慣例

本書使用下列的編排方式：

斜體字（*Italic*）

代表新術語、URL、email 地址、檔名，與副檔名。中文以楷體表示。

定寬字（`Constant width`）

用於程式碼清單，以及段落中提及的程式元素，如變數、函式名稱、資料庫、資料型別、環境變數、語句與關鍵字。

定寬粗體字（**`Constant width bold`**）

表示由使用者按照字面輸入的命令或文字。

定寬斜體字（*`Constant width italic`*）

表示用使用者輸入的值，或上下文決定的值來取代的文字。

這個符號代表提示或建議。

這個符號代表一般附註。

這個符號代表警告或注意事項。

第二章

匯入更多資料（多很多）

你與 ElectricHarmony 團隊開始面臨匯入大量資料的挑戰。大量的資料更能夠代表整體使用者，並推薦曲目給使用者。

你試著使用第 1 章的資料匯入方法，但當你將曲目資料集的規模從十萬筆增加到一百萬筆時，這個方法非常緩慢，經常導致令人畏懼的「死亡旋轉游標」，讓你開始懷疑自己選用的資料庫是否適合大規模情境。

你的擔心是合理的，因為你會面臨團隊與利害關係人提出的關鍵問題，例如：

- 系統匯入資料的速度能不能跟上事業產生資料的速度？
- 系統能不能近乎即時推薦曲目，並跟上其他系統產生的資料？
- 發生災難時，需要多久才能復原？

本章將說明如何匯入大型資料集，以便回答這些問題。

在這趟學習旅程中，我們會先介紹一個容易理解的資料庫管理系統的內部機制，包括交易（transaction）與記憶體管理。接著，你將學習如何優化第 1 章的 `LOAD CSV` 命令，然後進入更接近實際正式環境的情境，使用你偏好的程式語言來撰寫自動化程式以匯入資料。你還會嘗試各種不同的鎖定策略，以瞭解在什麼情況下，可以有效率地平行匯入資料，而不造成負面影響。我們最後會介紹離線資料匯入策略。

資料庫交易

交易就是將一組操作視為一個整體，使全部的操作若非全部成功，就是全部失敗，以確保資料的完整性（integrity）。資料庫的限制條件（例如唯一性與型態限制）可以用來實

施一致性與準確性的規則。交易可確保所有操作都遵守這些限制條件，但許多檢查（例如唯一性檢查）通常會延後至交易結束（提交時）才執行，這種設計，可讓更新過程更有彈性，例如將資料從一種狀態轉換為另一種狀態時，可以暫時違反某項限制條件。如果有任何操作違反限制條件，整個交易便會失敗並復原，確保不會只有部分變更生效，從而避免資料不一致。

你已經在第 1 章用過資料庫限制條件（CONSTRAINT）了。舉例來說，你建立了一個 CONSTRAINT 來確保 HAS_TRACK 關係的 position 屬性是整數：

```
CREATE CONSTRAINT has_track_position_integer
FOR ()-[r:HAS_TRACK]-()
REQUIRE r.position IS TYPED INTEGER
```

試著寫入不符合條件的資料會怎樣？例如下面這個 Cypher 查詢指令？

```
CREATE (n:Playlist)-[:HAS_TRACK {position: 'some string'}]->(track);
```

這個指令無法執行，系統會顯示如圖 2-1 所示的錯誤訊息。用資料庫術語來說，意思就是你的交易因為違反限制條件而被中止。

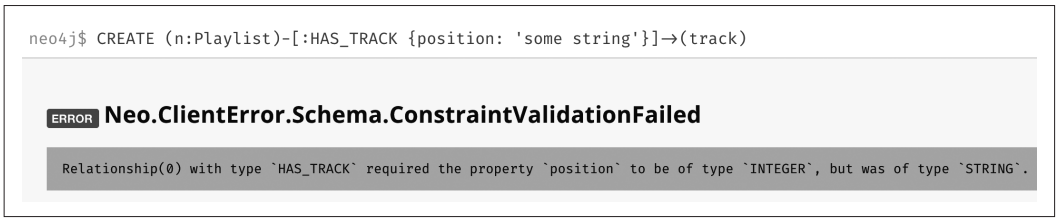

圖 2-1　試著執行不符合資料庫限制條件的查詢時產生的錯誤訊息

但等等，你甚至沒有使用交易（*transaction*）一詞——事實上，到目前為止，你根本沒有在指令內看過它！ Neo4j 瀏覽器會私下使用交易，多數的 Neo4j 驅動程式也提供一些方法，讓你不必自己管理交易。你只要知道，在 Neo4j 中的所有操作都有交易性質即可。

那麼，底層究竟做了什麼事情？系統會依序執行以下步驟：

1. Neo4j 伺服器開啟一個交易。
2. 它在資料庫裡執行變更（例如建立資料）。
3. 它提交交易（套用變更）或拒絕交易（捨棄變更）。

這些步驟符合資料庫系統（包括關聯式與非關聯式）為了確保安全而必須具備的交易特性，這些特性通常稱為 ACID 特性，各個字母代表：

原子性（*Atomicity*）

這個特性將所有操作視為一個整體。你不會希望資料庫裡面有半成品：要嘛全部成功，要嘛全部失敗。

一致性（*Consistency*）

這個特性確保交易能讓資料庫從一個有效狀態轉換到另一個有效狀態，維持資料的完整性。

隔離性（*Isolation*）

交易是獨立執行的；即使有多個交易同時執行，它們也無法看見彼此的中間狀態。這正是 Neo4j 的運作方式，其預設的隔離等級（isolation level）為 `READ COMMITTED`。

持久性（*Durability*）

一旦交易成功提交，它就會被永久記錄，即使系統停電或崩潰也不會遺失。

交易最終會將資料寫入磁碟上的交易日誌，以確保持久性，如圖 2-2 所示。在本章中，你只要知道磁碟的速度（每秒輸入 / 輸出操作次數，即 IOPS）很重要：越快越好。第 9 章會詳細說明寫入路徑，包括交易日誌。

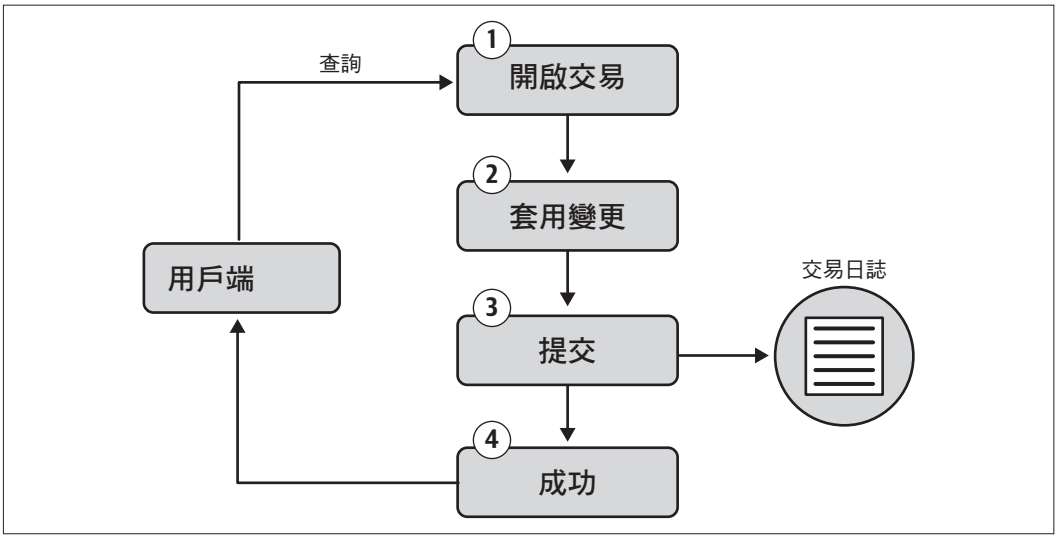

圖 2-2　成功交易的流程

交易的核心目的，是確保資料的新增與刪除能夠安全且一致地執行。然而，這種方法也有其侷限，下一節將進一步探討。

~~Beat~~ Heap Box

Neo4j 是用 Java 來開發，並在 Java 虛擬機（JVM）上執行的，所以它有 JVM 的一項重要元素——heap（堆）。*heap* 是一塊專用的記憶體區域，用於儲存 Java 物件和應用程式執行過程中需要保留的所有工作資料。當 Neo4j 產生新的實體（例如節點）時，它可以將交易狀態儲存在 heap 內或 heap 外。然而，建立交易日誌的程序會在 heap 中產生交易日誌命令。

heap 由 JVM 管理，JVM 會自動為新物件分配空間，並回收再也用不到的物件的空間，這個回收程序稱為資源回收（*garbage collection*），它可以有效率地重複利用記憶體。圖 2-3 是一個為 heap 保留的有限空間，已配置的物件會占用其中的部分區域。

