

為你自己
Gift 學

高見龍 @ 五倍紅寶石

目錄

一、入門篇

為你自己學 Git	1.1
寫在最前面 - 為你自己學 Git !	1.2
什麼是 Git ? 為什麼要學習它 ?	1.3
與其它版本控制系統的差異	1.4

二、環境安裝

安裝在 Windows 作業系統	2.1
安裝在 Mac OSX 作業系統	2.2
安裝在 Linux 作業系統	2.3
圖形化介面工具	2.4

三、終端機/命令提示字元

終端機及常用指令介紹	3.1
超簡明 Vim 操作介紹	3.2

四、設定 Git

使用者設定	4.1
其它方便的設定	4.2

五、開始使用 Git

新增、初始 Repository	5.1
把檔案交給 Git 控管	5.2
工作區、暫存區與儲存庫	5.3
檢視紀錄	5.4

【狀況題】如何在 Git 裡刪除檔案或變更檔名？	5.5
【狀況題】修改 Commit 紀錄	5.6
【狀況題】追加檔案到最近一次的 Commit	5.7
【狀況題】新增目錄？	5.8
【狀況題】有些檔案我不想放在 Git 裡面...	5.9
【狀況題】檢視特定檔案的 Commit 紀錄	5.10
【狀況題】等等，這行程式誰寫的？	5.11
【狀況題】啊！不小心把檔案或目錄刪掉了...	5.12
【狀況題】剛才的 Commit 後悔了，想要拆掉重做...	5.13
【狀況題】不小心使用 hard 模式 Reset 了某個 Commit，救得回來嗎？	5.14
【冷知識】HEAD 是什麼東西？	5.15
【狀況題】可以只 Commit 一個檔案的部份的內容嗎？	5.16
【冷知識】那個長得很像亂碼 SHA-1 是怎麼算出來的？	5.17
【超冷知識】在 .git 目錄裡有什麼東西？Part 1	5.18
【超冷知識】在 .git 目錄裡有什麼東西？Part 2	5.19

六、使用分支

為什麼要使用分支？	6.1
開始使用分支	6.2
對分支的誤解	6.3
合併分支	6.4
【狀況題】為什麼我的分支都沒有「小耳朵」？	6.5
【常見問題】合併過的分支要留著嗎？	6.6
【狀況題】不小心把還沒合併的分支砍掉了，救得回來嗎？	6.7
另一種合併方式（使用 rebase）	6.8
合併發生衝突了，怎麼辦？	6.9
【冷知識】為什麼大家都說在 Git 開分支「很便宜」？	6.10
【冷知識】Git 怎麼知道現在是在哪一個分支？	6.11
【狀況題】我可以從過去的某個 Commit 再長一個新的分支出來嗎？	6.12

七、修改歷史紀錄

【狀況題】修改歷史訊息	7.1
【狀況題】把多個 Commit 合併成一個 Commit	7.2
【狀況題】把一個 Commit 拆解成多個 Commit	7.3
【狀況題】想要在某些 Commit 之間再加新的 Commit	7.4
【狀況題】想要刪除某幾個 Commit 或是調整 Commit 的順序	7.5
Reset、Revert 跟 Rebase 指令有什麼差別？	7.6

八、標籤

使用標籤	8.1
【冷知識】標籤跟分支有什麼不一樣？	8.2

九、其它常見狀況題

【狀況題】手邊的工作做到一半，臨時要切換到別的任務	9.1
【狀況題】不小心把帳號密碼放在 Git 裡了，想把它刪掉...	9.2
【狀況題】如果你只想要某個分支的某幾個 Commit？	9.3
【冷知識】怎麼樣把檔案真正的從 Git 裡移掉？	9.4
【冷知識】你知道 Git 有資源回收機制嗎？	9.5
【冷知識】斷頭(detached HEAD) 是怎麼一回事？	9.6

