對本書的讚譽

本書提供清楚的說明和實用的技巧，若要利用 LangChain 來建立生成式 AI 與 agent，並將它正式上線，這是必備的資源。如果你想要發揮這個平台的最大威力，本書是必讀之作。

—— *Tom Taulli*，
IT 顧問暨《*AI* 輔助程式開發》作者

這一本探討 LangChain 的完整指南不只介紹文件提取與檢索，也探討正式部署與監控 AI agent 的各個層面。本書透過引人入勝的範例、直覺的圖表，以及實際的程式，讓 LangChain 學習起來既有趣又好玩！

—— *Rajat K. Goel*，
IBM 資深軟體工程師

這是一本完整的 LLM 指南，不只介紹基礎知識，也探討生產階段，充滿技術見解、實用策略，以及強大的 AI 模式。

—— *Gourav Singh Bais*，*Allianz Services* 資深資料科學家暨技術內容撰寫人

設計生成式 AI app 的雛型不難，真正的挑戰是實際部署它們。《*LangChain* 學習手冊》介紹的策略與工具可讓你將腦海中的想法轉化為可正式上線的現代應用程式。

—— *James Spiteri*，
Elastic 產品安全管理總監

《*LangChain* 學習手冊》為 AI 應用程式的建構流程提供清楚的路徑。本書詳細解析靈活的架構與穩健的檢查點，為你打下紮實的基礎，讓你能夠建立可靠、可大規模部署的 AI agent。

—— *David O'Regan*，
GitLab AI/ML 工程經理

第二章

RAG 首部曲：為你的資料建立索引

上一章介紹建立 LLM 應用程式所需的重要 LangChain 組件。你也建立了一個簡單的 AI 聊天機器人，它由一個傳給模型的提示詞和模型產生的輸出組成。但是這個簡單的聊天機器人有一些限制。

如果你的使用情境需要模型未曾學過的知識呢？舉例來說，假設你想使用 AI 來詢問關於某間公司的問題，但相關資訊被放在私人 PDF 或其他類型的文件中，雖然模型供應商會不斷擴充他們的訓練資料集，納入越來越多公開的資訊（無論儲存格式為何），但 LLM 的知識庫依然有兩項主要限制：

私人資料
: 未被公開的資訊不會被放入 LLM 的訓練資料。

即時事件
: 訓練 LLM 是既耗時又昂貴的過程，可能耗時數年，而資料蒐集是最早進行的步驟之一。這導致所謂的知識截止點（knowledge cutoff），也就是 LLM 無法得知某個日期之後的現實事件，該日期通常是訓練資料集定案的時間。這個日期可能是幾個月前，也可能是幾年前，依實際的模型而定。

在這兩種情況下，模型很有可能會輸出幻覺（有誤導性或錯誤的資訊）並輸出不準確的資訊。這個問題就算調整提示詞也無法解決，因為它依賴模型當下的知識。

目標：為 LLM 挑選相關的脈絡

如果 LLM 使用情境所需要的私人 / 即時資料只有一兩頁文本，那麼本章將簡單許多：你只要在傳給模型的提示詞中，直接加入全部的那一段文字即可。

提供資料給 LLM 使用的主要挑戰在於資料量，也就是你想提供的資料超過每次傳給 LLM 的提示詞的上限。當你呼叫模型時，該從如此大量的文字中選擇哪一部分傳入？換句話說，你該如何（在模型協助下）選出最有關的文字來回答各個問題？

在本章與下一章，你將學會如何執行兩個步驟來克服這個挑戰：

1. 為你的文件**建立索引**，也就是預先處理文件，讓應用程式能夠為每一個問題找出最相關的文件。
2. 使用索引來**提取**外部資料，並將它們當成**脈絡**，讓 LLM 基於那些資料產生正確的輸出。

本章的重點是第一個步驟：索引製作，也就是先將文件處理為 LLM 能夠理解與搜尋的格式。這項技術稱為 *retrieval-augmented generation*（RAG，檢索增強生成）。但是在介紹它之前，我們要先討論**為什麼**要先處理文件。

假設你要使用 LLM 來分析特斯拉在 2022 年的年報（*https://oreil.ly/Bp51E*）之中揭露的財務表現與風險，這份年報被儲存為 PDF 格式。你想要向模型詢問「特斯拉在 2022 年面臨哪些主要風險？」之類的問題，並且得到一個基於文件內的風險因素章節的擬人回應。

