前言

你應該想知道我們是誰，還有我們為什麼要寫這本書。

Harry 在完成他的上一本書《**測試驅動開發｜使用** *Python*》（*https://www.obeythetestinggoat.com/*）（O'Reilly）的時候，發現自己有一堆關於結構的疑問，例如，如何以最好的方式架構 app，讓它容易被測試？更具體地說，如何讓單元測試完全覆蓋核心的商務邏輯，將所需的整合及端對端測試工作降到最低？他曾經含糊地提到「六邊形架構（Hexagonal Architecture）」和「連接埠和配接器（Ports and Adapters）」以及「功能核心，命令外殼（Functional Core, Imperative Shell）」，但他應該誠實地承認，他並未真正理解這些東西，也從未實際動手做過。

後來，他很幸運地認識 Bob，Bob 知道所有這些問題的解答。

因為 Bob 的團隊裡面沒有人負責軟體架構，所以他被迫成為軟體架構師。事實上，他非常不擅長這項工作，但他很幸運地認識 Ian Cooper，從他那裡學到撰寫與理解程式碼的方法。

管理複雜性，解決商業問題

我們兩人都在 MADE.com 工作，它是一家在網路販售家具的歐洲電子商務公司，我們在這家公司裡面採用本書介紹的技術來建構分散式系統，用它來建立真正的商業問題的模型。我們的領域（domain）範例是 Bob 為 MADE 建構的第一個系統，我們希望用這本書來記錄當新程式員加入團隊時，我們想要傳授給他們的所有教材。

碁峰 www.gotop.com.tw

MADE.com 運營一個連結全球貨運合作夥伴與製造商的供應鏈。為了降低成本，我們試著優化貨物入倉的配送程序，避免到處亂放未出售的貨物。

在理想的情況下，我們會讓你購買的沙發在你下單的當天就到港，並且直接將它送到你家，完全不需要入倉存放。如果貨櫃船需要三個月才能將貨物送達，正確安排所有時間是非常麻煩的平衡工作，在過程中，我們可能遇到貨物損壞或進水、暴風雨導致意外的延誤、物流業者不當處理貨物、文件遺失、顧客改變主意並修改訂單等事情，為了解決這些問題，我們做了一個智慧軟體來代表真實世界的各種活動，盡量將這個業務自動化。

為何選擇 Python？

既然你已經在看這本書了，我們應該不需要說服你為何 Python 很棒了，真正問題應該是「為什麼 *Python* 社群需要這種書？」答案與 Python 的流行度和成熟度有關：或許 Python 是這個世界成長速度最快的程式語言，而且在流行度排行榜名列前茅，但是它直到最近才開始被用來解決多年來一直都是用 C# 與 Java 解決的問題。初創公司已經變成真正的企業了，web app 與腳本自動化也變成（悄悄說）**企業軟體**了。

在 Python 世界中，我們通常會引用 Zen of Python：「用明顯的方法來完成一件事，而且最好只有一種[1]。」不幸的是，隨著專案規模的成長，最明顯的做法不一定有助於管理複雜性與不斷演進的需求。

雖然這本書探討的技術和模式都不新，但它們在 Python 世界裡大都是新的。本書無法取代這個領域的經典，例如 Eric Evans 的《*Domain-Driven Design*》與 Martin Fowler 的《*Patterns of Enterprise Application Architecture*》（兩本都是 Addison-Wesley Professional 出版的），我們經常參考這兩本書，也建議你閱讀。

但是在經典書籍中的範例程式通常都是用 Java 或 C++/# 編寫的，如果你是 Python 人，而且沒有長時間用過這些語言（或其實完全沒有用過），這些程式理解起來可能非常…費力。這也是另一本經典文獻的新版使用 JavaScript 的原因——Fowler 的《**重構**（*Refactoring*）》（Addison-Wesley Professional）。

1 `python -c "import this"`。

碁峰 www.gotop.com.tw

TDD、DDD 與事件驅動架構

我們都知道，管理複雜性的工具有三種，按照著名程度依序是：

1. **測試驅動開發**（TDD）可以協助我們建構正確的程式以及重構或加入新功能，而不需要擔心問題回歸。但測試程式很難帶來最好的結果：該如何確定它們是否以最快的速度執行？我們可以從快速、無依賴關係的單元測試得到盡可能多的覆蓋率和回饋，並且盡量減少緩慢、不穩定的端對端測試嗎？
2. **領域驅動測試**（DDD）讓我們把精力放在建構好的商業領域模型上，但如何確保模型不受基礎設施問題困擾，而且不會變得難以更改？
3. 用訊息來整合的鬆耦合（微）服務（有時稱為**反應式微服務**（*reactive microservice*））是管理多個 app 或商業領域之間的複雜性的成熟解決方案。但大家不太知道如何使用 Python 領域的成熟工具來製作它，例如 Flask、Django、Celery 等。

如果你不太瞭解微服務（或對它不感興趣），別害怕，我們探討的絕大多數模式，包括事件驅動架構教材，都一定可以在單體架構中使用。

本書的目標是介紹一些經典的架構模式，並展示它們如何支援 TDD、DDD 與事件驅動服務，我們希望把它寫成一本參考書，告訴你如何以符合 Python 風格的方式製作它們，並且提供一個起點，協助你在這個領域中進行更深入的研究。

誰需要讀這本書？

親愛的讀者，我們假設你有這些背景：

- 你已經熟悉一些相當複雜的 Python app 了。
- 你已經吃過管理那種複雜性的苦頭了。
- 你不需要知道任何關於 DDD，或任何關於經典 app 架構模式的事情。

我們會圍繞著一個範例 app 建構我們摸索出來的架構模式，一章一章地搭建它。因為我們採用 TDD，所以會先展示測試程式，再進行實作。如果你沒有先編寫測試程式的習慣，可能一開始會覺得很奇怪，但我們希望你儘早習慣在知道程式的內在如何建構之前，先看到程式如何「被使用」（也就是從外面看）。

碁峯 www.gotop.com.tw

我們會使用一些 Python 框架與技術，包括 Flask、SQLAlchemy 和 pytest，以及 Docker 和 Redis。如果你已經熟悉它們了，很好，不過這不是必要的。這本書的主要目標是建立架構，將它裡面的具體技術選項變成次要的實作細節。

簡介你將學會的東西

本書分成兩大部分，以下是我們將探討的主題，以及介紹它們的章節。

第一部分，建立架構來支援領域模型的建構

領域建模與 DDD（第 1 章與第 7 章）

在某種層面上，大家都已經從教訓中知道，複雜的商業問題必須以領域模型的形式反映在程式碼裡面。但是為什麼在做這件事的同時，很難避免與基礎設施、web 框架或其他問題糾纏不清？在第一章，我們會粗略介紹**領域建模**以及 DDD，並且使用一個沒有外部依賴項目的模型以及簡單的單元測試來展示如何上手。稍後我們會回到 DDD 模式，討論如何選擇正確的 aggregate（集合體），以及它與資料完整性問題有什麼關係。

Repository、 Service Layer，以及 Unit of Work 模式（第 2、4、5 章）

這三章將介紹三種密切相關而且相輔相成的模式，這種模式可以支持我們的偉大目標，讓模型沒有外部依賴項目。我們將圍繞著持久保存機制建構一個抽象層，並且建立一個服務層來定義系統的入口和描述主要的用例（use case）。我們將展示這一層如何輕鬆地為系統建構精簡的入口，無論系統是 Flask API 還是 CLI。

關於測試與抽象的一些想法（第 3 章與第 6 章）

在展示第一個抽象（Repository 模式）之後，我們借此機會概述如何選擇抽象，以及它們在選擇軟體的耦合方式時發揮什麼作用。介紹 Service Layer 模式之後，我們會探討一些關於實作**測試金字塔**，以及在最高抽象級別編寫單元測試的事項。

