前言

Git 是一個由 Linus Torvalds 建立的免費、開源、分散式版本控制系統，它在操作上要求的技術能力低，但又夠靈活和強大，得以支持複雜、大規模、分散式的軟體開發專案。

這本書的目標，是展示充分利用 Git，並輕鬆管理 Git 儲存庫的方法，最後讓你學習到 Git 的理念、基本概念以及追蹤內容、協作和團隊項目管理的中階到進階技巧。

這本書適合誰？

本書撰寫時，主要受眾是軟體工程師，如開發人員、基礎架構工程師、DevOps 等；因此，使用的大部分概念和範例都與軟體開發行業的日常例行工作和任務有關。然而，因為 Git 夠強大，可以追蹤各種領域的內容，例如資料科學、圖形設計和書籍撰寫等，例如事實證明：本書撰寫時，就使用 Git 作為底層版本控制系統，來追蹤評論和編輯！所以，無論職稱或熟練程度，只要使用 Git 作為版本控制系統，都能在這些頁面中找到價值。

重要技巧

已具有任何版本控制系統的經驗、了解它的受眾群和目標，在閱讀本書時，將有助於理解 Git 的工作原理，並且能夠建立知識基礎。你應該多少熟悉任一命令列工具的使用，如 Unix shell，並且了解基本的 shell 命令，因為本書中的範例和討論將使用很多命令列指令。對程式概念有基本理解也會加到分。

我們在 macOS 和 Ubuntu 環境下開發這些範例，也應該可以在其他平台，如 Debian、Solaris 和使用命令列工具安裝的 Git for Windows 上執行，只是可能會有一些小變化。

範例中的一些練習可能因為需要系統層級操作，而必須在機器上擁有根（root）權限。碰到這種情況，需要清楚理解根授權的操作責任。

本次版本的新增內容

第三版採取一種全新的模塊化方法來解構 Git 的主題。會從基礎知識和 Git 的基本原理開始介紹，然後逐步介紹中階指令，好協助有效地補充日常的開發工作流程，最後以進階的 `git` 命令和概念總結，讓你成為 Git 在底層內部運作機制方面的熟練使用者。

這個版本還有另一個改變：增加更多插圖來解釋複雜的 Git 概念，提供更容易理解的心智模型。同時，還強調最新版本的 Git 特點，並以舉例和提示，幫助改善目前的分散式開發工作流程。

本書導航

根據讀者對 Git 的熟悉程度和使用經驗，這版本組織不同的類別。雖然一樣將章節依序分類，好讓讀者逐漸提升 Git 技能，但每個章節內的各個主題都設計成可以單獨閱讀，或者按照順序逐漸深入研究的一系列主題。

我們努力在每一章套用一致架構和教學方法來傳授概念，這裡鼓勵你多花些時間內化這個格式；這將有助於你在未來任何時候使用這本書時，都可作為方便的參考。

如果你在閱讀本書時仍有其他要事在忙，並且不知道要以怎樣的順序開始，請不要擔心，表格 P-1 將幫助指引你到我們認為可以在最短時間內獲得最多知識的章節。

表 P-1　分類矩陣

	Git 基礎思維	Git 基礎知識	Git 中階技巧	Git 進階技巧	小提示和技巧
軟體工程	X	X	X	X	X
資料科學家	X	X	X		X
圖形設計師	X	X			X
學術界	X	X			X
內容作者	X	X			X

安裝 Git

為了強化書中所教授的課程，強烈鼓勵你在開發機器上練習範例程式碼片段。為了按部就班操作，需要在選擇的平台上安裝 Git。安裝步驟會根據操作系統版本而有所不同，可見附錄 B 提供的安裝說明。

關於包容性語言的說明[譯註]

這裡還想強調一下這些例子的另一個重要觀點，那就是對科技界的多元化和包容性抱持強烈信念，提高意識是我們認真對待的責任，就從使用「主要」（main）一詞來表示預設分支名稱作為開始。

遺漏

由於其活躍的社群基礎，Git 不斷進化。即使在撰寫此版本時，另一個新版本的 Git 2.37.1 版本已經發布供商業使用。本書不打算遺漏資訊，但這是在介紹一個不斷變化的技術時，無法避免的現實。