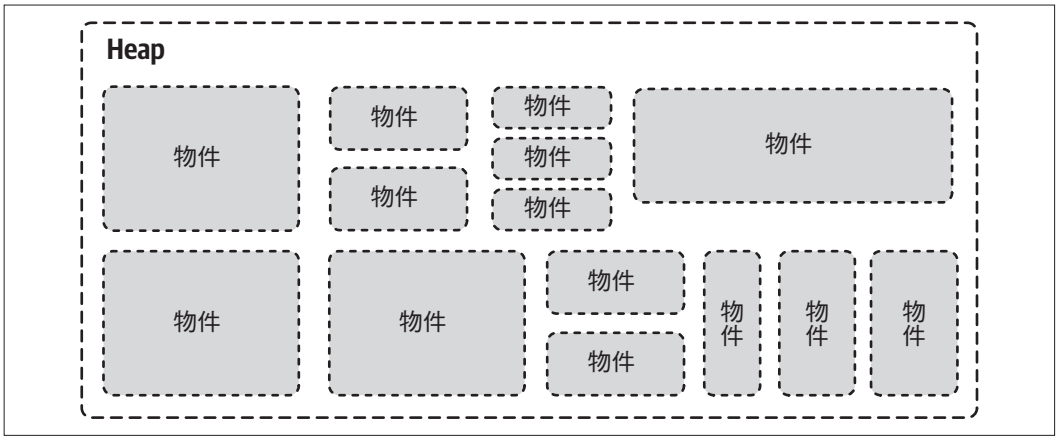

圖 2-3　heap，物件占用了其中的部分空間

heap 不僅用來將資料寫入資料庫，在交易的讀取期間，它還會在這個記憶體區域中暫時儲存資訊，例如查詢指令的執行計畫、交易狀態、查詢的中間狀態，以及查詢的結果。

就像實際的箱子一樣，heap 的容量是有限的。切記，heap 是記憶體的一塊區域，所以它的大小本質上會被機器或 Neo4j 所在伺服器的可用記憶體總量限制。實際上，使用 Neo4j 來建構的應用程式往往會同時執行多個交易，這些交易會互相競爭 heap 中的記憶體空間。

當你使用 LOAD CSV 命令來匯入資料（如同第 1 章的做法）時，每建立一個節點、屬性或關係，都會消耗 heap 的空間（如圖 2-4 所示）。因此，heap 的大小限制了你在單一交易中，可以匯入的 CSV 資料列數量，該數量不能耗盡 heap 的空間。

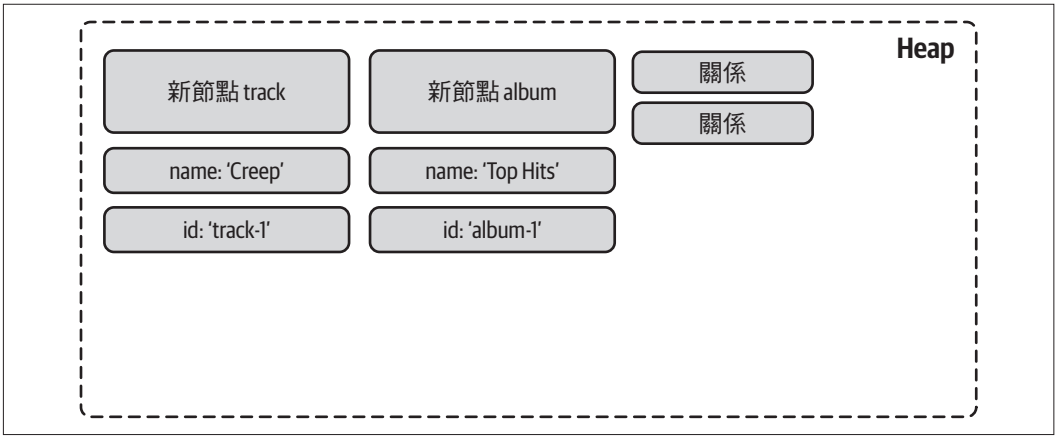

圖 2-4　LOAD CSV 子句建立的物件，會在 heap 中占用空間

試著匯入包含 100 萬列資料的 CSV 檔案可能會超出 heap 的記憶體容量上限，導致交易失敗。根據你匯入資料的方式，這個問題有多種方法可以解決。雖然 LOAD CSV 確實是個方便的選項，但本章也會探討如何使用你喜愛的程式語言，直接從你的應用程式匯入資料。

在 Neo4j 伺服器中，用來設定 heap 最大記憶體配置空間的參數是 `server.memory.heap.max_size`。在 Docker Compose 範例中，heap 被限制為 512 MB。你可以在操作手冊中找到組態設定的參考資料（*https://oreil.ly/-r5TJ*）。

CALL IN TRANSACTIONS 操作

Cypher 可讓你指定在單一交易中，要提交多少列資料。如果有一個包含 100,000 列資料的 CSV 檔，那麼設定每 10,000 列資料提交一次會產生 10 個獨立的交易，每一個交易處理 10,000 列資料，如圖 2-5 所示。

當操作不需要百分之百的原子性時，才能將它拆成多個交易。拆成多個交易時，每一個交易會被獨立提交，所以，如果其中的一個交易失敗，可能只有部分的操作會被更新。大量匯入資料與大規模轉換資料都是典型的應用場景，這些情況不一定需要百分之百的原子性。

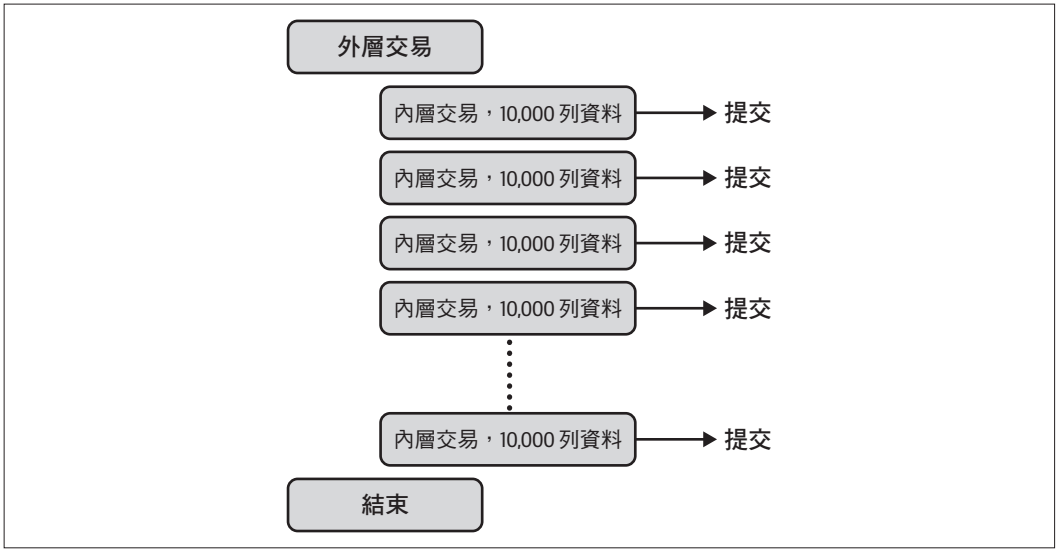

圖 2-5　逐漸提交 CSV 的資料，每次 10,000 列

要在交易中逐漸提交資料列，就要將陳述式包在子查詢中，並在子查詢之後使用 IN TRANSACTIONS：

```
LOAD CSV WITH HEADERS FROM "file:///mydata.csv" AS row
CALL(row) {
  MERGE ...
  MERGE ...
} IN TRANSACTIONS OF 10_000 ROWS;
// 數字中的底線 _ 只是為了幫助閱讀，
// 在實際執行時，會使用 10000 這個值
```