第二部分，事件驅動架構

事件驅動結構（第 8-11 章）

我們會再介紹三種相輔相成的模式：Domain Events、Message Bus 以及 Handler 模式。**領域事件**（*domain event*）是用來描述「與系統的互動會觸發其他的互動」這種概念的工具。我們使用 *message bus*（**訊息匯流排**）來讓一些動作觸發事件並呼叫適

碁峰 www.gotop.com.tw

當的**處理式**（*handler*）。接下來我們會討論如何將事件當成一種模式，在微服務架構中，用來整合不同的服務。最後，我們將區分**指令**（*command*）與**事件**（*event*）的不同。現在的 app 基本上已經是一個訊息處理系統了。

命令查詢責任隔離（第 *12* 章）

我們將舉一個例子說明**命令查詢責任隔離**，包含使用事件以及不使用事件。

依賴注入（第 *13* 章）

我們將整理明確與隱性依賴關係，並且實作一個簡單的依賴注入框架。

其他內容

我該如何由此至彼？（結語）

雖然舉一個簡單的例子從零開始實作架構模式看起來很容易，但很多人可能會問，該如何將這些原則應用在既有的軟體中。我們會在結語提供一些提示和參考讀物的連結。

範例程式，以及在過程中一起編寫程式

雖然你正在看這本書，但你應該可以認同這句話：學程式最好的方式就是動手寫程式。我們的知識大部分都是從和別人合作、一起編寫程式，在實作中學習來的，我們希望在這本書裡面盡量為你重現這些體驗。

因此，我們圍繞著一個範例專案來建構這本書（不過有時也會加入其他範例）。我們會隨著章節的進展建構這個專案，很像你真的與我們一起工作，我們也會在各個步驟解釋我們在做什麼，以及為何如此。

但是若要真正掌握這些模式，你必須修改程式，瞭解它們是如何工作的。你可以在 GitHub 找到所有程式碼，每一章都有它自己的分支。你也可以在 GitHub 找到分支的清單（*https://github.com/cosmicpython/code/branches/all*）。

你可以透過這三種方式，跟著這本書一起寫程式：

- 建立你自己的 repo，並且按照本書的範例跟著我們一起建構 app，偶爾察看我們的 repo 裡面的提示。不過，提醒你一下，如果你看過 Harry 的上一本書，並且曾經跟著它一起寫程式，你將發現，這本書需要你自行探索更多東西，你可能會非常依賴 GitHub 上的可用版本。

碁峰 www.gotop.com.tw

- 試著將每一章的模式用在你自己的專案上（最好是小型的玩具專案），看看它們是否能在你的用例中發揮作用。這是高風險、高報酬的工作（而且高努力！），雖然將書中的內容正確地用在你自己的專案需要費很多工夫，但從另一面看，你會學到最多東西。
- 為了少花點工夫，我們在每一章也會提出一個「給讀者的習題」，並告訴你一個 GitHub 地點，你可以在那裡下載部分完成的程式，然後自行編寫缺少的部分。

如果你想要在自己的專案中使用其中一些模式，透過簡單的範例來安全地練習是很好的做法。

當你閱讀各章時，至少要 `git checkout` 我們的 repo 的程式。閱讀真的可以運作的 app 程式可以幫助你在過程中解決很多問題，讓一切更加真實。每一章的開頭會告訴你如何做這件事。

本書編排慣例

本書使用下列的編排規則：

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`）

在長程式中使用，或是在文章中代表變數、函式名稱、資料庫、資料型態、環境變數、陳述式、關鍵字等程式元素。

定寬粗體字（**`Constant width bold`**）

代表應由使用者親自輸入的命令或其他文字。

定寬斜體字（*`Constant width italic`*）

代表應換成使用者提供的值，或由上下文決定的值。

這個圖案代表提示或建議。

碁峰 www.gotop.com.tw

引言

為什麼我們的設計會出錯？

聽到**混亂**這個詞的時候，你想到什麼？或許你想到嘈雜的證券交易所，或是早上的廚房，所有東西都亂糟糟的，至於**秩序**，你可能會想到一間寧靜的空房間。但是，對科學家來說，混亂的特徵是同質性（homogeneity）（同一性，sameness），秩序的特徵是複雜性（complexity）（差一性，difference）。

例如，井然有序的花園是高秩序的系統。園丁使用小路和柵欄劃分界限，並且劃出花壇和菜田。隨著時間過去，花園會不斷演變，植物會越來越豐富且濃密，但是如果沒有精心照顧，花園就會失控。雜草會讓其他植物難以存活，蔓延到小路上面，直到最後，每一個部分看起來都大同小異，雜草叢生且無人管理。

軟體系統也會趨向混亂，當我們剛開始建構新系統時，我們都有宏偉的構想，認為程式可以保持乾淨整潔，但是隨著時間流逝，它會累積很多殘留物和邊緣情況，最終變成一個令人疑惑的管理類別及工具模組沼澤，曾經被理性地分層的架構已經頹然崩塌。混亂的軟體系統的特點是功能的同一性，裡面有具備領域知識，卻也可以送出 email 和執行 logging 的 API 處理式、不進行計算卻執行 I/O 的「商務邏輯」，而且所有東西都與任何其他東西互相關聯，因此改變系統的每一個地方都危機重重。因為這種情況太常見了，所以軟體工程師用他們自己的說法來描述混亂：大泥球反模式（圖 P-1）。

碁峯
www.gotop.com.tw

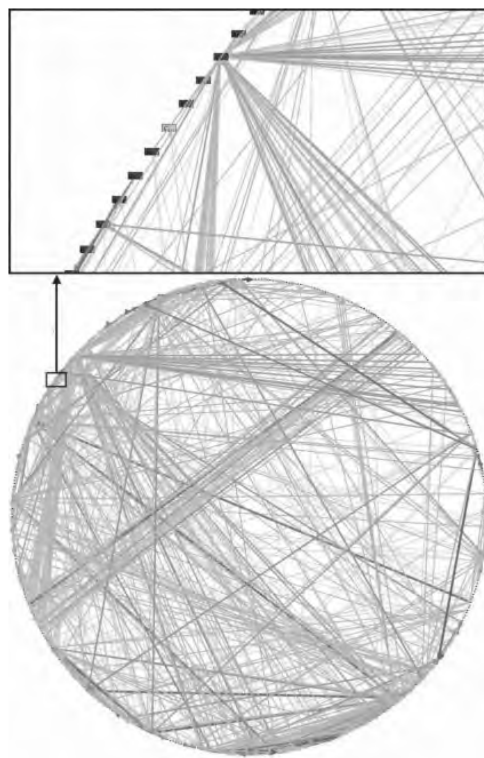

圖 P-1 真實世界的依賴關係圖（來自「Enterprise Dependency: Big Ball of Yarn」（*https://oreil.ly/dbGTW*），Alex Papadimoulis 著）

就像花園的自然狀態是雜草叢生，軟體的自然狀態就是大泥球。防止崩潰需要付出精力和指出方向。

幸好避免建立大泥球的技術並不複雜。

封裝與抽象

身為程式員，封裝與抽象是我們自然就會使用的工具，雖然我們不一定會使用準確的字眼。因為它們是本書反復出現的主題，所以請容我們為它們做詳細的介紹。

碁峰 www.gotop.com.tw

封裝涵蓋兩個密切相關的概念：簡化行為和隱藏資料。在接下來的說明中，我們使用第一種概念。當我們想要封裝行為時，我們會在程式中找出需要完成的任務，再給那個任務一個定義明確的物件或函式，我們將這個物件或函式稱為**抽象**。