我們特意選擇不涵蓋 Git 的所有核心命令和選項，以便更加專注於常見且常用的命令。同樣地，因數量眾多，我們也未涵蓋所有可用的 Git 相關工具。

儘管有這些遺漏，相信這本書仍能為你提供充實基礎，並在你需要的時候準備好深入 Git 領域。如果需要，可以參考 Git 的版本發布說明檔案（*https://oreil.ly/R2nn4*），查看詳細版本更改列表。

本書編排慣例

本書中使用下列排版慣例：

斜體字（*Italic*）

表示新詞彙、網址、電子郵件地址、檔案名和檔案副檔名。中文以楷體表示。

譯註　過往版本控制系統中，最主要的分支會使用「master」為預設名稱。然而，這個詞彙也有「主人」的意思，可能在無意中帶有種族歧視或壓迫性意涵，基於包容性以及促進社會正義，特將預設分支改為「main」。

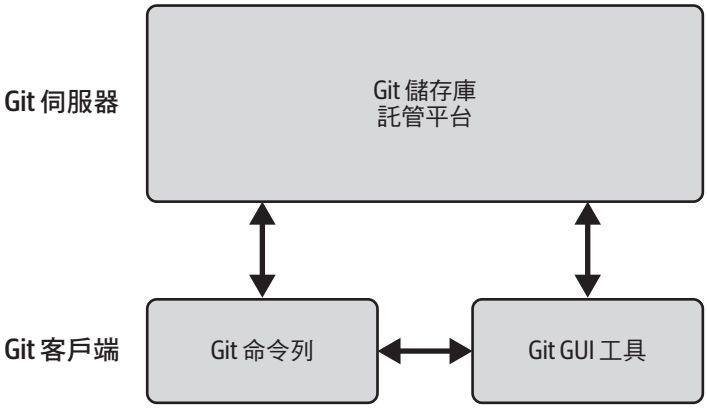

圖 1-1　Git 元件概述

使用 Git 時，典型的設置包括一個 Git 伺服器和 Git 客戶端，可能可以不使用伺服器，但這會增加在共同協作的設置中維護和管理版本庫時的複雜性，並且會使一致性更加困難；第 11 章會再次討論這個問題。Git 伺服器和客戶端的工作方式如下：

Git 伺服器

Git 伺服器能夠讓協作更容易，因為它確保即將工作的儲存庫有一個集中且可靠的真實來源。Git 伺服器也是存放遠端 Git 儲存庫之處；根據一般慣例，儲存庫應該擁有專案最新且穩定的來源。你可以選擇安裝和配置自己的 Git 伺服器，或者不管這些負擔，選擇在可靠的第三方託管網站例如 GitHub、GitLab 和 Bitbucket，託管 Git 儲存庫。

Git 客戶端

Git 客戶端會與本地儲存庫互動，可以藉由 Git 命令列或 Git GUI 工具與 Git 客戶端互動。安裝和配置 Git 客戶端後，就能夠存取遠端儲存庫，在本地版本儲存庫工作並將更改推送回 Git 伺服器。如果你是 Git 新手，建議從使用 Git 命令列開始；熟悉了解日常操作使用所需的 Git 命令子集，然後選擇你喜歡的 Git GUI 工具。

這種方法的原因在於，在某種程度上，Git GUI 工具往往提供代表期望結果的術語，而這些術語可能不是 Git 標準命令的一部分。舉例來說，一個名為 `sync` 的工具選項，掩蓋了使用兩個或更多的 `git` 命令進行底層串連式操作，以達到期望結果。如果因為某些原因，而在命令列上輸入 `sync` 子命令，可能會得到以下混亂的輸出：

```
$ git sync

git: 'sync' is not a git command. See 'git --help'.

The most similar command is
       svn
```

`git sync` 不是一個有效的 `git` 子指令。為了確保本地工作副本與遠端 Git 儲存庫的更改同步，需要執行這些指令的組合：`git fetch`、`git merge`、`git pull` 或者 `git push`。

可以使用很多工具來處理。有些 Git GUI 工具很豪華，並能藉由插件模型擴展，這樣可以選擇連接和利用在流行的第三方 Git 代管站點上提供的功能。雖然藉由 GUI 工具學習 Git 很方便，但這裡還是將重點放在 Git 命令列工具上，以便討論範例和程式，因為這將建立紮實的基礎知識，從而使 Git 更為熟練。

Git 特色

既然已經概述 Git 的組成部分，現在來了解一下 Git 的特性。了解 Git 的獨特之處，能使你輕鬆從集中式版本控制思維，轉換為分散式版本控制思維，或是我們喜歡的說法，「以 Git 思考」：