為了實現這個目標，我們將執行四個關鍵步驟（如圖 2-1 所示）：

1. 從文件中提取文字。
2. 將文字拆成可管理的小段落。
3. 將文字轉換成電腦能夠理解的數值。
4. 將這些文字的數值表徵（representation）儲存至某處，以便快速、輕鬆地取出與問題有關的小段落。

圖 2-1 展示這個預先處理流程，以及轉換文件的過程，這個流程稱為 ingestion（直譯為攝取）。*ingestion* 就是將個人的文件轉換成電腦可以理解和分析的數字，並將之儲存於一種特殊的資料庫之中，以利提取的程序。這些數值的正式名稱是 *embedding*，而特殊的資料庫則稱為**向量庫**（*vector store*）。接下來，我們將仔細看看什麼是 embedding，

以及為什麼它們很重要。讓我們先從一個比較簡單的例子看起，之後會介紹比較複雜的「使用 LLM 來產生 embedding」的例子。

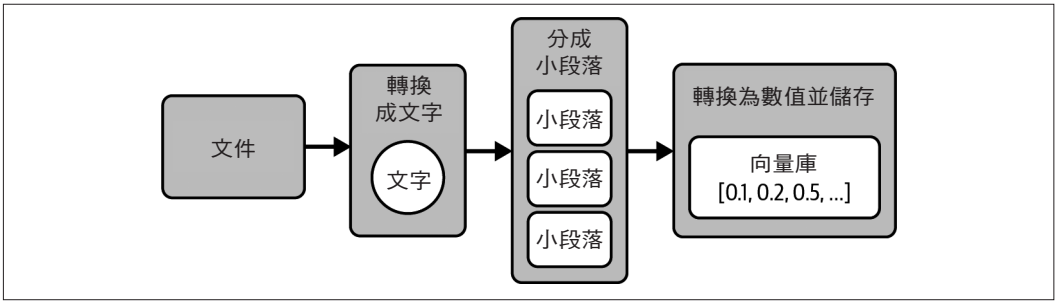

圖 2-1　預先處理文件以供 LLM 使用的四個關鍵步驟

embedding：將文字轉換成數值

embedding 指的是用一串（很長的）數值來表示文字。這是一種有損的表示法，也就是說，這些數值序列無法還原成原始的文字，因此我們通常同時儲存原始文字以及它的數值表徵。

何必如此大費周章？因為這樣可以獲得數值運算的彈性與威力：你可以拿單字來做數學運算！我們來看看這個特性為什麼如此令人興奮。

在 LLM 出現之前的 embedding

早在 LLM 出現之前，電腦科學家就已經在使用 embedding 了，例如，用它來讓網站具備全文搜尋功能，或將電子郵件歸類為垃圾郵件。我們來看一個範例：

1. 取這三個句子：
   - What a sunny day.
   - Such bright skies today.
   - I haven't seen a sunny day in weeks.
2. 列出其中的所有單字，每個單字只列出一次：*what*、*a*、*sunny*、*day*、*such*、*bright* …等。
3. 為每一個句子之中的每一個單字指定數字，若該單字未出現，將它標為 0，若出現一次，標為 1，若出現兩次，標為 2，以此類推。

結果如表 2-1 所示。

表 2-1　三個句子的單字 embedding

單字	What a sunny day.	Such bright skies today.	I haven't seen a sunny day in weeks.
what	1	0	0
a	1	0	1
sunny	1	0	1
day	1	0	1
such	0	1	0
bright	0	1	0
skies	0	1	0
today	0	1	0
I	0	0	1
haven't	0	0	1
seen	0	0	1
in	0	0	1
weeks	0	0	1

在這個模型中，*I haven't seen a sunny day in weeks* 的 embedding 是數值 *0 1 1 1 0 0 0 0 1 1 1 1 1*。這種技術稱為 *bag-of-words*（詞袋）模型，而這些 embedding 也稱為 *sparse*（稀疏）*embedding*（或 sparse vector，*vector*（向量）是數值序列的另一個名稱），因為大部分的數字都是 0。大多數的英文句子都只會用到所有英文單字中的一小部分。