看一下接下來的兩段 Python 程式：

用 *urllib* 進行搜尋

```
import json
from urllib.request import urlopen
from urllib.parse import urlencode

params = dict(q='Sausages', format='json')
handle = urlopen('http://api.duckduckgo.com' + '?' + urlencode(params))
raw_text = handle.read().decode('utf8')
parsed = json.loads(raw_text)

results = parsed['RelatedTopics']
for r in results:
    if 'Text' in r:
        print(r['FirstURL'] + ' - ' + r['Text'])
```

用 *requests* 進行搜尋

```
import requests

params = dict(q='Sausages', format='json')
parsed = requests.get('http://api.duckduckgo.com/', params=params).json()

results = parsed['RelatedTopics']
for r in results:
    if 'Text' in r:
        print(r['FirstURL'] + ' - ' + r['Text'])
```

這兩段程式做同一件事：將值送到 URL 來使用搜尋引擎 API，但是第二段比較易讀和瞭解，因為它是在更高層的抽象上操作的。

我們可以進一步確認我們希望程式執行的任務，並且為它命名，使用更高階的抽象來讓它更明確：

用 *duckduckgo* 模組進行搜尋

```
import duckduckgo
for r in duckduckgo.query('Sausages').results:
    print(r.url + ' - ' + r.text)
```

碁峯 www.gotop.com.tw

使用抽象來封裝行為是一種強大的工具，可讓程式更富表達性、更容易測試，且更容易維護。

在物件導向（OO）世界的文獻中，這種做法的典型稱謂是**責任驅動設計**（*responsibility-driven design*）（*http://www.wirfs-brock.com/Design.html*），它使用 *role*（**角色**）與 *responsibility*（**責任**）這種字眼，而不是 *task*（**任務**）。它的重點是從行為的角度來考慮程式碼，而不是從資料或演算法的角度[1]。

抽象與 ABC

在 Java 或 C# 等傳統 OO 語言中，你可能會使用抽象基礎類別（ABC）或介面來定義抽象。在 Python 中，你可以使用 ABC（有時我們也會這樣做），但也可以開心地使用鴨子定型。

抽象的意思只是「你所使用的東西的公用 API」——例如，一個函式名稱加上一些引數。

本書大部分的模式都涉及抽象的選擇，所以你會在各章看到許多範例。此外，第 3 章會專門討論選擇抽象的通用啟發式方法。

分層

封裝與抽象可以藉著隱藏細節和保護資料的一致性來提供協助，但我們也要留意物件與函式之間的互動。當函式、模組或物件使用另一個函式、模組或物件時，我們稱為一樣東西**依賴**（*depends on*）另一樣東西，這種依賴關係形成一種網路或圖（graph）。

在大泥球中，依賴關係是失控的（如圖 P-1 所示），改變圖中的節點非常困難，因為它可能會影響系統的許多其他部分。分層式架構是解決這種問題的方式之一，在分層式架構中，我們將程式碼分成分散的分類或角色，並且加入一些規則來規定哪些程式種類可以互相呼叫。

1 如果你看過 class-responsibility-collaborator（CRC）卡，它們的作用是相同的，從職責的角度來考慮問題，可以協助你決定如何進行分解。

碁峰 www.gotop.com.tw

這種做法最常見的例子就是圖 P-2 的**三層架構**。

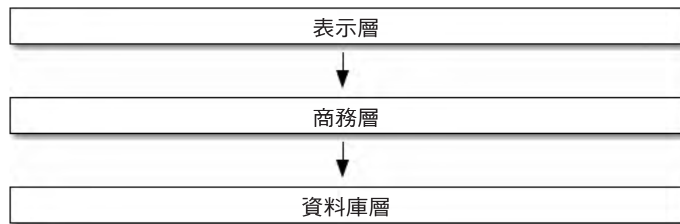

圖 P-2　分層架構

分層架構應該是商業軟體最常見的模式。在這種模型中，我們有用戶介面元件，它可能是個網頁、API 或命令列，這些用戶介面元件會和商務邏輯層溝通，商務邏輯層裡面有商務規則和工作流程，最後有一個資料庫層，它負責儲存和取出資料。

在本書接下來的內容中，我們會藉著遵守一條簡單的原則，系統性地徹底顛覆這個模型。

依賴反轉原則

或許你已經熟悉**依賴反轉原則**（DIP）了，因為它是 SOLID 裡面的 *D*[2]。

很遺憾，我們沒辦法像解釋封裝那樣，用三行小程式來解釋 DIP。但是，本書的第一部分實質上就是在整個 app 裡面實作 DIP 的範例，所以你將會看到一些具體的範例。

在此同時，我們也可以討論 DIP 的正式定義：

1. 高階模組不應該依賴低階模組，它們都應該依賴抽象。
2. 抽象不應該依賴細節，相反，細節應該依賴抽象。

但它們是什麼意思？我們來逐步說明。

高階模組是你的所屬機構真正在乎的程式碼。如果你在製藥公司工作，你的高階模組負責處理的就是患者與試驗；如果你在銀行工作，你的高階模組負責的是管理交易和匯兌。軟體系統的高階模組包含處理真實世界概念的函式、類別以及程式包。

2　SOLID 是 Robert C. Martin 提出的物件導向設計五大原則的縮寫，包括單一職責、對擴展開放，但是對修改封閉、Liskov 替換原則、介面隔離，以及依賴反轉。見 Samuel Oloruntoba 的「S.O.L.I.D: The First 5 Principles of Object-Oriented Design」（*https://oreil.ly/UFM7U*）。

碁峰 www.gotop.com.tw

相較之下，**低階模組**是你的機構不關心程式碼。你的 HR 部門應該不會對檔案系統或網路通訊端感興趣，你應該不會和財務團隊討論 SMTP、HTTP 或 AMQP。對非技術性關係人而言，這些低階概念既無趣且無關緊要，他們只在乎高階概念是否正確運作。如果薪資單可以準時發放，你的公司應該不會在乎你是在 cron job 還是在 Kubernetes 上運行一個暫時性函式。

*依賴*不一定是指*匯入*或*呼叫*，它是比較籠統的概念：一個模組*知道*或*需要*另一個模組。

我們已經談過*抽象*了，它們是封裝行為的簡化介面，就像我們的 duckduckgo 模組封裝了搜尋引擎的 API。

> *所有電腦科學問題都可以藉著加入額外的間接層來解決。*
>
> ——David Wheeler

所以 DIP 的定義的第一個部分的意思是，業務程式不應該依賴技術細節，相反，兩者都應該使用抽象。

為什麼？廣泛地說，因為我們希望能夠分別更改它們。高階模組必須能夠根據業務需求輕鬆地改變，低階模組（細節）在實務上難以改變，你可以想想藉著重構來改變函式名稱 vs. 定義、測試，和部署資料庫遷移（migration）來改變一個欄位名稱的情況。我們不希望因為商務邏輯與低階設施細節有密切的關係而拖慢修改速度。但是，類似的情況，在必要時*能夠*修改基礎設施細節（例如資料庫分片（sharding）），而且不需要改變商務層也非常重要。在它們之間加入抽象（著名的額外間接層）可讓兩者（更加）獨立於彼此進行改變。

第二個部分更神秘難懂了。「抽象不應該依賴細節」比較容易瞭解，但「細節應該依賴抽象」很難想像。我們在製作抽象時，該如何讓它不依賴被它抽象化的細節？在第 4 章，我們會用一個具體的範例清楚地說明這一點。

放置所有商務邏輯的地方：領域模型

但是在顛覆三層架構之前，我們要再討論一下中間層，即高階模組或商務邏輯。「設計」出錯最常見的原因是商務邏輯分散到 app 的各層，讓我們難以識別、瞭解與改變它。

第 1 章會介紹如何用 *Domain Model* 模式來建構商務層。第一部分其餘的模式將告訴你如何藉著選擇正確的抽象，以及持續執行 DIP，來保持領域模型的容易更改，而且不需要考慮低層。