Git 將修訂更改以快照形式儲存

第一個需要重新學習的概念是 Git 儲存檔案多個修訂版本的方式。與其他版本控制系統不同，Git 不將修訂更改作為一系列修改，即通常稱為增量的追蹤；相反的，它在特定時間點上，對於儲存庫狀態的更改進行快照。在 Git 術語中，這稱為提交（*commit*），可以想像為捕捉下一個時間點，如同照片一般。

Git 為了本地開發增強

在 Git 中，你在本地開發機器上的儲存庫副本上工作，這稱為本地儲存庫，或者是 Git 伺服器上遠端儲存庫的複製品。你的本地儲存庫將擁有資源以及在這些資源上的修訂變更快照，它們都位於同一個位置。Git 把這些連結快照的集合稱為儲存庫提交歷史，簡稱儲存庫歷史，這能讓人在離線環境中工作，因為 Git 不需要與 Git 伺服器保持連接才能對變更進行版本控制，你能夠在分散團隊中進行大型、複雜項目的同時，又理所當然地不損害版本控制操作的效率和性能。

Git 是絕對的

絕對（*definitive*），意味著 **git** 命令是明確的，它會等待你提供指示以及執行該指示的時機。舉例來說，Git 不會自動同步本地儲存庫與遠端儲存庫之間的變更，也不會自動將修訂的快照保存到本地儲存庫的歷史紀錄中。每個操作都需要明確的命令或指示告訴 Git 該做的事，包括添加新的提交、修復現有的提交、將本地儲存庫中的變更推送到遠端儲存庫，甚至取得遠端儲存庫中的新變更。簡而言之，一切操作都需要有意義，這還包括讓 Git 知道要追蹤的檔案，因為 Git 不會自動添加新檔案以進行版本控制。

Git 旨在加強非線性開發

Git 能透過分支操作來構思並嘗試實現各種不同功能，為專案找到可行的解決方案，可以在專案主要、穩定的程式碼庫中並行工作，這種方法稱為**分支**（*branching*），非常普遍，可以確保主要開發線的完整性，並防止任何可能破壞它的意外更改。

在 Git 中，分支的概念可說是輕量且不昂貴，因為在 Git 中，分支只是一個指向一系列相關提交中最新提交的指標；對於每個建立的分支，Git 都會追蹤它的一系列提交，你可以在本地切換分支；然後，Git 將專案的狀態恢復到建立指定分支的快照最近時刻。當你決定將任何分支的更改合併到主要開發線時，Git 能夠藉由應用第 6 章會討論到的技術，來結合這些系列提交。

由於 Git 提供許多新意，請記住其他版本控制系統的概念和實踐在 Git 中可能會有不同運作方式，或者完全不適用。

Git 命令列介面

Git 的命令列介面很好使用，它的設計是將儲存庫的完全控制權交到你手中。因此，有很多種方式可以完成同樣的事情。經由聚焦在每日工作的重要命令，可以簡化並更深入地學習。

想開始，只需輸入 **git version** 或 **git --version**，以確保機器預載了 Git。應該會看到和下面類似的輸出：

```
$ git --version
git version 2.37.0
```

管理分支

除了預設的分支外，一個儲存庫在專案的壽命期間可能會有很多不同的分支；然而，每次當然都只能在一個分支中工作，這就是當前的或活動的分支。活動的分支有助於確定工作目錄中正在取出的檔案：換句話說，它反映當前正在開發中檔案的狀態。

此外，當前分支在 Git 命令中通常是一個隱含的運算元，如合併操作的目標。預設情況下，每次初始化一個新的 Git 儲存庫時，主分支是活動分支，但可以建立其他分支，並且使任何分支成為當前分支。

圖 3-1 呈現包含兩個分支的提交圖。請注意，HEAD 指向分支名稱並保持更新，指向分支中最新的提交。在操縱分支時請牢記這個圖形結構，因為它會加強你對 Git 分支底層優雅且簡單的物件模型理解，第 4 章會詳細探索提交。