這個模型可以用來：

搜尋關鍵字
: 找出哪些文件有某個單字或多個單字。

分類文件
: 先為一批已被標上垃圾郵件或非垃圾郵件的範例算出 embedding，計算它們的平均值，得到每一個單字在各個類別（垃圾郵件或非垃圾郵件）之中的平均出現頻率。之後只要比較每一份新文件與這些平均數即可進行分類。

這種做法的限制在於，這個模型完全不瞭解意思，只知道出現了哪些單字。例如，*sunny day* 和 *bright skies* 的 embedding 看起來截然不同，因為它們沒有重複的單字，即使我們

知道它們的意思相近。又或者，在電子郵件分類問題中，垃圾郵件的寄件者可以將常見的「垃圾郵件單字」換成同義詞來繞過篩選。

下一節將展示語意 embedding 如何克服這個限制：它們用數值來表示文字的意思，而不是用數值來表示文本中的確切單字。

使用 LLM 來產生 embedding

我們直接進入 LLM-based embedding（由 LLM 生成的 embedding），省略中間的所有 ML 發展過程。你只要知道，從上一節介紹的簡單方法到這一節介紹的進階方法之間有一個逐步演變的過程就好了。

你可以將 embedding 模型視為 LLM 訓練過程的副產品。前言提過，LLM 的訓練程序（讓它從大量文本中學習）可讓 LLM 知道如何以適當的文字（輸出）來延續提示詞（或輸入）並完成它。模型的這種能力來自它能夠理解單字與句子在周圍脈絡之中的意義，而這種理解能力，是從訓練資料運用單字的方式學來的。我們可以將這種對於提示詞意義（或語意）的**理解**提取為一種數值表徵（也就是 embedding），並直接應用在其他有趣的使用情境。

在實務上，大多數的 embedding 模型都是專門為了這個目的而訓練的，在訓練時採用和 LLM 類似的架構與訓練流程，因為這樣比較有效率，而且能夠產生品質更好的 embedding[1]。

所以 *embedding* **模型**是一種演算法，它接收一段文字，並輸出該文字之意義的數值表徵，事實上，數值表徵就是一個浮點（小數）數值序列，通常一個序列之中的數值數量介於 100 到 2,000 個數字之間，也就是 100 到 2,000 個**維度**。這些 embedding 也稱為 *dense*（**密集**）embedding，以區別上一節提到的 *sparse*（**稀疏**）embedding，因為在 dense embedding 中，通常不會有維度是 0。

不同的模型會產生不同的數值，也會產生不同長度的數字序列。它們都是各個模型特有的；也就是說，你不能直接比較不同模型生成的 embedding，即使數值序列一樣長。切勿一起使用不同模型生成的 embedding。

1 Arvind Neelakantan et al., "Text and Code Embeddings by Contrastive Pre-Training" (*https://oreil.ly/YOVmh*), arXiv, January 21, 2022.

語意 embedding 解說

考慮這三個單字：*lion*、*pet* 和 *dog*。直覺上，哪一對單字的特性比較相似？顯然答案是 *pet* 和 *dog*。但電腦無法利用這種直覺，或是英文的微妙語感。因此，為了讓電腦能夠區分 lion、pet 和 dog，你要將它們轉換成電腦能夠理解的語言，也就是數值。

圖 2-2 說明如何將每一個單字轉換成假想的數值表徵，同時保留它們的意義。

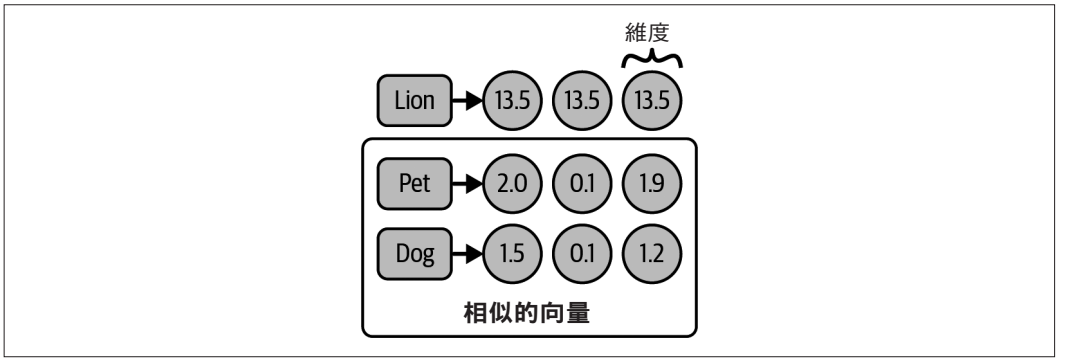

圖 2-2　單字的語意表徵

圖 2-2 是每一個單字及其語意 embedding。注意，這些數值本身沒有特定的意義，重點在於，當兩個單字（或句子）意義相近時，它們的數值序列將互相接近，當意義不相關時，它們的數字序列會遠離彼此。如你所見，每一個數值都是一個**浮點值**，而每一個數值都代表一個語意**維度**。我們來看看所謂的**接近**是什麼意思：