如果你沒有指定每一個內部交易的資料列數，系統預設使用 1,000 列資料。

做好實際操作這項技術的準備了嗎？接下來，你會明確地把 Neo4j 可用的 heap 記憶體減少至 128MB，建立一個新的測試資料庫，並匯入你在第 1 章用過的範例檔案。

在本書 GitHub 版本庫的 Docker Compose 範例中，將下面這行程式的 # 字元移除，讓它成為可執行的程式碼：

```
NEO4J_server_memory_heap_max__size: "128M"
```

接著，在終端機中執行 `docker compose up -d` 以重新啟動 Neo4j 容器。開啟 Neo4j 瀏覽器並執行下列指令，以建立名為 `testload` 的新資料庫、切換至該資料庫，並為曲目建立限制條件：

```
//001-create-database.cypher
CREATE DATABASE testload WAIT
:use testload
CREATE CONSTRAINT FOR (n:Track) REQUIRE n.id IS NODE KEY;
```

使用 `:use testload` 來切換到新資料庫，然後建立限制條件：

```
//002-create-constraint.cypher
CREATE CONSTRAINT FOR (n:Track) REQUIRE n.id IS NODE KEY;
```

現在匯入我們在第 1 章結尾用過的那個曲目範例 CSV 檔案。這次只會匯入曲目資料：

```
//003-load-tracks.cypher
LOAD CSV WITH HEADERS FROM "file:///medium/sample_tracks_medium.csv" AS row
MERGE (track:Track {id: row.track_id})
SET track.uri = row.track_uri,
track.name = row.track_name;
```

幾秒鐘後，你應該會看到如圖 2-6 所示的錯誤訊息，它指出你試著執行的交易使用了超過系統分配給 Neo4j 的記憶體上限。

圖 2-6　在交易使用的記憶體超出上限時出現的錯誤訊息也稱為 `OutOfMemory` 錯誤

CSV 檔案大約有 100,000 列資料。我們每處理 10,000 列資料後提交一次，以便在每次迭代後，釋放它們占用的記憶體：

```
//004-call-in-transactions.cypher
:auto // 如果你在 Neo4j 瀏覽器中執行這個查詢，就必須使用它
LOAD CSV WITH HEADERS FROM "file:///medium/sample_tracks_medium.csv" AS row
CALL(row) {
    MERGE (track:Track {id: row.track_id})
```

```
    SET track.uri = row.track_uri,
    track.name = row.track_name
} IN TRANSACTIONS OF 10_000 ROWS;
```

`:auto` 是 Neo4j 瀏覽器的專用命令，用來讓瀏覽器以自動提交交易的方式送出 Cypher 查詢。我們通常不建議使用自動提交的交易，因為它不支援在失敗時自動重試。然而，有一些查詢（例如 `CALL { … } IN TRANSACTIONS`）必須以這種方式執行，才能在 Neo4j 瀏覽器中正常運作。

可喜可賀！你已經大幅減少記憶體使用量，並將資料匯入 Neo4j 資料庫了。

`CALL IN TRANSACTIONS` 並非只能搭配 `LOAD CSV` 使用。你也可以在其他大規模的更新操作中使用它，例如以較小的增量，刪除節點密度極高的關係，以確保可控、穩定的記憶體使用量。

在瀏覽器中使用的 `LOAD CSV` 不適合在正式環境中用來將匯入操作自動化。

下一節將詳細說明如何在 Python 用戶端應用程式中執行相同的操作。

動手試試：從用戶端應用程式匯入資料

本節要教你如何有效率地從用戶端應用程式匯入資料。

Cypher 查詢參數

Cypher 的查詢參數可讓你使用相同的查詢字串和不同的輸入值來查詢或修改資料，這可以讓 Neo4j 重複使用快取內的執行計畫，避免重複執行解析與規劃，進而提升效率。對應用程式開發者而言，查詢參數可以讓他們有效率地使用不同的輸入來執行多筆查詢。第 4 章將深入探討 Neo4j 的查詢分析過程，第 6 章會說明如何使用查詢參數來防範惡意注入攻擊。

在具有名稱的參數占位符開頭應附上 `$`：

```
CREATE (track:Track {id: $id})
RETURN track
```

你可以在執行指令之前，宣告一些參數與值，我們試著在 `testload` 資料庫內建立一個 Track：

```
:params {id: 'test-track', name:'Thunderstruck'};
CREATE (track:Track {id: $id}) SET track.name = $name RETURN track;
```

Neo4j 驅動程式

驅動程式（*driver*）是一種軟體元件，可讓程式與資料庫互相通訊。在 Neo4j 的情境中，驅動程式能協助你和 Neo4j 資料庫互動，開發者可使用 Java、Python、JavaScript 等多種程式語言來查詢與操作資料。這些驅動程式為應用程式提供一個介面，讓應用程式能連接 Neo4j 資料庫並與它互動。在執行上一段程式時，從 Neo4j 驅動程式的角度來看，我們是一併傳入查詢指令和一個參數字典（dictionary）。

注意：Neo4j 手冊（*https://oreil.ly/BVJmo*）為多種語言寫了一個驅動程式指南。GraphAcademy 也有一些課程（*https://graphacademy.neo4j.com/categories/development*），介紹如何使用 Neo4j 驅動程式來開發應用程式。

以下是一起使用 Python 版 Neo4j 驅動程式與參數的範例：

```
001-cypher-parameters.py
from neo4j import GraphDatabase
from neo4j import Result

URI = 'bolt://localhost:7687'
AUTH = ('neo4j', 'password')

query = '''
    CREATE (track:Track {id: $id})
    SET track.name = $name
    RETURN track
    '''

def create_track(driver, parameters):
    record = driver.execute_query(
        query,
        parameters,
        database_='testload',
        result_transformer_=Result.single
        )
    return record['track']['name']

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    t1 = create_track(driver, {'id': 'track-01', 'name': 'Sin City'})
    t2 = create_track(driver, {'id': 'track-02', 'name': 'Creep'})
```

注意：GitHub 版本庫也有如何執行這些範例程式的完整說明。

瞭解 Cypher 參數如何運作之後，我們可以繼續進入下一節了。

使用 UNWIND 來處理批次資料

Cypher 的 UNWIND 子句可將一個串列（值的集合）拆成個別的資料列。這可以讓你分別處理串列裡的每一個項目，所以 UNWIND 是幫助處理批次資料的強大工具：

```
UNWIND [1, 2, 3] AS x
RETURN x

//Result
1
2
3
```

一起使用 Cypher 參數與 UNWIND 子句是有效率地匯入資料的重要技巧。UNWIND 不僅能處理簡單的串列，還能將字典串列轉換為資料列（字典在 Neo4j 中稱為 *map*）：

```
UNWIND [
{id: 'track-7', name:'Start me up'},
{id: 'track-8', name:'I feel good'}
] AS trackInfo
CREATE (track:Track {
  id: trackInfo.id,
  name: trackInfo.name
})
RETURN track
```

Cypher 查詢參數可以使用 map 組成的串列，這種型態在從用戶端應用程式匯入資料時極為常見。本節將探討如何有效地使用這類參數。

首先，你要在用戶端蒐集一些物件，以建立一個臨時的串列。接著，使用 UNWIND 來執行查詢，並將這個臨時串列當成 Cypher 參數，傳入查詢指令。以下的程式示範如何一起應用這些概念。