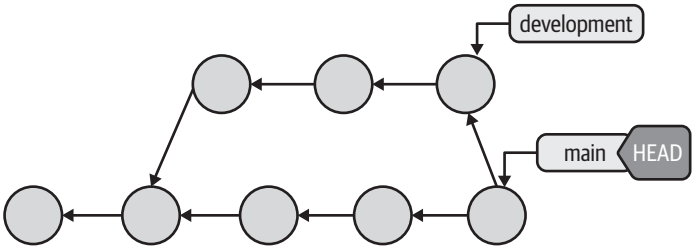

圖 3-1　帶有分支的提交圖

在分支中工作

在儲存庫中建立一個分支，可以使儲存庫的內容在多個方向上分裂，每個方向表示一個分支。假設儲存庫中有兩個分支：預設分支和分離的開發線，則建立的每個提交都只應用於其中一個分支，即標記為活動分支的分支。

遵循分支命名規則和指南，如果分支名稱簡潔並反映目的情境，將會很有幫助。分支名稱會一直引用分支上的最新提交，也稱為該分支的*尖端*或 *HEAD*。根據技術定義，分支名稱是對特定提交的簡單指標；因此，隨著對正在開發的活動分支添加新的提交，分支名稱會以增量方式向前移動。圖 3-2 能說明這個概念。

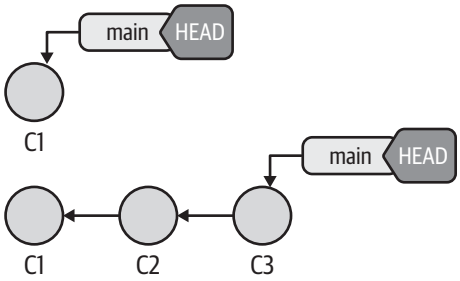

圖 3-2　作為動態指標的分支名稱

由於分支名稱是動態的，並且指向分支上一系列提交中的最新提交或 HEAD 提交，因此 Git 不會追蹤分支來自於哪個具體提交，或從哪次提交開始。如果要引用分支中的舊提交，需要提供明確的提交物件 ID 或相對名稱，例如 dev~5，這表示從當前提交開始，將分支名稱指標往回移動 5 個提交。因為分支代表著專案中的一個穩定點，如果需要在一個分支中的特定提交靜態參考指標，可以明確地指定一個輕量標籤或者註解標籤名稱。圖 3-3 說明了這個概念。

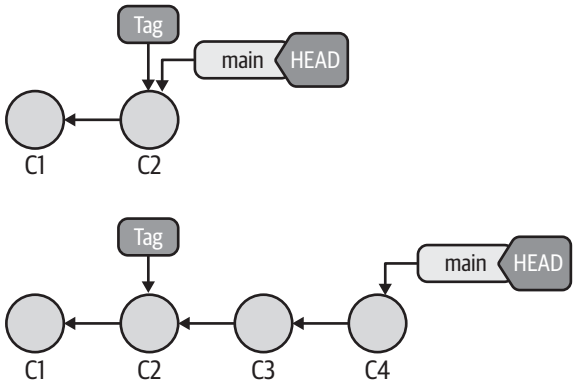

圖 3-3　分支名稱 vs. 標籤名稱

一個分支包含足夠的提交紀錄，可以重新建立專案的歷史，從分支開始到項目開始的全部過程。

分支與標籤

分支和標籤看起來可能很相似，但它們有不同的用途。

標籤可以用作臨時書籤，即輕量標籤；或作為固定的參考點，即註釋標籤，以區分產品的特定版本發布時間。因此，標籤是靜態的，總是指向 Git 儲存庫中的特定提交物件。

分支指向並隨著在開發過程中進行的每個提交而移動，對分支的每個引用，即分支名稱，將遵循儲存庫中發展路徑的每個分歧線。因此，分支是動態的。

所以，何時該使用標籤，何時又該使用分支呢？決策最終還是取決於你和專案的開發政策。然而，應該考慮的一個關鍵區分特徵是：引用是靜態且不可變的，還是動態的？如果是前者，就應該使用標籤，後者就應該使用分支。

可以使用相同的名稱來命名分支和標籤，但這樣做，會需要使用它們的完整參照名稱來區分，例如，可以使用 `refs/tags/v1.0` 和 `refs/heads/v1.0`。建議分支和標籤盡量避免使用相同的名字，除非有充分的理由。Git 決定要使用哪個參考的基準，來自於 gitrevisions 文件中解釋的規則（*https://oreil.ly/Mvm6S*）。