如果將這些向量畫在三維空間中，它可能會像圖 2-3 這樣。

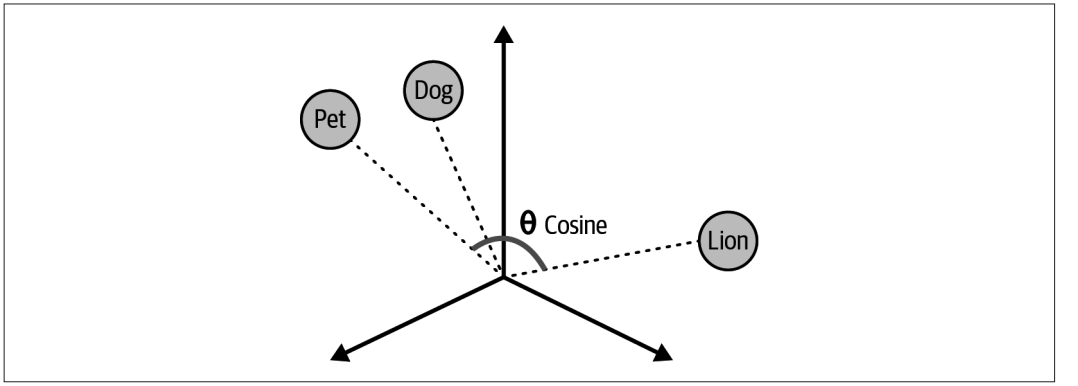

圖 2-3　在多維空間中畫出單字向量

從圖 2-3 可以看到，*pet* 和 *dog* 向量之間的距離比它們和 *lion* 之間的距離更近。你也可以看到，不同向量之間的夾角大小會隨著它們之間的相似程度而異。例如，*pet* 和 *lion* 的夾角比 *pet* 和 *dog* 的夾角更大，這意味著後面那一對單字更加相似。向量之間的夾角越小、距離越短，代表它們之間的相似度越高。

要計算多維空間中的兩個向量之間的相似度，有一種有效的方法稱為餘弦相似度。**餘弦相似度**（*cosine similarity*）的算法是先算出兩個向量的內積，再除以它們的長度的乘積，最後輸出一個介於 –1 到 1 之間的數值，0 代表向量之間沒有關係，–1 代表完全不相似，1 則代表完全相似。因此，對於這三個單字來說，*pet* 和 *dog* 之間的餘弦相似度可能是 0.75，而 *pet* 和 *lion* 可能是 0.1。

這種技巧將句子轉換成捕捉語意的 embedding，並計算不同句子之間的語意相似度。我們可以利用這種技巧來讓 LLM 找出最相關的文件，以回答之前的那個關於特斯拉 PDF 文件的問題。現在你已經瞭解整體概念了，接下來要回到預先處理文件的第一步（製作索引）。

embedding 的其他用途

這些數字序列和向量有一些有趣的特性：

- 就像你之前學到的，如果說向量描述了高維空間之中的一個點所在的位置，那麼彼此相近的點有比較相似的意思，因此我們可以用距離函式來衡量相似度。
- 一群彼此接近的點可以視為彼此相關，因此，你可以使用分群（clustering）演算法來辨識主題（也就是一群點），並將新的輸入歸類為既有的主題之一。
- 若取多個 embedding 的平均值，這個平均後的 embedding 可視為這群 embedding 的整體意義，也就是說，你可以為一篇長文件（例如這本書）建立 embedding：
    1. 分別產生每一頁的 embedding。
    2. 取所有頁面的 embedding 的均值，作為整本書的 embedding。