Python：

```
#002-cypher-unwind-parameters.py
query = '''
    UNWIND $trackInfos AS trackInfo
    CREATE (track:Track {id: trackInfo.id})
    SET track.name = trackInfo.name
    RETURN track
    '''
```

```
def create_tracks(driver, tracks):
    records, _, _ = driver.execute_query(
        query,
        trackInfos=tracks,
        database_='testload'
        )
    for record in records:
        print(record['track']['name'])

with GraphDatabase.driver(URI, auth=AUTH) as driver:
  # 蒐集一定數量的曲目
    all_tracks = []
    all_tracks.append({'id': 'track-01', 'name': 'Sin City'})
    all_tracks.append({'id': 'track-02', 'name': 'Creep'})

    # 使用曲目集合來執行查詢
    create_tracks(driver, all_tracks)
```

為了脫離概念驗證階段，並處理真實規模的資料，接下來要用 Python 來匯入標準曲目 CSV 檔案。我們會將每一行資料轉換成一個字典，然後加入集合中，並作為 Cypher 查詢參數傳入。

我們先將所有紀錄存入同一個集合，然後試著重現先前使用 `LOAD CSV` 匯入整個檔案時遇到的 `OutOfMemoryError` 錯誤：

```
#003-read-csv-1.py
import csv
from neo4j import GraphDatabase

URI = 'bolt://localhost:7687'
AUTH = ('neo4j', 'password')

all_rows = []

with open('./files/sample_tracks_medium.csv', 'r') as file:
    reader = csv.DictReader(file)
    # 將檔案中的所有紀錄
    # 放入 all_rows 變數中
    for row in reader:
        all_rows.append(row)

query = '''
    UNWIND $trackInfos AS trackInfo
    CREATE (track:Track {id: trackInfo.id})
    SET track.name = trackInfo.name
    '''
```

```
with GraphDatabase.driver(URI, auth=AUTH) as driver:
    records, _, _ = driver.execute_query(
        query,
        trackInfos=all_rows,
        database_='testload'
    )
```

執行這個腳本會出現預期的錯誤：`java.lang.OutOfMemoryError: Java heap space`。

調整批次大小

將整個 CSV 檔案的全部紀錄都放入單一批次是有問題的，與使用 `LOAD CSV` 來匯入整個檔案一樣。解決方法是遍歷整個資料集，並定期提交較小的批次，以分段匯入資料。

以下程式示範如何使用上述的調整批次大小（batch sizing）的概念，邏輯如圖 2-7 所示：

```
#004-import-sized-batch.py
import csv
from neo4j import GraphDatabase

URI = 'bolt://localhost:7687'
AUTH = ('neo4j', 'password')
driver = GraphDatabase.driver(URI, auth=AUTH)

query = '''
    UNWIND $trackInfos AS trackInfo
    CREATE (track:Track {id: trackInfo.id})
    SET track.name = trackInfo.name
    '''

BATCH_SIZE = 10_000
current_batch = []

def commit_batch(current_batch):
    records, _, _ = driver.execute_query(
        query,
        trackInfos=current_batch,
        database_='testload'
    )
    # 將 current_batch 重設為空串列
    current_batch.clear()

def process_row(row, current_batch):
    current_batch.append(row)
```

```
        if len(current_batch) >= BATCH_SIZE:
            commit_batch(current_batch)

    with open('./files/sample_tracks_medium.csv', 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            process_row(row, current_batch)
        else:
            commit_batch(current_batch)
```

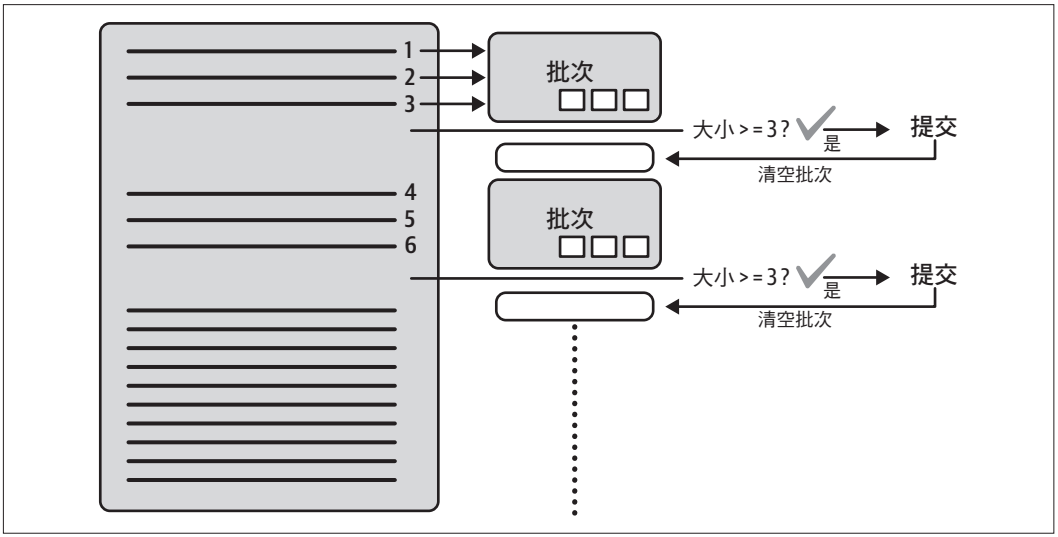

圖 2-7　分批流程

這段 Python 程式就是將資料匯入 Neo4j 時會使用的典型程式碼。在實際操作時，你也要留意程式針對 Neo4j 進行的交易操作。批次太小會限制寫入 Neo4j 的速度；太大可能造成 heap 記憶體被大量使用，進而影響其他正在執行的交易。

在處理 CSV 檔案時，你可能會想：「為什麼不直接使用 LOAD CSV 就好？」。這是個好問題。LOAD CSV 在許多情況下確實非常好用，尤其是它可以用來快速匯入資料，與設計簡單的作業流程。不過在真實世界的應用中，資料往往不會以如此乾淨且規則的格式出現。你經常需要處理動態輸入、條件邏輯、錯誤，以及監視系統，這些工作在通用程式語言中比較容易管理。即使 LOAD CSV 有時確實是適合的工具，瞭解分批與交易的運作方式仍然有其必要。

UNWIND 的特殊性

UNWIND 子句有一個特性，至今仍讓許多開發者困惑不已，往往要等到「啊哈」時刻才恍然大悟。但你正在閱讀這一節，所以不會這樣！

如前文所述，UNWIND 子句會將串列轉換成多列。然而，若串列為空，則不會產生任何資料列，後續的查詢也不會執行。以下這個 Cypher 查詢不會回傳任何結果，因為第二個 UNWIND 沒有產生任何資料列：

```
UNWIND [1,2,3] AS i
UNWIND [] AS x
RETURN i, x
```

隨著資料集越來越多樣化，你可能會使用比 CSV 資料列複雜許多的資料結構，例如 JSON 物件。

假設以下 JSON 物件是資料中的 User 物件結構。

```
{
    "id": "user-789",
    "podcasts": [],
    "playlists": [
        "playlist-12",
        "playlist-14"
    ]
}
```

你會下意識地按照資料結構的順序來撰寫 Cypher 查詢指令：

```
MERGE (u:User {id: $id})
WITH u
UNWIND $podcasts AS podcast
  MERGE (p:Podcast {id: podcast})
  MERGE (u)-[:SUBSCRIBED_TO]->(p)
WITH u
UNWIND $playlists AS playlist
  MERGE (pl:Playlist {id: playlist})
  MERGE (u)-[:HAS_PLAYLIST]->(pl)
```

使用上面的 JSON 物件作為 Cypher 參數來執行指令會建立使用者，但不會建立使用者與播放清單之間的關係，因為 $podcasts 之前的 UNWIND 並未產生任何資料列（rows）。為了解決這個問題，你可以將建立播放清單的指令包在一個專屬的子指令中：

```
MERGE (u:User {id: $id})
WITH u
CALL(u) {
    UNWIND $podcasts AS podcast
    MERGE (p:Podcast {id: podcast})
    MERGE (u)-[:SUBSCRIBED_TO]->(p)
    RETURN count(*) AS podcasts
}
CALL(u) {
    UNWIND $playlists AS playlist
    MERGE (pl:Playlist {id: playlist})
    MERGE (u)-[:HAS_PLAYLIST]->(pl)
    RETURN count(*) AS playlists
}
RETURN podcasts, playlists
```