在圖 3-4 中，`development` 分支指向最新的提交 `Z`，即分支中最新的提交。如果想重現儲存庫在 `Z` 提交時的狀態，需要從 `Z` 回到最初提交 `A` 的所有可接觸到提交。圖中寬線突出的部分涵蓋除了 `S`、`G`、`H`、`J`、`K` 和 `L` 之外的每個提交，請注意，提交 `W` 到 `F` 的箭頭，和提交 `X` 到 `R` 的箭頭表示相應分支之間的合併操作，第 6 章會再次提到合併操作。

每個分支名稱，以及每個分支上的提交內容，都屬於你的本地儲存庫。然而，將儲存庫提供給他人時，可以選擇發布一個或多個分支以及相關的提交內容；發布分支必須明確執行，詳細內容可見第 11 章的說明。此外，如果複製儲存庫，分支名稱和這些分支上的開發內容，也都將成為新複製儲存庫的一部分。

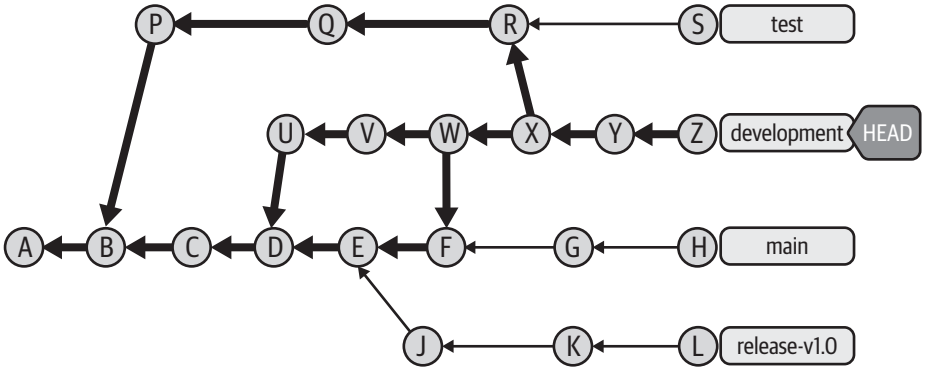

圖 3-4　從開發可接觸到的提交

建立分支

Git 的設計是為了支援任意複雜的分支結構，這包括從現有分支中建立新的分支，以及從同一個提交分岔成多個分支。

建立一個新的分支時，它一定會基於儲存庫中的一個現有提交，可以是當前的最新提交或 HEAD 提交，也可以是明確使用提交 SHA（提交物件 ID）來引用的不同提交。

一個分支可能有短暫或長期存在的時間，它的壽命完全是你的決定。在它的壽命範圍內，根據開發需求，特定的分支名稱可能會多次添加和刪除。

建立新分支的基本命令形式如下：

```
$ git branch branchname start-point
```

如果不指定 *start-point*，預設起點將是目前作用中分支的尖端或 HEAD 提交。換句話說，預設是在你目前工作的位置開啟一個新分支。

請注意，`git branch` 命令僅將分支名稱引入儲存庫，不會將工作目錄更改為使用新分支。該命令只是在給定的提交位置建立一個新的命名分支。這也代表工作目錄中的檔案將不會發生變化，也不會隱含更改分支情境。當然，執行指令時也不會建立新的提交。

只有在切換到分支後，才能開始在該分支上工作，「切換（取出）分支」一節，即第 66 頁將會示範。這樣做的主要原因在於建立新分支時，頂點或 HEAD 目前仍指向建立分支時的活動分支，如圖 3-5 說明的概念。

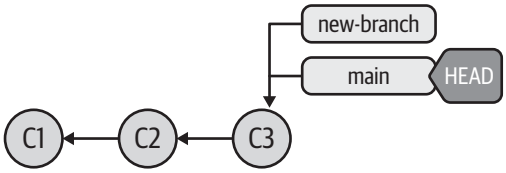

圖 3-5　HEAD、活動分支和新分支

假設專案分支策略反映其產品發行版本，而你的任務是修復某個發行版本中的一個錯誤，使用 *start-point* 參數建立一個新的錯誤修復分支，從指定發行版本分支中最近的提交開始可能很方便。這樣能夠切換工作目錄，以反映代表該發行版本的點上的專案狀態。