碁峰 www.gotop.com.tw

第一部分

建立架構來支援領域模型的建構

> 大部分的開發者都沒有看過領域模型，只看過資料模型。
>
> ——Cyrille Martraire, *DDD EU 2017*

很多曾經與我們討論架構的開發者都認為情況還有挽回的餘地，因而不肯放棄，他們試圖挽救一個莫名其妙出了問題的系統，想把一些架構放到一團爛泥球裡面。他們知道商務邏輯不應該散布各處，卻不知道如何修正。

我們發現許多開發者被要求設計新系統時，都會立刻開始建立資料庫綱要（schema），事後再考慮物件模型，這就是一切問題的源頭。我們應該反其道而行，**先處理行為，再讓它驅動儲存需求**。畢竟，顧客並不在乎資料模型，他們在乎的是系統究竟會做什麼，否則他們只要使用試算表就好了。

本書的第一部分將探討如何使用 TDD 建構豐富物件模型（第 1 章），然後介紹如何讓那個模型與技術問題解耦。我們會展示如何建構不需要採用特定持久保存機制的程式，以及如何圍繞著領域建立穩定的 API，以方便積極地進行重構。

為此，我們要介紹四種重要的設計模式：

- Repository 模式，位於持久儲存的概念之上的抽象
- Service Layer 模式，明確地定義用例（use case）的開始與結束之處

碁峰 www.gotop.com.tw

- Unit of Work 模式，提供原子操作（atomic operation）
- Aggregate 模式，確保資料的完整性

如果你想要瞭解我們的結果，看一下圖 I-1，但如果你覺得它們都很陌生，不用擔心！我們會在第一部分逐一介紹圖中的各個方塊。

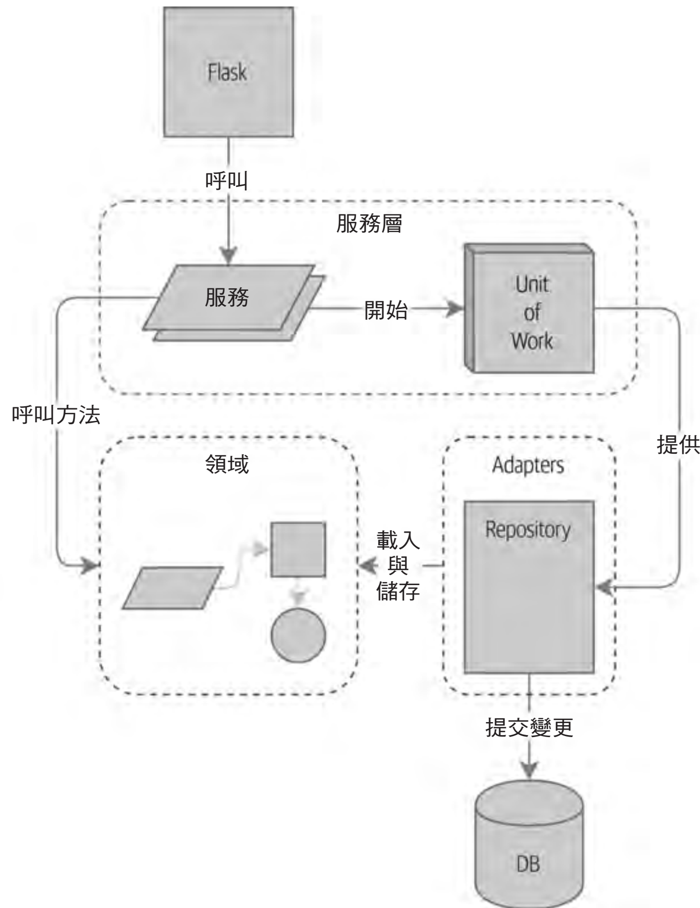

圖 I-1　在第一部分結束時，app 的元件圖

我們也會花一點時間討論耦合與抽象，用一個簡單的例子來展示我們如何以及為何選擇抽象。

碁峰 www.gotop.com.tw

本書的三個附錄是第一部分的延伸：

- 附錄 B 是關於我們的範例程式的基礎設施的點評：如何建構與執行 Docker 映像、在哪裡管理組態資訊，以及如何執行不同種類的測試。
- 附錄 C 是「東西好不好，用了才知道」類型的內容，展示將整個基礎設施（包括 Flask API、ORM 與 Postgres）換成完全不同的 I/O 模型（涉及 CLI 和 CSV）有多麼容易。
- 最後，如果你想要知道這些模式在你使用 Django、而不是 Flask 與 SQLAlchemy 時會是什麼樣子，那麼你應該很想看附錄 D。

碁峯
www.gotop.com.tw

第一章

建立領域模型

本章介紹如何以一種和 TDD 高度相容的方式，使用程式碼來模擬商務流程。我們將討論**為何**領域建模如此重要，並且介紹一些建立領域模型的重要模式：Entity、Value Object 與 Domain Service。

圖 1-1 是 Domain Model 模式的簡單視覺化占位圖案。本章會填入一些細節，隨著我們進入其他章節，我們也會圍繞著領域模型建構一些東西，不過你都會在核心發現這些形狀。

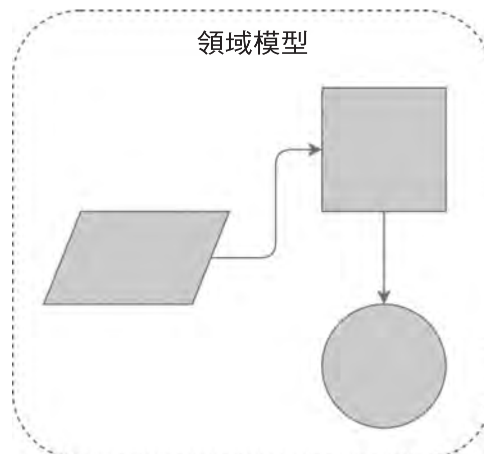

圖 1-1　領域模型的占位圖案

碁峰
www.gotop.com.tw

什麼是領域模型？

在引言中，我們使用**商務邏輯層**這個術語來描述三層架構的中間層。在本書接下來的內容中，我們會改用**領域模型**（*domain model*）這個術語。它是來自 DDD 社群的術語，比較能夠代表我們的本意（關於 DDD 的更多說明請見下一個專欄）。

領域只是**你試著解決的問題**的花俏說法。本書的作者目前任職於一家網路家具零售商，取決於你的系統，領域可能是進行採購、產品設計，或物流與配送。大部分的程式員都把時間花在改善商務流程或將它自動化上面，領域是這些流程支援的一組活動。

模型是某種流程或現象的對映，那些流程或現象描述了有用的特性。人類特別擅長在腦海中製作事物的模型。例如，有人對你丟一顆球時，你幾乎可以無意識地預測它的動向，因為你有一個「物體在空中移動的方式」的模型。你的模型不可能完美，人類對於接近光速或是真空之中的物體的直覺很差，因為我們的模型的設計從未涵蓋這些案例。但是這不代表模型是錯的，而是代表有些預測不在它的領域範圍之內。

領域模型是公司老闆思考商務活動的心智圖，所有商業人士都有這種心智圖——他們是人類思考複雜流程的方式。

你可以從別人使用的商務用語知道他們何時使用這種圖。一起製作複雜系統的人會自然而然地使用行話。

想像一下，你這位不幸的讀者突然之間被傳送到距離地球好幾光年之遠的星球，你和朋友及家人找到一艘外星飛船，想要從最基本的規則開始摸索，找到回家的方法。

在最初的幾天裡，你可能會隨意按下按鈕，但很快你就知道按鈕的功能是什麼，因此你們可以向彼此下達指令。你可能會說「按下閃爍的小東西旁邊的紅色按鈕，再把那根大桿子往雷達設備推」。