注意：在兩個子指令裡面的 `RETURN count(*)` 是用來控制基數（cardinality）的。在 Cypher 中，基數指的是查詢指令在任一段處理的列數。在使用子指令時，管理基數非常重要，因為部分指令的結果可能會在後續操作中，無意間成倍放大列數，特別是在建立或合併節點與關係時。

你可以在 Neo4j 瀏覽器中模擬這兩個查詢指令的參數，並實驗它們的行為：

```
:params {id: "user-789", podcasts: [],
  playlists: ["playlist-12", "playlist-14"]}
```

善用 Cypher 查詢參數，並正確地設定批次大小，是快速匯入大量資料的關鍵（也能讓開發者更愉快）。

平行寫入

為了加快資料匯入速度，有時開發者會同時執行多個 Neo4j 指令，以平行寫入資料。這項操作聽起來很簡單，但它涉及多項複雜因素。本節將介紹有效執行平行指令的核心原則，並說明可能遇到的挑戰，協助你順利推進專案。這是一份實用指南，希望提升你的技術，讓你高效率且成功地平行匯入資料。

記憶體競用

如果有兩個交易同時試著使用 heap，則 heap 的容量必須足以接受這兩個交易同時運作。在圖 2-8 所示的情境中，如果第三個交易試著寫入與前兩個交易相同數量的資料，Neo4j 伺服器就會達到 heap 記憶體上限，並拒絕該交易，甚至更糟，導致伺服器關閉。

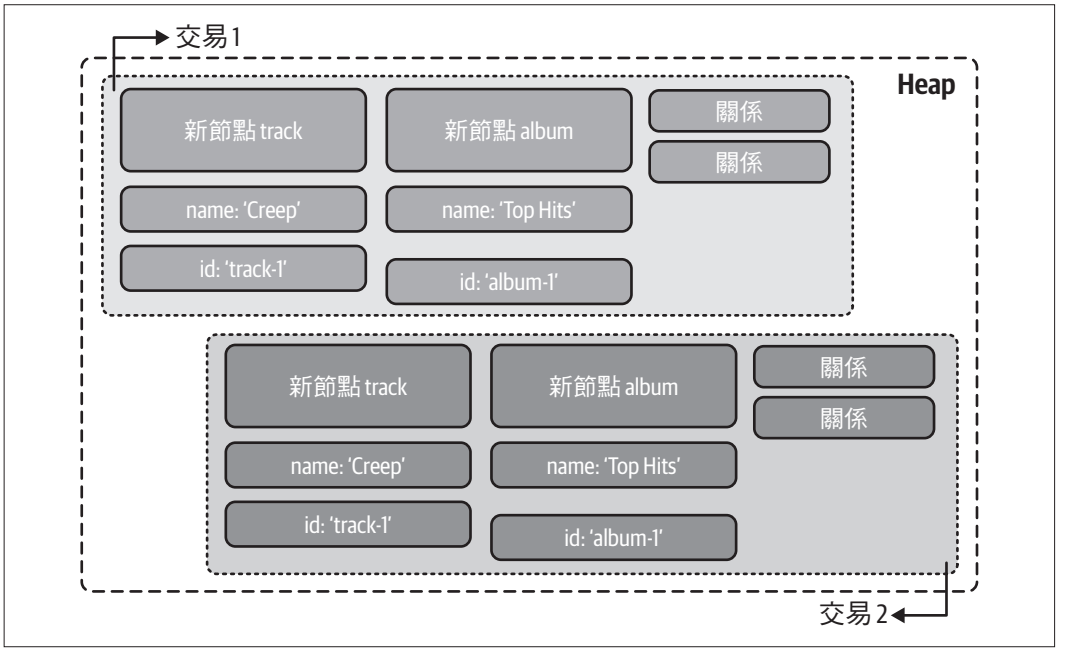

圖 2-8　多個交易的工作單元在 heap 中占用空間

務必確保 heap 的記憶體足以支持可能同時出現的交易工作量。別忘了，讀取型交易也會消耗記憶體空間。

你可以設定兩個關鍵參數來限制交易的記憶體使用量，使用 `db.memory.transaction.max` 來定義單一交易可使用的最大記憶體量，以及使用 `db.memory.transaction.total.max` 來限制資料庫中所有交易的總記憶體使用量。你也可以分析（profile）特定的 Cypher 查詢指令，來預估其記憶體消耗量，第 5 章會教你怎麼做。

鎖定機制

在執行寫入操作時，Neo4j 會取得鎖（acquires locks）來維持資料的一致性。如果你在兩個不同的交易中修改同一筆紀錄，其中一個交易必須等待另一個完成後才能提交。

我們用圖 2-9 來說明這個概念，其中，交易 1 刪除一個節點，交易 2 嘗試更新同一個節點的屬性。

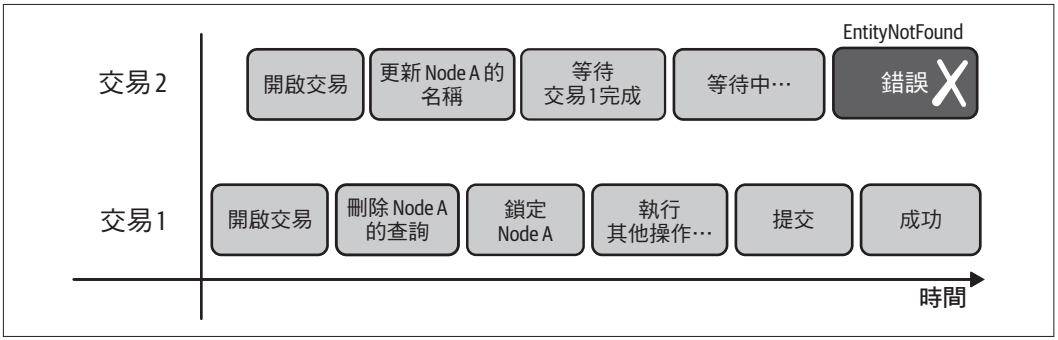

圖 2-9　兩個同時執行的交易試著修改同一個節點的情形

為了讓你更瞭解這個機制，本節將帶你實際操作，希望透過這些練習來提供很少人會探討的實務見解，讓你全面掌握這些概念。

首先，建立一個新資料庫：

```
//005-create-database-locking.cypher
CREATE DATABASE locking WAIT
```

接著使用 `:use locking` 來切換至該資料庫。

然後，觀察以下 Cypher 查詢指令。`apoc.util.sleep` 會暫停交易，到了指定時間（以毫秒為單位）後，再繼續執行。這個簡單的技巧在未來的測試情境中非常實用！

執行這個查詢指令會讓你的 Neo4j 瀏覽器顯示旋轉游標約 60 秒，等待交易提交：

```
//006-merge-wait-spinning.cypher
MERGE (n:Track {id: 1})
WITH n
// 交易將暫停 60 秒
CALL apoc.util.sleep(60000)
// 交易現在繼續執行
SET n.name = 'Creep'
RETURN n
```

Neo4j 程序是已經用 Java 寫好的函式，或使用者用 Java 撰寫的函式，其目的是擴充 Cypher 查詢指令。它們能在資料庫中直接執行進階操作，例如資料處理、自訂演算法，或系統整合。常見的程式庫 Awesome Procedures on Cypher（APOC）提供豐富的程序與函式，可用來匯入、匯出資料、執行圖演算法、管理結構定義…等。它的程序是用 Cypher 的 `CALL` 關鍵字來呼叫的，可加強 Neo4j 的彈性與功能。本書 GitHub 版本庫的 Docker Compose 環境已預設啟用 APOC。