- 你可以使用簡單的加法和減法在「意義」空間中「穿梭」，例如，*king* – *man* + *woman* = *queen*。理論上，將 *king* 的意義（語意 embedding）減去 *man* 的意義會得到一個較抽象的 *monarch*（君主）的意義，然後，加上 *woman* 的意義，會得到近似 *queen* 的意義（或 embedding）。
- 有些模型除了可以輸出文字之外，也可以輸出非文字的 embedding，例如圖像、影片和聲音，讓你能夠找出（舉例）與一段文字最相似或最相關的圖像。

本書不深入探討以上的所有特性，但你要知道的是，這些特性可以應用於許多領域（*https://oreil.ly/PU2C8*），例如：

搜尋
: 為新的查詢找到最相關的文件

分群
: 將一批文件分成幾個群組（例如主題）

分類
: 將新文件指派給已被辨識出來的群組或標籤（例如某個主題）

推薦
: 基於一份文件推薦相似的其他文件

異常偵測
: 找出和之前看過的文件非常不同的文件

希望這個專欄能夠讓你直覺地感受 embedding 的彈性，並在未來的專案中派上用場。

將文件轉換成文字

正如本章開頭所說的，預先處理文件的第一步是將它轉換成文字。為了做到這件事，你要設計邏輯來解析並提取文件內容，同時盡量避免降低品質。所幸，LangChain 的文件

載入器（*document loaders*）可以用來處理「解析邏輯」，有助於將各種來源的資料「載入」至一個包含文字與相關中繼資料的 `Document` 類別裡。

舉例來說，考慮一個簡單的 *.txt* 檔案。你可以直接匯入 LangChain 的 `TextLoader` 類別來提取文字，像這樣：

Python

```
from langchain_community.document_loaders import TextLoader

loader = TextLoader("./test.txt")
loader.load()
```

JavaScript

```
import { TextLoader } from "langchain/document_loaders/fs/text";

const loader = new TextLoader("./test.txt");

const docs = await loader.load();
```

輸出：

```
[Document(page_content='text content \n', metadata={'line_number': 0, 'source':
    './test.txt'})]
```

上面的程式假設在當下的目錄內有一個名為 *test.txt* 的檔案。LangChain 文件載入器的用法皆遵循一種相似的模式：

1. 從一長串的整合列表（*https://oreil.ly/iLJ33*）中選擇適合你的文件類型的載入器。
2. 建立該載入器的實例，並傳入設定參數，包括文件的位置（通常是檔案系統路徑或網頁位址）。
3. 呼叫 `load()` 來載入文件，該方法回傳一個文件串列，可傳到下一個階段（稍後會詳細說明）。

除了 *.txt* 檔案之外，LangChain 也提供其他常見檔案類型的文件載入器，包括 *.csv*、*.json*、Markdown，以及與常見平台之間的整合，例如 Slack 和 Notion。

舉例來說，你可以使用 `WebBaseLoader` 來載入一個網頁 URL 的 HTML，並將之解析成文字。

請安裝 beautifulsoup4 套件：

```
pip install beautifulsoup4
```

Python

```
from langchain_community.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://www.langchain.com/")
loader.load()
```

JavaScript

```
// 安裝 cheerio：npm install cheerio
import {
  CheerioWebBaseLoader
} from "@langchain/community/document_loaders/web/cheerio";

const loader = new CheerioWebBaseLoader("https://www.langchain.com/");

const docs = await loader.load();
```

在特斯拉 PDF 案例中，我們可以使用 LangChain 的 `PDFLoader` 來提取 PDF 文件的文字：

Python

```
# 安裝 pdf 解析程式庫
# pip install pypdf

from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("./test.pdf")
pages = loader.load()
```

JavaScript

```
// 安裝 pdf 解析程式庫：npm install pdf-parse

import { PDFLoader } from "langchain/document_loaders/fs/pdf";

const loader = new PDFLoader("./test.pdf");

const docs = await loader.load();
```

如此一來，即可從 PDF 文件提取文字，並將它儲存在 `Document` 類別中。但有一個問題，我們載入的文件超過 100,000 個字元，因此無法放入大多數 LLM 或 embedding 模型的脈絡窗口（context window）中。為了克服這個限制，我們要先將 `Document` 分成可