經過幾週之後，你們會使用更精確的言語來描述太空船的功能：「將貨艙的氧氣量提升三級」或「打開小推進器」。經過幾個月之後，你會用特定的語言來描述整個複雜的程序：「啟動著陸程序」或「準備曲速前進」，這個過程會自然地發生，不需要正式地建立共用的術語表。

碁峰 www.gotop.com.tw

> **這不是一本討論 DDD 的書，你應該去看 DDD 書**
>
> 領域驅動設計（DDD）推廣領域建模的概念[1]，藉著關注核心商務領域，它成功地改變大家設計軟體的方式。本書探討的許多架構模式都來自 DDD 傳統，架構模式包括 Entity、Aggregate、Value Object（見第 7 章）與 Repository（下一章）。
>
> 簡而言之，DDD 認為軟體最重要的事情在於它提供了實用的問題模型，如果我們做出正確的模型，軟體就可以提供價值，讓新事物得以發生。
>
> 如果我們把模型做錯了，它就會變成工作的障礙。在這本書，我們可以展示建構領域模型的基本知識，並且圍繞著它打造一個架構，讓模型盡量不被外部因素約束，如此一來，它就可以輕鬆地演進與更改。
>
> 但是 DDD 以及開發領域模型所需的流程、工具及技術還有許多可以探討的地方。雖然我們希望讓你盡量品嘗它的滋味，但也熱切希望你可以閱讀更合適的 DDD 書籍：
>
> - 原始的「藍皮書」，*Domain-Driven Design*，Eric Evans 著（Addison-Wesley Professional）
> - 「紅皮書」，*Implementing Domain-Driven Design*，Vaughn Vernon 著（Addison-Wesley Professional）

世俗的商務活動也是如此。商務關係人使用的術語濃縮了他們對於領域模型的理解，用一個單字或短語來歸納複雜的思想與過程。

當我們聽到商務關係人使用陌生的字眼，或是以特定的方式來使用術語時，我們應該注意聆聽更深的含義，並將他們得之不易的經驗寫入軟體。

本書將使用一個真實世界的領域模型，它是我們目前的工作所使用的模型。MADE.com 是一家成功的家具零售商，它的家具來自世界各地的製造商，在歐洲銷售。

當你購買沙發或咖啡桌時，我們必須找出最好的方法，從波蘭、中國或越南把商品送到你的客廳。

1　DDD 不是領域建模的起源，它來自 Eric Evans 引用的一本書，Rebecca Wirfs-Brock 與 Alan McKean 合著並在 2002 年出版的《*Object Design*》（Addison-Wesley Professional），這本書提出職責驅動設計，DDD 是這種設計法處理「領域」的一個特例。但 2002 年還不是最早的年份，OO 狂熱者會建議你再參考 Ivar Jacobson 與 Grady Booch 的著作，這個名詞早在 1980 年代中期就已經出現了。

碁峰 www.gotop.com.tw

在較高的層面上，我們有獨立的系統負責將購買庫存、銷售庫存給顧客，並且向顧客發貨。位於中間的系統必須協調流程，將庫存分配給顧客的訂單，見圖 1-2。

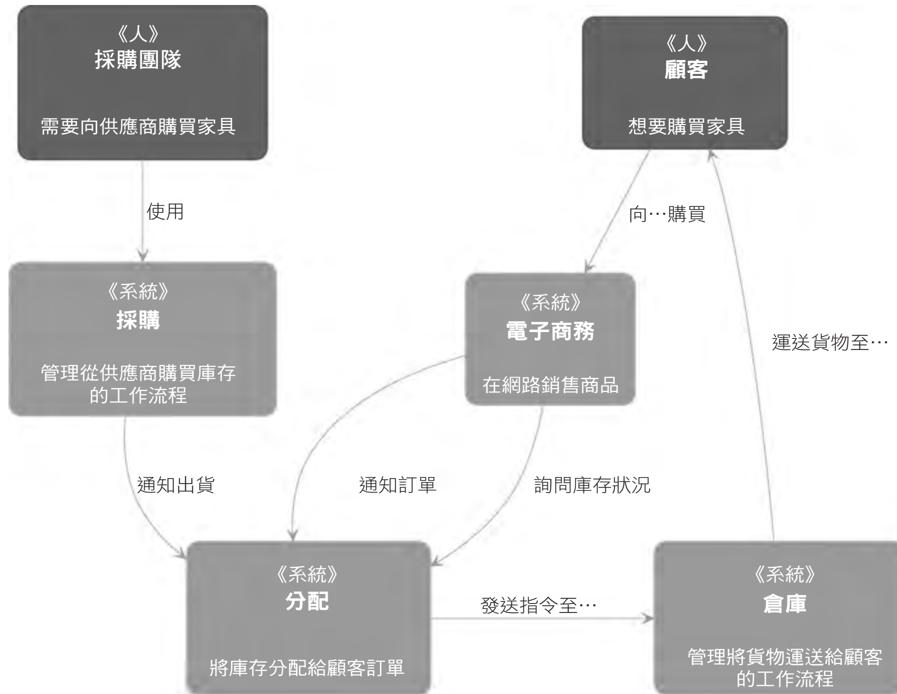

圖 1-2　分配服務的情境圖

出於本書的目的，我們假設公司決定實施一種令人期待的新方法來分配庫存，截至目前為止，公司一直都根據倉庫中實際的庫存量來提出庫存量與交貨時間。如果庫存沒了，公司就會將產品列為「缺貨」，直到廠商送來下一批貨物為止。

我們決定採取創新的做法：如果系統可以追蹤所有的發貨情況以及何時到貨，我們就可以將還在船上的貨物視為真正的庫存以及部分的庫存，只是需要較長的交貨時間。因此，缺貨的商品看起來會比較少，我們可以賣出更多東西，而且藉著降低國內倉庫的庫存量，公司還可以節省成本。

但是如此一來，為訂單配貨就不是只要減少倉庫系統的數量就好了，我們需要更複雜的配貨機制。是時候建立領域模型了。

碁峯 www.gotop.com.tw

探索領域語言

瞭解領域模型需要時間與耐心，還有便利貼。我們已經和商務專家進行了初步的討論，對於領域模型的最初精簡版本的術語和規則取得共識。在可能的情況下，我們要求提供具體的範例來描述每一條規則。

我們會用商業術語來敘述這些規則（DDD 稱之為**統一術語**（*ubiquitous language*））。我們幫物件選擇易記的代號，以便輕鬆地討論範例。

「關於配貨的注意事項」展示我們與領域專家就配貨問題進行討論時，可能會記下來的一些注意事項。

關於配貨的注意事項

產品（*product*）是用 *SKU* 來識別的，讀成「skew」，它是 *stock-keeping unit*（**庫存單位**）的簡稱。**顧客**（*customer*）會下**訂單**（*order*）。訂單是用**訂單編號**（*order reference*）來識別的，每一張訂單有多個**訂單行**（*order line*），每一行都有一個 *SKU* 與**數量**（*quantity*）。例如：

- 10 單位的 RED-CHAIR
- 1 單位的 TASTELESS-LAMP

採購部會訂購小**批**（*batch*）的庫存，一個**貨批**（*batch*）庫存有一個稱為**參考**（*reference*）的專屬 ID，一個 *SKU* 與一個**數量**（*quantity*）。

我們要將**訂單行**（*order line*）**分配給貨批**（*batch*）。為訂單行分配一個貨批之後，我們必須將那一批特定的存貨送到顧客的收貨地址。當我們將 x 單位的庫存分配給一個貨批時，就要將**存貨量**減 x。例如：

- 我們有一個包含 20 張 SMALL-TABLE 的貨批，並且將 2 張 SMALL-TABLE 分配給一個訂單行（order line）。
- 貨批應剩下 18 張 SMALL-TABLE。