同時更新節點與關係

程式的不同部分可能會同時更新圖，這會增加不同的程序同時修改相同的節點或關係的機率。本節將示範如何運用鎖定機制來控制這類操作，以及提供一些減少衝突發生的策略。

在這個實驗中，你將開啟兩個 Neo4j 瀏覽器分頁，模擬兩個同時運行的交易，其中一個將使用暫停交易的技巧。

在第一個瀏覽器分頁中，建立一個節點，讓兩個交易試著同時修改它：

```
//007-create-node-1.cypher
CREATE (t:Track {id: 1})
```

接著，在第一個 Neo4j 瀏覽器分頁中執行以下 Cypher 指令：

```
//008-tab-1-1.cypher
MATCH (t:Track {id: 1})
SET t.name = 'Creep'
WITH t
CALL apoc.util.sleep(60000)
RETURN t
```

然後，在第二個分頁中執行以下指令：

```
//009-tab-1-2.cypher
MATCH (t:Track {id: 1})
SET t.name = 'Creep from transaction 2'
RETURN t
```

你會發現，即使第二個指令沒有使用 *apoc.util.sleep* 程序，它仍會顯示旋轉游標 60 秒左右，代表該交易正在等待另一個交易完成，如圖 2-10 所示。你也可以觀察到，曲目的最終名稱是第二個交易設定的值。

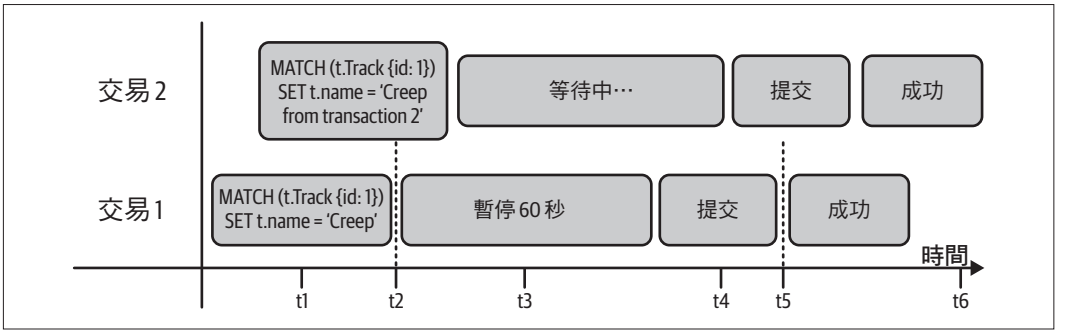

圖 2-10　一個交易必須等待另一個交易完成後，才能繼續執行

你可以使用 `SHOW TRANSACTIONS` 命令來檢查交易的狀態。在上述的兩個查詢都處於閒置狀態時，於 Neo4j 瀏覽器執行以下程式碼：

```
//010-transactions-info.cypher
SHOW TRANSACTIONS YIELD *
RETURN transactionId, status, currentQueryId, currentQuery,
resourceInformation.lockMode AS lockMode,
resourceInformation.resourceType AS lockOnResource
```

如圖 2-11 所示，第二個交易的狀態顯示，它被第一個交易塞住（blocked）了。

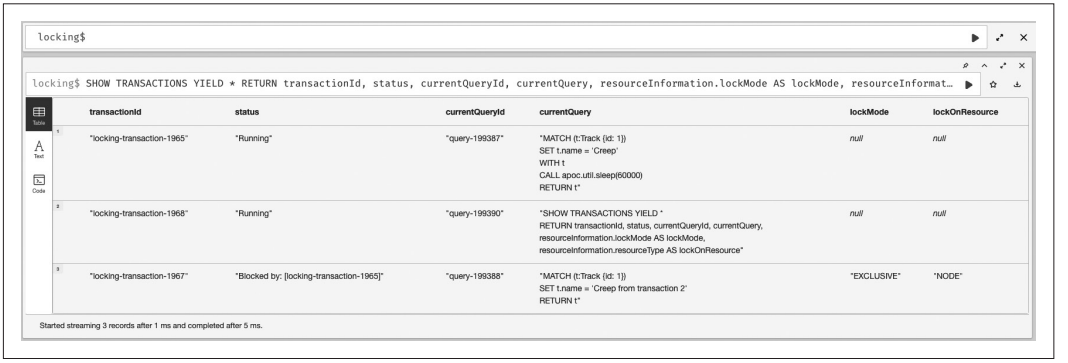

圖 2-11　一個交易被另一個交易塞住，必須等它完成後，才能繼續執行

為了盡量減少同時執行的匯入流程造成的衝突，我們必須設計能降低資源競用的匯入流程。做法之一是將匯入任務分配給多個並行的執行緒，讓每一個執行緒負責不同的資料部分。例如，你可以讓執行緒處理具有特定標籤的節點，或是處理代號的字母或數字屬於特定範圍的節點，確保兩個執行緒不會同時修改相同的節點。

多個交易同時試著存取相同的節點或關係可能造成鎖定衝突，導致系統壅塞。雖然像 LOAD CSV 這類的 Cypher 匯入陳述式能有效地處理大量資料，但若批次切分不當，或節點處理互相重疊，競用問題就會出現，特別是在大規模匯入作業中（見圖 2-12）。設計匯入流程來避免這類重疊，可以有效緩解問題。

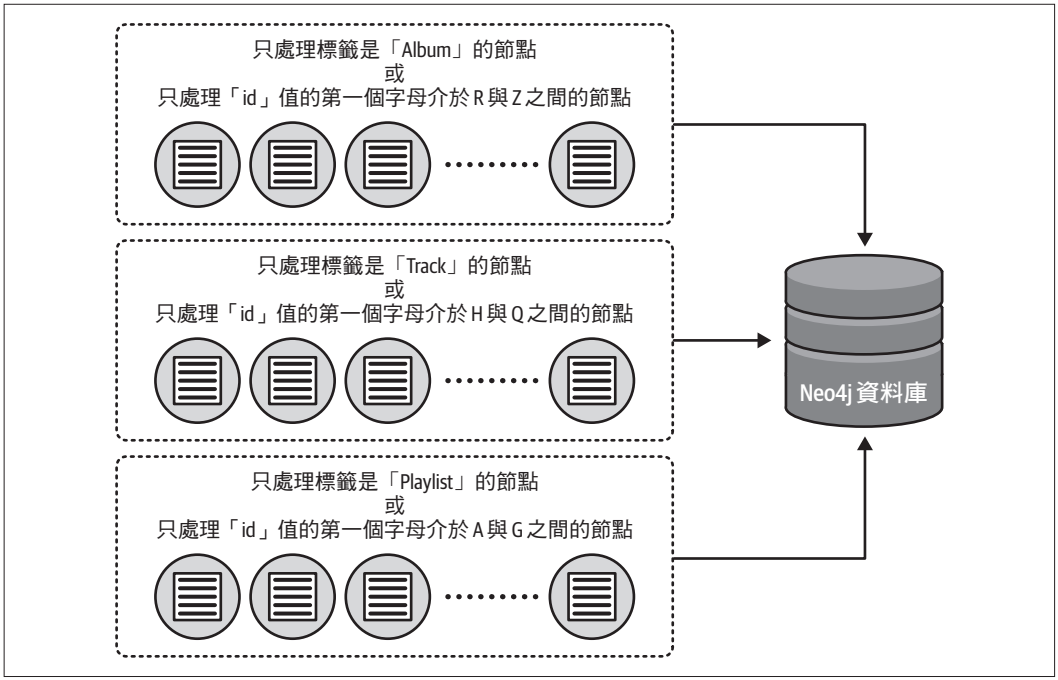

圖 2-12　將不同的陳述式分配給不同的交易池

一起建立節點與關係

在初始化一個幾乎沒有既有資料的新資料庫時，通常會同時處理兩種資料集：一種用來建立節點，另一種則用於各種類型的關係，裡面有大量的節點代號（通常是 id 這類的主鍵屬性）。

或許你想同時匯入這兩組資料，問題在於，將關係匯入資料庫時，你無法確定另一組資料集是否已建立相應的節點，所以你可能想要採取一種看起來很方便，其實很麻煩的方法：在 MERGE 陳述式中使用節點的辨識欄位（例如 id），並且依靠唯一性或節點鍵限制來確保節點存在（於是又遇到鎖定問題了！）。