管理的文字段落，再將它們轉換成 embedding，並進行語意搜尋，這就帶來第二個步驟（提取）。

LLM 和 embedding 模型都有它們能夠處理的輸入 / 輸出詞元（token）數量的嚴格上限。這個上限一般稱為**脈絡窗口**（*context window*），通常是指輸入加輸出，也就是說，如果脈絡窗口是 100（稍後會討論單位），而輸入是 90，那麼輸出最多只能是 10。脈絡窗口通常以詞元數量為單位，例如 8,192 個詞元。如前言所述，詞元是一種文字的數值表徵，每一個詞元通常包含大約三、四個英文字。

將文字拆成小段落

將一大段文字拆成小段落看起來是很簡單的事情，事實上，將**語意相關**（有相關的意思）的文字段落放在一起是相當複雜的過程。為了協助將大型文件拆成有意義的小段落，LangChain 提供了 `RecursiveCharacterTextSplitter`，它的運作方式如下：

1. 接收一系列的分隔符號，並按照重要性排序。預設的分隔符號有：
   a. 段落分隔符號：`\n\n`
   b. 行分隔符號：`\n`
   c. 單字分隔符號：空格字元
2. 為了遵守所指定的段落長度（例如 1,000 個字元），先按照段落將文本拆開。
3. 如果段落的長度超過所指定的長度，那就使用下一個分隔符號（行）來拆分。持續這個程序，直到所有段落皆小於目標長度，或沒有其他分隔符號可用為止。
4. 以 `Document` 的形式輸出每一個段落，並附帶原始文件的中繼資料，以及它在原始文件之中的位置資訊。

我們來看一個範例：

Python

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

loader = TextLoader("./test.txt") # 或任何其他載入器
docs = loader.load()
```

```
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
)
splitted_docs = splitter.split_documents(docs)
```

JavaScript

```
import { TextLoader } from "langchain/document_loaders/fs/text";
import { RecursiveCharacterTextSplitter } from "@langchain/textsplitters";

const loader = new TextLoader("./test.txt"); // 或任何其他載入器
const docs = await loader.load();

const splitter = new RecursiveCharacterTextSplitter({
  chunkSize: 1000,
  chunkOverlap: 200,
});

const splittedDocs = await splitter.splitDocuments(docs)
```

在上面的程式碼中，文件載入器所建立的文件被分成每段 1,000 個字元之內的小段落，且各個段落之間有 200 個字元的重疊，以保留部分脈絡。最終的結果仍然是一系列的文件，每一個文件最多有 1,000 個字元，並且沿著自然的文本結構（段落、換行，最後到單字）來劃分。這種技巧利用了文字本身的結構來讓每一個段落皆保持一致且易讀。

`RecursiveCharacterTextSplitter` 也可以將程式語言和 Markdown 文件分成語意相關的段落。這是使用各種語言本身的關鍵字作為分隔符號來做的，可確保（舉例）每一個函式的主體皆被放在同一個段落中，不會被拆成多個段落。程式語言通常比文本更具結構性，因此較不需要讓段落互相重疊。LangChain 內建了許多常見程式語言（例如 Python、JS、Markdown、HTML 等等）的 splitter。舉一個例子：

Python

```
from langchain_text_splitters import (
    Language,
    RecursiveCharacterTextSplitter,
)

PYTHON_CODE = """
def hello_world():
    print("Hello, World!")

# 呼叫函式
```

```
});

console.log(index_attempt_2);

// 如果文件被修改，新版本會被寫入，
// 且來源相同的所有舊版本都會被刪除。
docs[0].pageContent = 'I modified the first document content';
const index_attempt_3 = await index({
  docsSource: docs,
  recordManager,
  vectorStore,
  options: {
    cleanup: 'incremental',
    sourceIdKey: 'source',
  },
});