如果庫存量少於訂單行（order line）的數量，就不能分配貨批（batch）。例如：

- 我們有一個包含 1 塊 BLUE-CUSHION 的貨批，以及訂購 2 塊 BLUE-CUSHION 的訂單行（order line）。
- 我們不能將該行分到給貨批。

碁峰 www.gotop.com.tw

我們不能分配同一行兩次。例如：

- 我們有一個包含 10 個 BLUE-VASE 的貨批，並且幫一個購買 2 個 BLUE-VASE 的訂單行（order line）配貨。
- 如果我們再次為這個訂單行（order line）分配同一批貨，貨批的剩餘數量必須仍然是 8。

如果貨批正在運送中，它會有一個 *ETA*（**預計到達時間**），否則貨批應該在**倉庫庫存**（*warehouse stock*）中。我們會優先分配倉庫庫存，再分配運送中的貨批。我們按照 ETA 的順序來分配運送中的貨批。

使用測試領域模型

這本書不會告訴你 TDD 如何運作，但我們想要告訴你如何透過這次商務訪談來建構模型。

給讀者的習題

何不親自解決這個問題？編寫一些單元測試，看看你能不能用簡潔的程式來描述這些商務規則的本質。

你可以在 GitHub 找到一些占位（placeholder）單元測試（*https://github.com/cosmicpython/code/tree/chapter_01_domain_model_exercise*），但你也可以直接從零開始，或隨意組合 / 重寫它們。

這是我們的第一個測試：

「配貨」的第一個測試程式（*test_batches.py*）

```
def test_allocating_to_a_batch_reduces_the_available_quantity():
    batch = Batch("batch-001", "SMALL-TABLE", qty=20, eta=date.today())
    line = OrderLine('order-ref', "SMALL-TABLE", 2)

    batch.allocate(line)

    assert batch.available_quantity == 18
```

碁峰 www.gotop.com.tw

這個單元測試的名稱指出我們想要看到的系統行為，類別與變數的名稱來自商業術語。當我們讓非技術人員閱讀這段程式時，他們可以認同這段程式正確地描述了系統的行為。

這是符合我們需求的領域模型：

貨批（*batch*）的第一版領域模型（*model.py*）

```
@dataclass(frozen=True)  ❶❷
class OrderLine:
    orderid: str
    sku: str
    qty: int


class Batch:
    def __init__(
        self, ref: str, sku: str, qty: int, eta: Optional[date]  ❷
    ):
        self.reference = ref
        self.sku = sku
        self.eta = eta
        self.available_quantity = qty

    def allocate(self, line: OrderLine):
        self.available_quantity -= line.qty  ❸
```

❶ `OrderLine` 是一個沒有行為的不可變資料類別[2]。

❷ 為了保持簡潔，大多數的程式都不會列出 import，希望你可以猜到它是用 `from dataclasses import dataclass` 取得的，`typing.Optional` 與 `datetime.date` 也是如此。如果你想要再次檢查任何內容，你可以在各章的分支找到完整的程式碼（例如 chapter_01_domain_model（*https://github.com/python-leap/code/tree/chapter_01_domain_model*））。

❸ 在 Python 世界中，型態提示仍然是個有爭議的問題。對領域模型而言，它們有時可以幫助釐清或記錄預期的參數是什麼，使用 IDE 的人通常會很感激有它們。不過，或許你會認為使用它時付出的可讀性代價太高了。

2 在之前的 Python 版本中，我們可能會使用具名 tuple。你也可以使用 Hynek Schlawack 製作的 attrs（*https://pypi.org/project/attrs*）。

碁峰 www.gotop.com.tw

這段程式很簡單：Batch 只包含一個整數 available_quantity，我們會在配貨時減少這個值。雖然寫了這麼多程式只是為了將一個數字減去另一個數字，但是我們認為準確地建構領域模型是有好處的[3]。

我們來寫一些新的失敗測試（failing test）：

測試可以分配什麼東西的邏輯（*test_batches.py*）

```
def make_batch_and_line(sku, batch_qty, line_qty):
    return (
        Batch("batch-001", sku, batch_qty, eta=date.today()),
        OrderLine("order-123", sku, line_qty)
    )


def test_can_allocate_if_available_greater_than_required():
    large_batch, small_line = make_batch_and_line("ELEGANT-LAMP", 20, 2)
    assert large_batch.can_allocate(small_line)

def test_cannot_allocate_if_available_smaller_than_required():
    small_batch, large_line = make_batch_and_line("ELEGANT-LAMP", 2, 20)
    assert small_batch.can_allocate(large_line) is False

def test_can_allocate_if_available_equal_to_required():
    batch, line = make_batch_and_line("ELEGANT-LAMP", 2, 2)
    assert batch.can_allocate(line)

def test_cannot_allocate_if_skus_do_not_match():
    batch = Batch("batch-001", "UNCOMFORTABLE-CHAIR", 100, eta=None)
    different_sku_line = OrderLine("order-123", "EXPENSIVE-TOASTER", 10)
    assert batch.can_allocate(different_sku_line) is False
```

這裡沒有什麼特別的事情。我們重構了測試組，以免重複使用同樣幾行程式來建立貨批（batch）以及用一行程式建立相同 SKU；我們也為新方法 can_allocate 寫了四個簡單的測試程式。注意，我們使用的名稱同樣可以和領域專家使用的語言相應，我們取得共識的案例已經被直接寫成程式了。

我們可以直接實作它，編寫 Batch 的 can_allocate 方法：

模型中的新方法（*model.py*）

```
    def can_allocate(self, line: OrderLine) -> bool:
        return self.sku == line.sku and self.available_quantity >= line.qty
```

3 還是你認為程式不夠多？用某種方式檢查 OrderLine 裡面的 SKU 符合 Batch.sku 如何？我們將一些關於驗證的考量保留給附錄 E。

碁峰 www.gotop.com.tw

到目前為止，我們可以藉著增加與減少 Batch.available_quantity 來直接管理實作，但是當我們進入 deallocate() 測試程式時，我們將被迫使用比較巧妙的解決方案：

這項測試將需要比較巧妙的模型（*test_batches.py*）

```
def test_can_only_deallocate_allocated_lines():
    batch, unallocated_line = make_batch_and_line("DECORATIVE-TRINKET", 20, 2)
    batch.deallocate(unallocated_line)
    assert batch.available_quantity == 20
```

在這個測試中，我們斷言（assert）除非我們已經幫某一行（line）分配一個貨批，否則無法將那一行的配貨取消。為此，Batch 必須知道有哪幾行已經被配貨了。我們來看一下實作：

現在領域模型可追蹤配貨（*model.py*）

```
class Batch:
    def __init__(
        self, ref: str, sku: str, qty: int, eta: Optional[date]
    ):
        self.reference = ref
        self.sku = sku
        self.eta = eta
        self._purchased_quantity = qty
        self._allocations = set()  # 型態：Set[OrderLine]

    def allocate(self, line: OrderLine):
        if self.can_allocate(line):
            self._allocations.add(line)

    def deallocate(self, line: OrderLine):
        if line in self._allocations:
            self._allocations.remove(line)

    @property
    def allocated_quantity(self) -> int:
        return sum(line.qty for line in self._allocations)

    @property
    def available_quantity(self) -> int:
        return self._purchased_quantity - self.allocated_quantity

    def can_allocate(self, line: OrderLine) -> bool:
        return self.sku == line.sku and self.available_quantity >= line.qty
```