為了模擬這個情境，我們開啟兩個瀏覽器分頁。在其中一個分頁中，用節點的 id 來 MERGE 它；在第二個分頁中，用兩個節點的 id 來 MERGE 它們，並試著建立它們之間的關係：

```
//011-tab-2-1.cypher
// 瀏覽器分頁 1
MERGE (t:Track {id:'123'})
WITH t
CALL apoc.util.sleep(60000)
RETURN t

//012-tab-2-2.cypher
// 瀏覽器分頁 2
MERGE (t1:Track {id:'123'})
MERGE (t2:Track {id:'234'})
MERGE (t1)-[:SIMILAR_TO]->(t2)
```

或許令你難以置信的是，第二個查詢指令立即 return 了，並成功建立節點與關係，但第一個指令仍在暫停中。難道本書的作者搞錯了嗎？不，這是刻意設計的。id 的唯一性是透過唯一性限制條件（*uniqueness constraint*）來實現的，而這些限制條件是使用鎖定機制來實施的，這是為了防止並行交易違反限制條件。

刪除所有測試資料，並為具有 Track 標籤的節點新增唯一性限制條件：

```
//013-unique-constraint.cypher
MATCH (n) DETACH DELETE n;
CREATE CONSTRAINT track_uk FOR (t:Track) REQUIRE t.id IS UNIQUE;
```

當你重複執行上述的鎖定實驗指令之後，第二個交易會等待第一個完成，以確保資料庫中，只會有一個具有 Track 標籤且 ID 為 123 的節點。

同時新增關係

Neo4j 5.23 版加入名為 block 的新儲存空間格式（*https://oreil.ly/0f43v*），現在即使其他交易持有兩個節點之一的寫入鎖，你仍然可以為這兩個節點建立關係。我們用以下的指令來展示這個行為：

```
//014-tab-3-1.cypher
// 在瀏覽器分頁 1
MATCH (t1:Track {id: 1})
SET t1.popularity = 0.9
WITH t1
CALL apoc.util.sleep(60000)
```

```
//015-tab-3-2.cypher
// 在瀏覽器分頁 2
MATCH (t1:Track {id: 1})
MATCH (t2:Track {id: 2})
CREATE (t1)-[r:SIMILAR]->(t2)
```