十、遠端共同協作 - 使用 GitHub

GitHub 是什麼？	10.1
Push 上傳到 GitHub	10.2
Pull 下載更新	10.3
【狀況題】怎麼有時候推不上去...	10.4
從伺服器上取得 Repository	10.5
【常見問題】Clone 跟 Pull 指令有什麼不一樣？	10.6
與其它開發者的互動 - 使用 Pull Request (PR)	10.7
【狀況題】怎麼跟上當初 fork 專案的進度？	10.8
【狀況題】怎麼刪除遠端的分支？	10.9
【狀況題】聽說 git push -f 這個指令很可怕，什麼情況可以用它呢？	10.10
使用 GitHub 免費製作個人網站	10.11

【冷知識】一定要有 GitHub 才能得到別人更新的檔案嗎？

10.12

十一、使用 Git flow

Git Flow 是什麼？為什麼需要這種東西？

11.1

使用 Git Flow

11.2

十二、團隊協作

【狀況題】啊，我還沒開分支就 Commit 下去了！

12.1

【狀況題】咦？這個問題是從什麼時候就有的？

12.2

【狀況題】測試還要跑好久，但老闆叫我去修別的分支的問題...

12.3

十三、冷知識

【冷知識】手工初始化 .git 目錄

13.1

【冷知識】Stash 是什麼？

13.2

【冷知識】~ 跟 ^ 有什麼不同？

13.3

為你自己學 Git

「為你自己學 Git」，如其標題，學習不需要為公司、不需要為長官、同事、不需要為別人，只為你自己。

本書所有內容是我在五倍紅寶石培訓課程以及線上課程「[人生不能重來，但 GIT 可以](#)」所用到的教材，同時本書的大部份內容也會在 <https://gitbook.tw> 網站同步發表。若發現內容有誤或有任何問題，歡迎直接來信告知（eddie@5xruby.tw）。

本電子書之內容與實體出版之紙本書內容相同，惟頁面編排方式以及頁碼與紙本書會有些差異。

高見龍 @ [五倍紅寶石](#)

寫在最前面 - 為你自己學 Git !

為什麼要寫這本書

在周星馳的《大話西遊》電影裡，至尊寶拿著月光寶盒大喊一聲「波若波羅密」便可穿越時空，回到過去救他的娘子；或像動畫《多啦 A 夢》房間抽屜裡時光機，隨時可以回到過去解救世界。Git 這個工具雖然沒辦法真的讓我們可以穿越時空（有的話請一定要讓我知道，我要回到過去買大樂透），但對電腦工作者來說，它就像時光機一樣的神奇，可以讓你回到指定的時間點，或是救回不小心被刪除的檔案。

Git 看起來很容易學（事實上也不算難學），但這只是表象，實際上 Git 是一款讓人一開始覺得很容易學但卻很難精通的工具。坊間的參考書籍或網路資料大多會教大家從終端機指令來學習 Git，這讓不少有興趣想學習的新手打退堂鼓。

我也認同 Git 指令很重要，因為那是整個 Git 的基礎，所以學習在終端機視窗敲打、輸入 Git 指令是必經過程。但是如果可以搭配圖形介面工具（GUI 工具），可以讓這個學習曲線稍微緩和一些。所以本書除了 Git 指令介紹外的同時，也會使用圖形介面工具（本書使用 SourceTree）輔助說明，讓大家更容易上手。

因為個性的關係，在學習新事物的過程中如果有疑惑的地方，總是希望可以搞懂為止，否則知其然而不知其所以然，沒辦法真的把一門技術搞懂會痛苦得睡不著覺。也因為這樣，本書在撰寫的時候也發揮了我這個人囉嗦的專長，即使是很簡單的小地方，也希望可以儘量解釋清楚。希望可以不只可以教大家如何用（How），也能讓大家知道在用什麼（What），以及為什麼（Why）要這樣用。

雖然本書是以中文撰寫，但專有名詞大多還是英文，這些名詞或我會儘量使用英文來表示。除了每個人的翻譯可能不一樣或是翻譯之後沒有原文貼切之外，例如「Commit」、

「Repository」、「Fetch」等字，最重要的一點，是希望各位能儘早習慣這些英文，因為實際在業界工作時，很多第一手的資料都是英文的，早點習慣英文對大家絕對是有幫助的。