圖 1-3 是以 UML 繪出模型的情況。

碁峯 www.gotop.com.tw

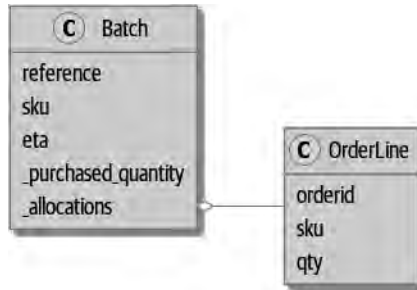

圖 1-3　以 UML 敘述模型

我們有了進展了！現在貨批（batch）可以追蹤一組已配貨的 OrderLine 物件。在配貨時，如果存貨量足夠，我們就直接將它加入集合（set）。現在 available_quantity 是一個算出來的 property：將採購量減去配貨量。

沒錯，我們還可以做很多事，雖然 allocate() 與 deallocate() 會沉默地失敗令人不安，但我們已經掌握基本狀況了。

順便說一下，讓 ._allocations 使用 set 可讓我們輕鬆地處理最後一項測試，因為在 set 裡面的項目都是唯一的：

最後一個貨批測試！（*test_batches.py*）

```
def test_allocation_is_idempotent():
    batch, line = make_batch_and_line("ANGULAR-DESK", 20, 2)
    batch.allocate(line)
    batch.allocate(line)
    assert batch.available_quantity == 18
```

此時，或許你會說這個領域模型太簡單了，不值得使用 DDD（甚至物件導向）。在真實世界中，有時會突然出現許多商務規則與邊緣案例，顧客可能會指定一個特定的到貨日期，這意味著我們可能不會將它們分配給最早的貨批。有些 SKU 沒有在貨批裡面，而是要視需求直接向供應商訂購，所以它們使用不同的邏輯。我們可以根據顧客的位置，只分配一部分的庫存，或是指派位於他們的區域的物流，如果本國的 SKU 缺貨，我們也樂於從其他地區的倉庫交貨，諸如此類。真正的公司知道複雜性會如何以更快的速度增加！

我們將這個簡單的領域模型當成更複雜的東西的占位（placeholder）模型，在本書的其餘部分會擴展這個簡單的領域模型，並且插入 API、資料庫與試算表。我們會看到堅守封裝原則以及謹慎地進行分層如何協助避免大泥球。

碁峰 www.gotop.com.tw

更多型態與更多型態提示

如果你真的想要盡情使用型態提示，你可以使用 `typing.NewType` 來包裝原始型態：

這就太過分了，*Bob*

```
from dataclasses import dataclass
from typing import NewType

Quantity = NewType("Quantity", int)
Sku = NewType("Sku", str)
Reference = NewType("Reference", str)
...

class Batch:
    def __init__(self, ref: Reference, sku: Sku, qty: Quantity):
        self.sku = sku
        self.reference = ref
        self._purchased_quantity = qty
```

舉例來說，這可讓型態檢查器確保我們不會將 `Sku` 傳到期望收到 `Reference` 的地方。

這種做法的好壞是有待商榷的[4]。

資料庫很適合值物件

我們在上面的程式中自由地使用 `line`，不過什麼是 line？在我們的商務語言中，一份訂單（*order*）有多行（*line*）項目，每一行都有一個 SKU 與一個數量。我們可以想像一個包含訂單資訊的 YAML 檔案可能長這樣：

以 *YAML* 描述訂單

```
Order_reference: 12345
Lines:
  - sku: RED-CHAIR
    qty: 25
  - sku: BLU-CHAIR
    qty: 25
  - sku: GRN-CHAIR
    qty: 25
```

4 這種做法很糟糕，拜託不要。——Harry

碁峰 www.gotop.com.tw

注意，雖然訂單有個用來識別的**參考**（*reference*），但**行**（*line*）沒有（即使我們在 `OrderLine` 類別加入訂單參考，它也不是可以唯一性地識別行（line）本身的東西）。

如果我們有個商務概念包含資料但沒有身分，我們通常會用 *Value Object*（**值物件**）模式來表示它。**值物件**是可以用它保存的資料來獨一無二地識別它自己的領域物件，我們通常讓它是不可變的：

OrderLine 是個值物件

```
@dataclass(frozen=True)
class OrderLine:
    orderid: OrderReference
    sku: ProductReference
    qty: Quantity
```

資料類別（或具名 tuple（namedtuple））提供一種很棒的特性 —— **值相等性**（*value equality*），意思是「如果兩個訂單行的 `orderid`、`sku` 與 `qty` 相同，它們就是相等的」。

更多值物件範例

```
from dataclasses import dataclass
from typing import NamedTuple
from collections import namedtuple

@dataclass(frozen=True)
class Name:
    first_name: str
    surname: str

class Money(NamedTuple):
    currency: str
    value: int

Line = namedtuple('Line', ['sku', 'qty'])

def test_equality():
    assert Money('gbp', 10) == Money('gbp', 10)
    assert Name('Harry', 'Percival') != Name('Bob', 'Gregory')
    assert Line('RED-CHAIR', 5) == Line('RED-CHAIR', 5)
```

這些值物件符合我們在真實世界中對於它們的值如何運作的直覺，我們討論的 £10 紙幣究竟是**哪一張**一點都不重要，因為它們都有相同的價值，同樣的，如果兩個名字的名與姓都相符，它們就是相等的，如果兩行（line）有相同的顧客訂單、產品代碼與數量，它們就是等效的。不過，我們仍然可以讓值物件具備複雜的行為，事實上，我們經常提供針對值的操作，例如數學運算子：

碁峰 www.gotop.com.tw

用值物件來運算

```
fiver = Money('gbp', 5)
tenner = Money('gbp', 10)

def can_add_money_values_for_the_same_currency():
    assert fiver + fiver == tenner

def can_subtract_money_values():
    assert tenner - fiver == fiver

def adding_different_currencies_fails():
    with pytest.raises(ValueError):
        Money('usd', 10) + Money('gbp', 10)

def can_multiply_money_by_a_number():
    assert fiver * 5 == Money('gbp', 25)

def multiplying_two_money_values_is_an_error():
    with pytest.raises(TypeError):
        tenner * fiver
```

值物件與實體

我們用訂單行（order line）的訂單 ID、SKU 與數量來唯一性地識別它，如果我們改變其中一個值，就會得到一個新行（line）。值物件是只用資料來識別而且沒有長期的身分的物件。那貨批（batch）呢？它是用參考（reference）來識別的。

我們使用**實體**（*entity*）來代表具備長期身分的領域物件。我們在上一頁建立一個 Name 類別作為值物件，如果我們取出姓名 Harry Percival 並且修改一個字母，我們就會得到新的 Name 物件 Barry Percival。

Harry Percival 不等於 Barry Percival 應該是再明顯不過的事情：

名字本身不能改變…

```
def test_name_equality():
    assert Name("Harry", "Percival") != Name("Barry", "Percival")
```

但如果 Harry 是人呢？有人可能改變他的名字、婚姻狀況，甚至性別，但我們仍然認為他是同一個人，因為人與姓名不同，具有持久性的**身分**：

碁峰 www.gotop.com.tw

但人可以！

```
class Person:

    def __init__(self, name: Name):
        self.name = name


def test_barry_is_harry():
    harry = Person(Name("Harry", "Percival"))
    barry = harry

    barry.name = Name("Barry", "Percival")

    assert harry is barry and barry is harry
```

實體與值不同，具備**身分相等性**（*identity equality*）。我們可以改變它們的值，並且仍然將它們視為同一個東西。在我們的例子中，貨批（batch）就是實體，我們可以將訂單行（line）分配給一個貨批，或改變它送達的日期，但它仍然是同一個實體。

在程式中，我們通常藉著對實體實作相等運算子來明確地指出這一點：

實作相等運算子（*model.py*）

```
class Batch:
    ...

    def __eq__(self, other):
        if not isinstance(other, Batch):
            return False
        return other.reference == self.reference

    def __hash__(self):
        return hash(self.reference)
```