你將發現，建立關係的指令會立即執行並返回。

當交易修改或刪除關係時，也會取得該關係的鎖。圖 2-13 是兩個交易同時刪除同一個關係時的情況——第二個交易因為關係被鎖住，而必須等待第一個完成，然後失敗。

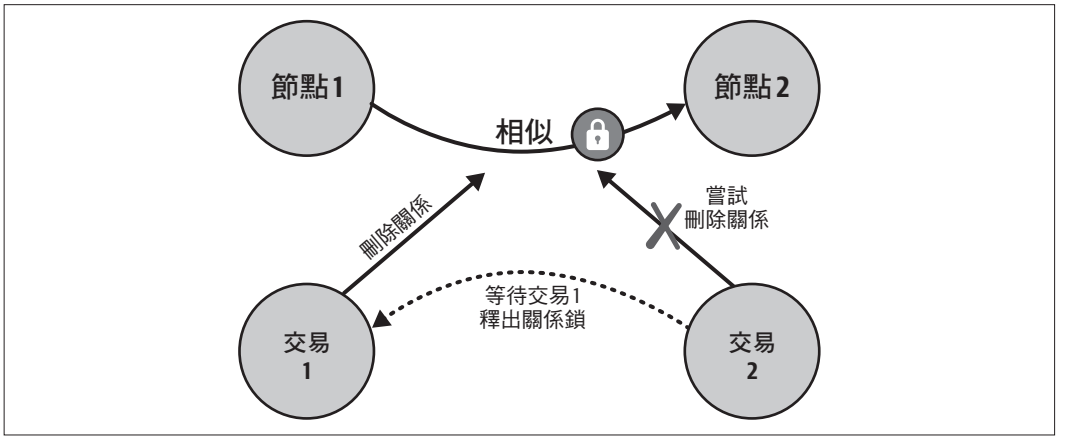

圖 2-13　兩個交易試著刪除同一個關係時的鎖定情況

最後，系統在某些情況下也可能鎖住節點。將節點之間的相似度關係具體化可以突顯這種情況（如圖 2-14 所示，此圖基於第 1 章所介紹的圖模型）。

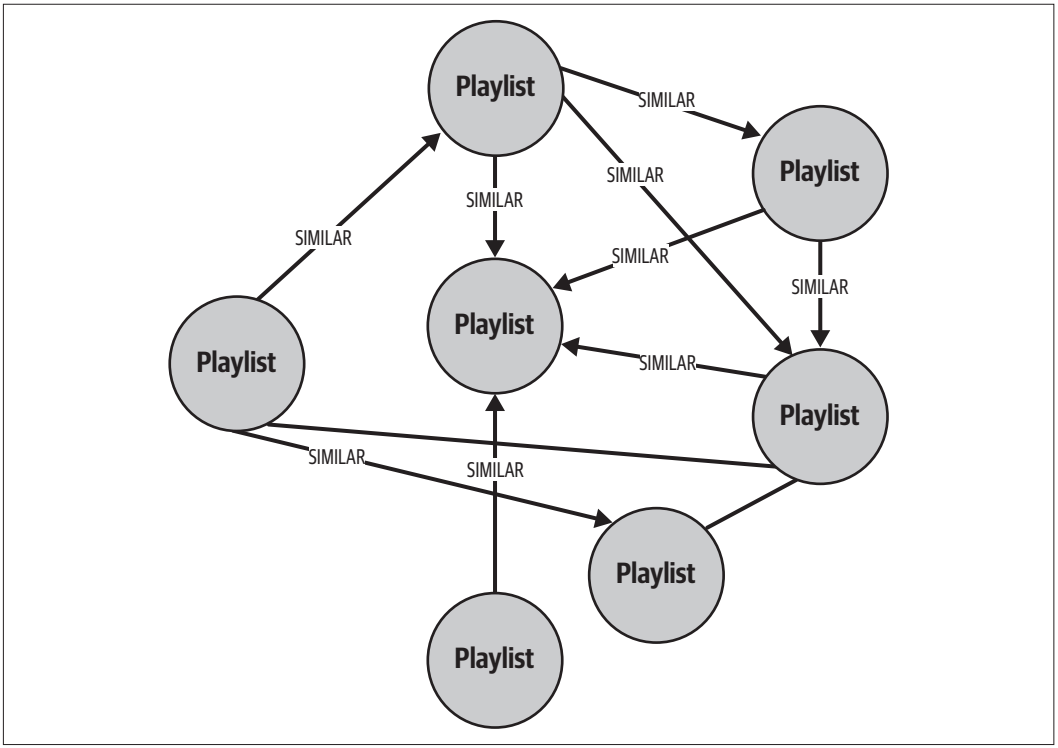

圖 2-14　表示播放清單之間的相似度網路的典型圖模型

由於播放清單之間的相似度會隨著時間而變，你可能要刪除許多舊的相似關係，並插入新計算的關係。

節點會追蹤每個群組（關係類型）的第一個關係，如圖 2-15 所示。

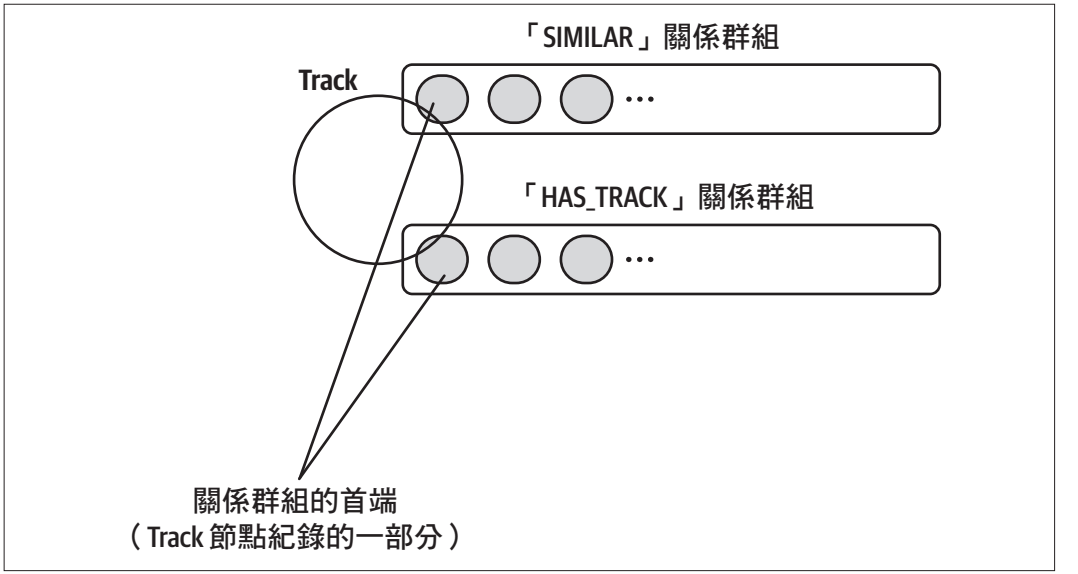

圖 2-15　關係群組首端。關係是用類型與方向來分組的

當兩個交易同時替換某個節點關係群組的首端時，第一個交易也會鎖定該節點。

在任何情況下，鎖定機制是確保資料庫一致性的關鍵。要在 Neo4j 中確保安全、穩定且順暢的寫入體驗，就要瞭解這些鎖的運作細節，並將這些知識應用到匯入流程的設計中。

離線匯入

Neo4j 提供一項專門為了迅速離線匯入初始資料集而設計的工具，稱為 *admin import* 功能。它可以讓你將 CSV 檔案中的資料直接載入資料庫的儲存層，繞過交易層與鎖定機制。這種方法能充分利用機器資源，啟用所有可用的 CPU 核心與 I/O。

大多數關聯式資料庫系統都可將資料匯出為 CSV 格式。只要「從關聯式資料表轉換為圖模型的過程」相對直接且變動不大，這種方法就特別適合用來匯入初始資料。

你的工作主要是產生必要的 CSV 檔案。每一類節點都需要一個或多個檔案，並搭配相應的關係檔案（ CSV ），這些檔案必須包含起始節點與結束節點的 ID 欄位，如圖 2-16 所示。

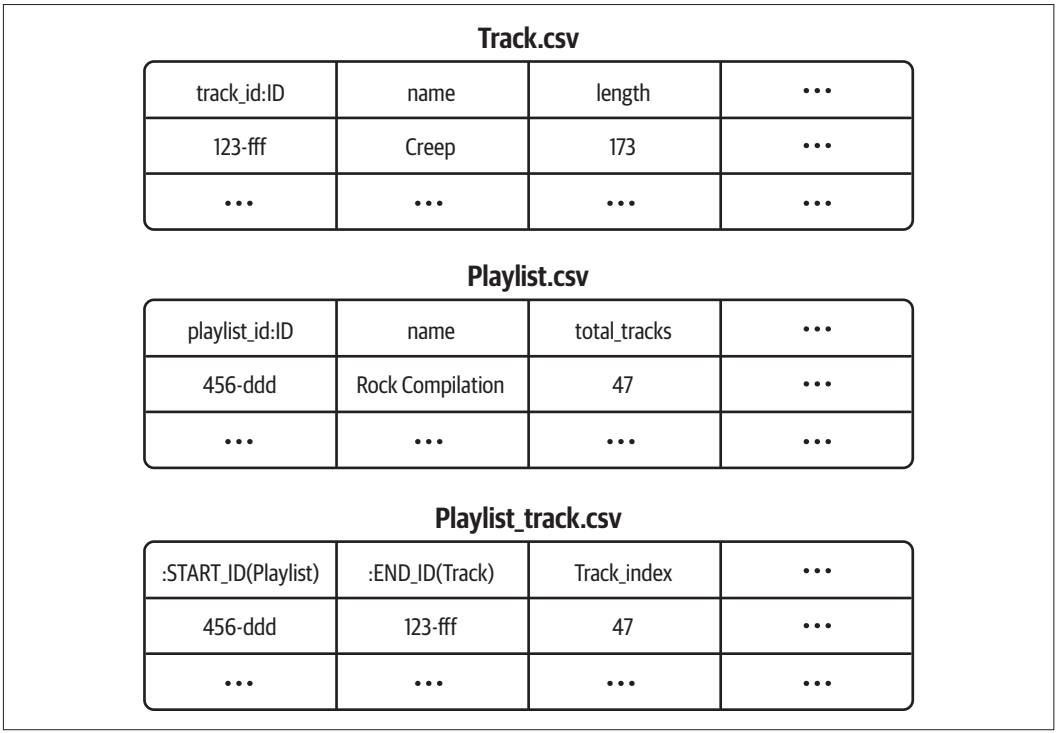

圖 2-16　適用於 neo4j-admin 匯入工具的典型 CSV 檔案結構

你可以從本書的 GitHub 版本庫下載完整的 CSV 格式資料集，並測試匯入流程。以下的效能數據是用 12 核心 MacBook Pro 執行下列命令來匯入該資料集所得：

```
./bin/neo4j-admin database import full
--nodes=:Track=import/full/track.csv
--nodes=:Playlist=import/full/playlist.csv
--relationships=:HAS_TRACK=import/full/track_playlist1.csv
--skip-duplicate-nodes
--multiline-fields=true tracksdb

IMPORT DONE in 1m 23s 708ms.
Imported:
  14336944 nodes
  125451006 relationships
  358453696 properties
Peak memory usage: 1.674GiB
```

admin import 工具提供多種設定選項：你可以逐步匯入資料、用獨立的檔案提供標頭、處理重複項目[1]、指定要使用的 CPU 核心數，並處理多行值…等。Neo4j 已撰寫完整的使用指南（*https://oreil.ly/-jLjy*），協助你根據所產生的 CSV 檔的特性來正確使用此工具。

neo4j-admin import 工具是專為乾淨的資料集而設計的。所謂的「乾淨」是指 CSV 檔不能有重複的節點 ID，而且不能建立指向不存在的節點的關係。這項工具會嚴格地檢查，如果發現重複的節點 ID，匯入就會失敗；它也會檢查那些指向未定義的節點的關係，如果這類遺缺的比例超過某個門檻，匯入程序就會中止。這是一個僅用來建立資料的高效率流程，因此在使用前，務必先處理並仔細檢查資料。

探索其他資料匯入工具

本章介紹了將資料匯入 Neo4j 的基本技術，但 Neo4j 生態系統還有多種工具可支援更進階或持續性的資料匯入流程。你可以根據系統架構，利用 Neo4j 的 Change Data Capture（CDC）功能，以串流的形式更新來自交易系統的新資料，或使用 Kafka Connect 外掛，將 Neo4j 整合至以事件來驅動的資料作業流程。如果你在 Java 環境中開發或整合 BI 工具，neo4j-jdbc 可讓你將 Neo4j 當成傳統資料來源使用。這些工具特別適合需要即時更新、與外部系統同步，或更緊密整合企業資料平台的正式環境。建議你隨著需求的演變，或軟體有需要時，進一步探索這些選項。

總結

你已經掌握能在專案各階段有效率地匯入資料的工具了。`LOAD CSV` 陳述式裡的 `CALL IN TRANSACTIONS` 子句可讓你有效率地匯入大量資料，並用它們來做實驗。當你需要讓自己的應用程式連接至 Neo4j 時，批次匯入是一種順暢的解決方案。雖然使用 Neo4j admin import 可以迅速地離線匯入資料，但如果需要在原始格式與圖模型之間做大量的資料轉換，它可能不是理想的選擇。

1　這項工具是為乾淨的資料集而優化的，因此建議你在使用前，先移除 CSV 檔中的重複資料。