誰適合本書

只要你對 Git 這個工具有興趣都適合。

如果你平日的工作已有在使用 Git，本書大部份的內容對你來說應該是相對的輕鬆；但即使已經平日有在使用 Git 的人，也可從本書學到一些「本來以為 Git 是這樣，但其實是那樣」的觀念。

本書內容

會包括以下內容

- 常用 Git 指令介紹。
- 各種 Git 的常見問題及使用情境。
- 如何修改 Git 的歷史紀錄。
- 如何使用 GitHub 與其它人一起工作。
- 以及一般平常工作用不到但對觀念建立有幫助的冷知識。

你需要準備什麼？

- 一台可以工作的電腦（不限定作業系統）
- 這樣就夠了 :)

如何使用這本書

本書主要分以下幾部份：

1. 環境安裝與設定
2. 開始使用 Git
3. 使用分支
4. 使用標籤
5. 修改歷史紀錄
6. 其它常見狀況題
7. 使用 GitHub
8. 使用 Git flow

雖然每個章節的內容多少都還跟前面的章節有關，但也不一定要從第一章開始依序閱讀（當然這也是一種方式），可依自己需要跳過部份章節。

使用版本

本書使用的 Git 版本為 2.21.0，您可使用 `git --version` 指令來檢視您目前所使用的 Git 版本：

```
$ git --version
git version 2.21.0
```

如果是不同的版本，一樣的指令或參數可能會有不同的執行結果。

程式碼慣例

在學習、使用 Git 的時候，會有很多機會需要在終端機（Terminal）模式下輸入指令，例如：

```
$ git add index
```

或是這樣：

```
$ git commit -m "init commit"
[master (root-commit) 5d47270] init commit
 2 files changed, 1 insertion(+)
 create mode 100644 config/database.yml
 create mode 100644 index.html
```

在最前面的 `$` 符號是系統提示字元，意思是告訴大家這是一個需要在終端機環境下手動輸入的指令，而它的下一行則是這個指令執行的結果。實際在輸入指令的時候請不要跟著輸入 `$`，不然可能會出現 `command not found` 的錯誤訊息。

程式範例及錯誤更正

本書所有的範例在 Git 版本 2.21.0 以及 MacOS 10.13 版本作業系統的環境下均已測試可正常執行，部份範例可在我的 GitHub 帳號<https://github.com/kaochenlong>取得。但隨著軟體的版本演進，或是作業系統的不同，範例程式執行的結果可能會有些微的差異（甚至是錯誤）。若有任何問題，或是有哪邊寫錯，還請各位先進不吝來信、留言指教。

最後，希望各位會喜歡這本書，一起來學習 Git 這個看似好學但又不容易學得好的有趣工具 :)

關於學習

輸入指令可能很嚇人，但它很重要！

對學習 Git 的新手來說，打開終端機、輸入 Git 指令是件嚇人的事。

即使有像 SourceTree 或 GitHub Desktop 之類方便的圖形介面工具可使用，我個人仍是強烈建議一定要了解 Git 的運作原理。而透過輸入、執行 Git 指令，正是最容易可以了解 Git 運作的方法之一。

不要害怕輸入指令、不要害怕那些看起來很嚇人的訊息，不然即使有圖形介面工具軟體，也可能不知道按了這個按鈕之後會發生什麼事，而導致不正確的使用 Git。

觀念很重要

很多人，包括我自己也是，在一開始學習 Git 的時候，只覺得它就是簡單的學習 `git add` 跟 `git commit` 之類的基本操作指令罷了。但其實這就有如跟冰山一角，沈在水底下的比在浮在水面上的還多，Git 的運作方式遠比這些指令來得複雜得多。所以，如果可以建立正確的觀念，在遇到問題的時候就比較不會疑惑、不知道該用什麼指令來解決。