舉例來說，可以指定一個名為 `rel-2.3` 的分支，來代表專案的產品版本 2.3；也可以按照以下方式，對指定版本進行漏洞修復的分支：

```
$ git branch bugs/fix-1311 rel-2.3
```

start-point 參數接受分支名稱、提交 SHA 或標籤名稱。使用提交 SHA 可自由指向任何分支開發線上的一系列提交，換句話說，可以從單個提交點在多個方向上進行分岔，從共同起點探索替代解決方案：

```
$ git branch bugs/fix-1311 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

列出分支名稱

`git branch` 命令會列出在儲存庫中找到的分支名稱：

```
$ git branch
bugs/pr-1311
development
* main
```

這個例子顯示了三個分支，目前正在工作目錄中取出的活動分支以星號（`*`）區隔，另外還顯示了兩個分支，`bugs/pr-1311` 和 `development`。

在沒有其他參數的情況下，會只列出本地儲存庫中的分支。正如第 11 章將介紹的，儲存庫中可能還有其他遠端追蹤分支，可以藉由提供 `git branch` 命令加上 `-r` 參數選項來一一列出，也可以使用 `-a` 參數選項列出本地（*local*）和遠端（*remote*）分支。

第三部分

中階技能

第三部分將準備在使用 Git 儲存庫時所需的中階技能。本書的這一部分會開始討論提交，然後最後介紹遠端儲存庫的概念，同時分享一些管理儲存庫的良好實踐方法。

儲存庫的歷史由提交組成，有時候出於一些正當理由，可能需要修改提交歷史。但在可以修改提交之前，要先知道如何找到它們，第 8 章就將教你找到特定提交和它們的後設資料方法；然後，第 9 章將分享各種修改提交的技巧，其中一些具有破壞性，但也有不會破壞儲存庫歷史的辦法。請記住，你在本章學到的技巧，不僅僅局限於修改提交的操作，還可以幫助除錯，或理解變更出現在儲存庫中的過程。接下來的第 10 章，將討論儲藏（stash）臨時變更和取消儲藏的方法，以及討論記錄每個引用（ref），或引入提交上支持操作的引用日誌（reflog）。

最後，第 11 章將協助你了解，在與多個需要存取你的儲存庫的人合作時，協作與分享更改最佳方式。我們還將提供一些指引，指導你發布儲存庫，並為分散式開發建立良好的結構。

檢索不同的歷史

為了讓 Git 在兩個不同的歷史之間合併，兩者必須存在於同一個儲存庫的兩個不同分支中。純粹是本地開發分支的分支，是它們已經存在於同一個儲存庫中的特殊（退化）情況。

然而，如果因為拷貝而在不同儲存庫中存在不同歷史，則遠端分支必須藉由抓取操作來導入到你的儲存庫中，可以直接執行 `git fetch` 命令，或以 `git pull` 命令的一部分來執行該操作。無論是哪種情況，抓取操作會將遠端的提交，也就是此處的 `C` 和 `D`，合併到你的儲存庫中。結果圖 11-6 所示。

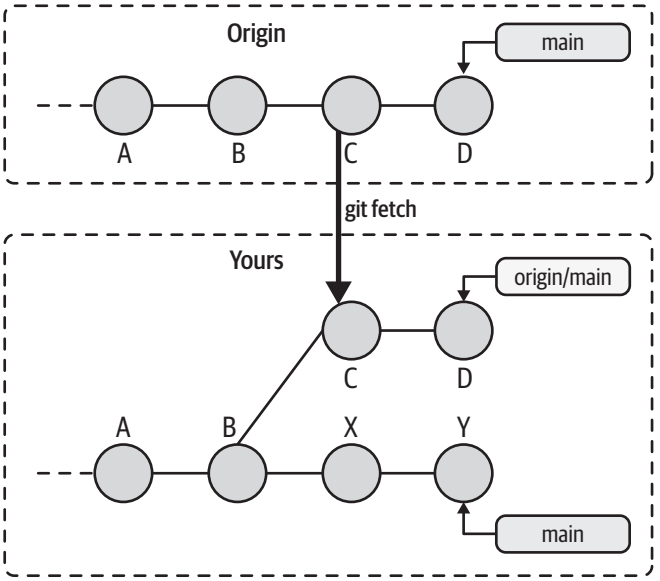