Python 用 `__eq__` 魔術方法定義該類別在使用 `==` 運算子時的行為[5]。

我們也要想一下對實體與值物件而言，`__hash__` 該如何運作。這個方法是當你將物件加入 set，或將它當成 dict 的鍵來使用時，Python 用來控制物件行為的魔術方法，你可以在 Python 的文件找到更多資訊（*https://oreil.ly/YUzg5*）。

對值物件而言，hash 應該根據所有的值屬性，我們應該確保物件是不可變的，我們可以對資料類別指定 `@frozen=True` 來輕鬆地做到這一點。

5 `__eq__` 方法唸成「dunder-EQ」。至少有些人是這樣唸的啦！

碁峰 www.gotop.com.tw

對實體而言，最簡單的選項是指定 hash 是 None，代表該物件是不可 hash 的，而且（舉例）不能在 set 中使用。如果因為某些原因，你認為真的需要用實體來進行 set 或 dict 操作，hash 應該根據「可以長時間定義實體的唯一身分」的屬性，例如 .reference。你也要試著讓**那個**屬性是唯讀的。

這是一個麻煩的領域，你不應該只修改 __hash__ 而不修改 __eq__。如果你不確定做得對不對，推薦你一本參考書，我們的技術校閱 Hynek Schlawack 寫的「Python Hashes and Equality」（*https://oreil.ly/vxkgX*）是很好的起點。

並非所有東西都必須是物件：領域服務函式

我們已經做了一個代表貨批的模型了，但我們真正需要做的事情是將訂單列分配給代表所有庫存的貨批的特定集合。

> 有時，它根本不是個東西。
>
> —Eric Evans,《*Domain-Driven Design*》

Evans 曾經探討無法被自然地放在實體或值物件裡面的 Domain Service 操作的概念[6]。用一組貨批為一個訂單行（line）配貨的東西聽起來很像函式，我們可以利用 Python 是多範式（multiparadigm）語言這個事實，直接將它做成函式。

我們來看一下如何測試這種函式：

測試領域服務（*test_allocate.py*）

```
def test_prefers_current_stock_batches_to_shipments():
    in_stock_batch = Batch("in-stock-batch", "RETRO-CLOCK", 100, eta=None)
    shipment_batch = Batch("shipment-batch", "RETRO-CLOCK", 100, eta=tomorrow)
    line = OrderLine("oref", "RETRO-CLOCK", 10)

    allocate(line, [in_stock_batch, shipment_batch])

    assert in_stock_batch.available_quantity == 90
    assert shipment_batch.available_quantity == 100
```

6 領域服務與服務層的服務不一樣，雖然它們有密切的關係。領域服務代表一種商務概念或程序，而服務層的服務代表 app 的一個用例。通常服務層會呼叫領域服務。

碁峰 www.gotop.com.tw

```
def test_prefers_earlier_batches():
    earliest = Batch("speedy-batch", "MINIMALIST-SPOON", 100, eta=today)
    medium = Batch("normal-batch", "MINIMALIST-SPOON", 100, eta=tomorrow)
    latest = Batch("slow-batch", "MINIMALIST-SPOON", 100, eta=later)
    line = OrderLine("order1", "MINIMALIST-SPOON", 10)

    allocate(line, [medium, earliest, latest])

    assert earliest.available_quantity == 90
    assert medium.available_quantity == 100
    assert latest.available_quantity == 100


def test_returns_allocated_batch_ref():
    in_stock_batch = Batch("in-stock-batch-ref", "HIGHBROW-POSTER", 100, eta=None)
    shipment_batch = Batch("shipment-batch-ref", "HIGHBROW-POSTER", 100, eta=tomorrow)
    line = OrderLine("oref", "HIGHBROW-POSTER", 10)
    allocation = allocate(line, [in_stock_batch, shipment_batch])
    assert allocation == in_stock_batch.reference
```

我們的服務可能長這樣：

獨立的領域服務函式（*model.py*）

```
def allocate(line: OrderLine, batches: List[Batch]) -> str:
    batch = next(
        b for b in sorted(batches) if b.can_allocate(line)
    )
    batch.allocate(line)
    return batch.reference
```

Python 的魔術方法可讓我們以典型的 Python 風格使用模型

無論你喜不喜歡使用上面程式中的 `next()`，我們都相信你同意對著貨批 list 使用 `sorted()` 是很棒、很典型的 Python 風格。

為了讓它生效，我們在領域模型實作了 `__gt__`：

魔術方法可以表達領域語意（*model.py*）

```
class Batch:
    ...

    def __gt__(self, other):
        if self.eta is None:
            return False
```

碁峰 www.gotop.com.tw

```
        if other.eta is None:
            return True
        return self.eta > other.eta
```

真可愛。

例外也可以表達領域概念

我們還要探討最後一個概念：例外也可以表達領域概念。在與領域專家討論時，我們知道，我們可能因為**缺貨**而無法為訂單配貨，我們可以使用**領域例外**來描述這個概念：

測試缺貨例外（*test_allocate.py*）

```
def test_raises_out_of_stock_exception_if_cannot_allocate():
    batch = Batch('batch1', 'SMALL-FORK', 10, eta=today)
    allocate(OrderLine('order1', 'SMALL-FORK', 10), [batch])

    with pytest.raises(OutOfStock, match='SMALL-FORK'):
        allocate(OrderLine('order2', 'SMALL-FORK', 1), [batch])
```

領域建構回顧

領域建模

這是你的程式碼最接近商務、最有可能改變的部分，也是你為公司帶來最大價值的地方。把它寫得容易瞭解與修改。

區分實體與值物件

值物件是用它的屬性來定義的，將它做成不可變型態通常是最好的做法。一旦你改變 Value Object 的屬性，它就代表不同的物件了。相較之下，實體的屬性可能隨著時間改變，但它仍然是同一個實體。你必須定義**可以**唯一性地識別實體的東西（通常是某種名稱或參考欄位）。

並非所有東西都必須是物件

Python 是一種多範式語言，所以在程式中以函式來呈現「動詞」。對於 `FooManager`、`BarBuilder` 與 `BazFactory`，我們通常可以用比較富表達性與易讀的 `manage_foo()`、`build_bar()` 或 `get_baz()` 來取代。

碁峰 www.gotop.com.tw

這是使用最佳 *OO* 設計原則的時刻

複習 SOLID 原則以及所有其他優秀的原則，例如「has a vs. is-a」、「優先使用組合而非繼承」等。

你也要考慮一致性界限與 *aggregate*（集合體）

不過那是第 7 章的主題。

我們不想用太多實作來困擾你，不過要注意的是，我們會謹慎地幫統一術語之中的例外取名字，就像我們為實體、值物件與服務所做的那樣。

發出領域例外（*model.py*）

```
class OutOfStock(Exception):
    pass


def allocate(line: OrderLine, batches: List[Batch]) -> str:
    try:
        batch = next(
        ...
    except StopIteration:
        raise OutOfStock(f'Out of stock for sku {line.sku}')
```

圖 1-4 是視覺化的最終結果。

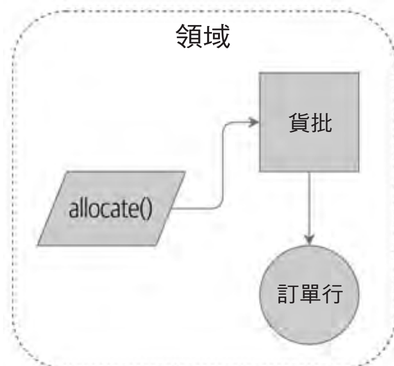

圖 1-4　本章結束時的領域模型

目前這樣大概就夠了！我們有個可讓第一個用例使用的領域服務了。但首先，我們需要資料庫…

碁峯 www.gotop.com.tw