console.log(index_attempt_3);
```

首先，你要建立一個 record manager，它會追蹤哪些文件已經被建立索引了。接著使用 `index` 函式，將向量庫與新的文件串列同步。這個範例使用的是 incremental 模式，所以有相同 ID 的文件都會被新的版本取代。

索引最佳化

RAG 索引建立階段的基本做法是將文件分段，並製作段落的 embedding。但是這種基本的做法會導致不一致的提取結果，也很容易出現幻覺，尤其是資料來源包含圖像與表格時。

你可以採取各種策略來提升索引建立階段的準確性與效能。接下來的幾節將介紹其中三種方法：MultiVectorRetriever、RAPTOR 與 ColBERT。

MultiVectorRetriever

如果一份文件裡除了有文字之外也有表格，你不能直接將文字拆成小段落並做成 embedding，否則，整份表格很容易遺失。為了解決這個問題，你可以將「用來產生回答的文件」與「供檢索器（retriever）參考的文件」分開。做法如圖 2-5 所示。

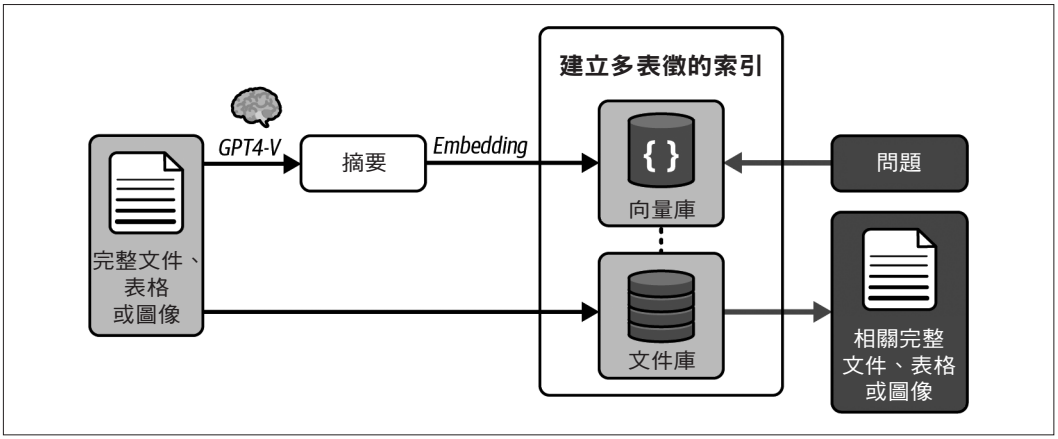

圖 2-5　為單一文件的多個表徵建立索引

舉例來說，當文件包含表格時，你可以為表格元素產生摘要並建立 embedding，並為每一個摘要附加指向原始表格的 `id` 參考。接著，將原始表格儲存在另一個文件庫中。最後，當使用者的查詢句提取表格摘要時，將完整的原始表格當成最終提示詞的脈絡資料，一併傳給 LLM 以生成回答。這種技巧能夠讓模型獲得回答問題所需的完整脈絡資訊。

我們來看一個範例。首先，使用 LLM 來為文件產生摘要：

Python

```
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings
from langchain_postgres.vectorstores import PGVector
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI
from langchain_core.documents import Document
from langchain.retrievers.multi_vector import MultiVectorRetriever
from langchain.storage import InMemoryStore
import uuid

connection = "postgresql+psycopg://langchain:langchain@localhost:6024/langchain"
collection_name = "summaries"
embeddings_model = OpenAIEmbeddings()
# 載入文件
```

```
        # 自訂 User-Agent header，以遵循 Wikipedia 的最佳做法
        headers = {"User-Agent": "RAGatouille_tutorial/0.0.1"}

        response = requests.get(URL, params=params, headers=headers)
        data = response.json()

        # 提取網頁內容
        page = next(iter(data["query"]["pages"].values()))
        return page["extract"] if "extract" in page else None

    full_document = get_wikipedia_page("Hayao_Miyazaki")

    ## 建立索引
    RAG.index(
        collection=[full_document],
        index_name="Miyazaki-123",
        max_document_length=180,
        split_documents=True,
    )

    # 查詢
    results = RAG.search(query="What animation studio did Miyazaki found?", k=3)
    results

    # 使用 langchain retriever
    retriever = RAG.as_langchain_retriever(k=3)
    retriever.invoke("What animation studio did Miyazaki found?")
```

使用 ColBERT 可以提升被 LLM 當成脈絡資料來提取的文件之間的相關性。

總結

在這一章，你學會如何使用各種 LangChain 模組來為 LLM 應用程式準備與預先處理文件。文件載入器可以從資料來源中提取文字，文字分段器能夠將文件分成語意相近的小段落，而 embedding 模型可將文字轉換成表示其意義的向量表徵。

此外，向量庫可對這些 embedding 執行 CRUD 操作及複雜的運算，以找出語意相似的文字段落。最後，索引最佳化策略可讓 AI app 提升 embedding 的品質，並準確地提取內含表格等半結構化資料的文件。

在第 3 章，你將學習如何根據查詢句，從向量庫快速地提取最相似的文件段落，將它傳給模型作為脈絡資料，進而輸出準確的結果。