圖 11-6　抓取不同的歷史

以提交 `C` 和 `D` 引入的不同歷史，完全不會改變由 `X` 和 `Y` 所代表的歷史；這兩個不同的歷史現在同時存在於你的儲存庫中，形成一個更複雜的圖形。你的歷史由 `main` 分支表示，遠端歷史由 `origin/main` 遠端追蹤分支表示。

合併歷史紀錄

現在兩個歷史都存在於同一個儲存庫中，只需要將 `origin/main` 分支合併到 `main` 分支，就可以統一它們。

合併操作可以藉由直接的 `git merge origin/main` 命令，或以 `git pull` 請求的第二步來啟動。不管是哪一個方法，合併操作的技巧都與第 6 章描述的完全相同。

圖 11-7 顯示在合併成功，將提交 D 和 Y 的兩個歷史合併到新的合併提交 M 後，儲存庫中的提交圖譜。`origin/main` 的引用仍然指向 D，因為它沒有改變，但是 `main` 已更新為合併提交 M，以指示合併是進入 `main` 分支的；這是新提交的位置。

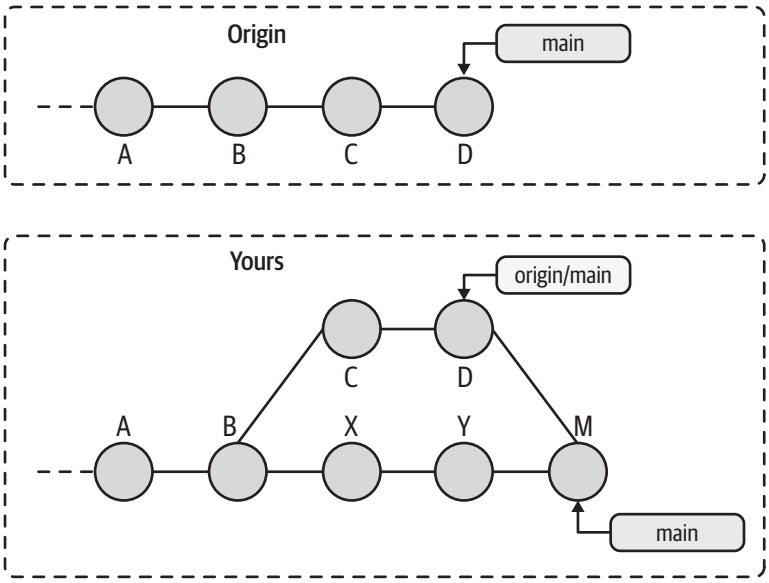

圖 11-7　合併歷史紀錄

合併衝突

偶爾會出現不同歷史之間的合併衝突。無論合併結果，抓取過程仍然發生，遠端儲存庫中的所有提交仍然存在於你的追蹤分支上。

可以選擇正常的解決合併，如第 6 章所述；或者選擇中止合併，並使用命令 `git reset --hard ORIG_HEAD`，將 `main` 分支重置為先前的 `ORIG_HEAD` 狀態。在此範例中，這樣做將會把 `main` 分支移至先前的 `HEAD` 值 Y，並更改你的工作目錄以符合這個狀況。它還將使 `origin/main` 保持在提交 D 上。

可以重讀第 82 頁「引用（Refs）與符號引用（Symrefs）」，進一步理解 `ORIG_HEAD`；同時也可以參考第 142 頁「中斷或重新開始合併」，來了解它的使用方式。

第四部分

進階技能

在接下來的章節中，將討論一些由分散式合作者分享和開發的儲存庫所傳播變更的替代方法，以幫助你了解，我們會討論一個簡單而有效的機制，藉由修補程式分享這些變更，且藉由電子郵件發送。

同時，也會開始展示擴展一些 Git 操作的標準執行流程可能性，這是日常工作中需要實作客製執行工作流程時可以受益的一點。儘管流行的 Git 主機平台現在支援現代開發工作流程的功能，但了解這種技巧，可以在需要無法藉由提供標準 Git 操作來獲得的工作流程功能時，增加額外技巧。

第 15 章在需要將專案模組化，並在不同的 Git 儲存庫中管理時提供幫助，會討論兩種常用的方法，並藉由一些範例展示技術實作，以引用相依的 Git 儲存庫。第 16 章中將進一步推進你在中階部分學到的技能，會分享一些優雅的技巧，幫助你將相關變更製作為原子提交，並教你尋找遺失的提交，和展示 `git rev-list` 命令的使用方法。本章的亮點是 `git-filter-repo` 工具，它很萬能，可以修復和操作所有儲存庫的提交和歷史紀錄，強烈建議你在這一部分多花點時間。