關於我

高見龍，這看起來有點像武俠小說的名字不是筆名，而是我父母給我的本名。目前是兩個小朋友的爸爸，是個愛寫程式而且希望可以寫一輩子程式的阿宅。

- 五倍紅寶石創辦人及負責人
- Blog: <https://kaochenlong.com>
- Facebook: <https://www.facebook.com/eddiekao>
- Twitter: <https://twitter.com/eddiekao>
- Github: <https://github.com/kaochenlong>
- Email: eddie@5xruby.tw

若發現本書內容有誤或有任何問題，歡迎直接來信，或到本書網站 <https://gitbook.tw> 留言 :)

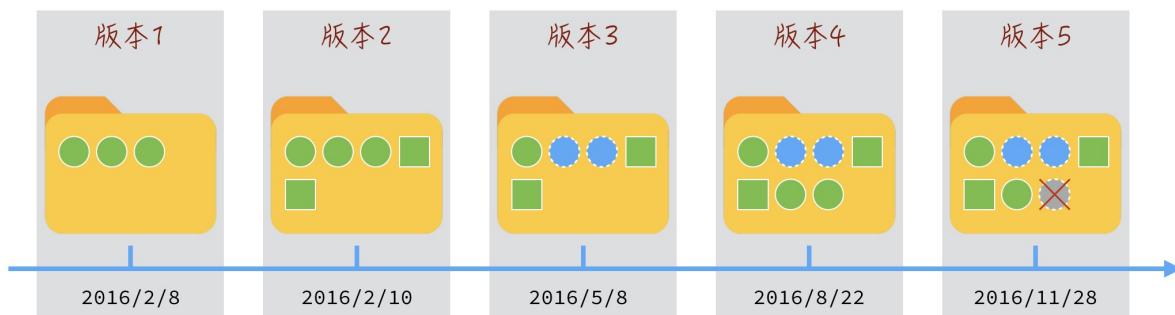
什麼是 Git？為什麼要學習它？

什麼是 Git？

如果你問大部份正在使用 Git 這個工具的人「什麼是 Git」，他們大多可能會回答你「Git 是一種版本控制系統」，專業一點的可能會回答你說「Git 是一種分散式版本的版本控制系統」。

「版本控制系統」的英文是「Version Control System」，但這個回答，對沒接觸過的新手來說，有講跟沒講差不多。到底什麼是「版本」？要「控制」什麼東西？什麼又是「分散式」？

不管你是不是工程師，只要你是電腦工作者，你每天的工作可能都是每天新增、編輯、修改許多檔案。舉個例子來說，你可能是一名人資部門主管，你有一個叫做 `resume` 的目錄，裡面專門用來存放面試者的資料。



讓我們來看圖說故事。

如圖所示，隨著時間的變化，一開始這個 `resume` 目錄裡只有 3 個檔案，過兩天增加到 5 個。不久之後，其中的 2 個被修改了，過了三個月後又增加到 7 個，最後又刪掉了 1 個，變成 6 個。這每一個「`resume` 目錄的狀態變化」，不管是新增或刪除檔案，亦或是修改檔案內容，都稱之為一個「版本」，例如上圖圖例的版本 1 ~ 5。而所謂的「版本控制系統」，就是指會幫你記錄這些所有的狀態變化，並且可以像搭乘時光機一樣，隨時切換到過去某個「版本」時候的狀態。

簡單的說，Git 就像玩遊戲的時候可以儲存進度一樣。舉例來說，為了避免打頭目打輸了而損失裝備，又或是打倒頭目卻沒有掉落期望的珍貴裝備，你也許在每次要去打頭目之前之前記錄一下，在發生狀況的時候可以載入舊進度，再來挑戰一次。

為什麼要學習它？

先問大家個問題，大家平常怎麼整理或備份檔案？

以上面這張圖例來說，最傳統也是最方便的方式，便是使用 Ctrl+C 與 Ctrl+V 的「複製、貼上大法」，然後你可能就會看到像這樣的畫面：

```
✗ └─ resumes
    > └─ resume-2016-02-08
    > └─ resume-2016-02-10
    > └─ resume-2016-05-08
    > └─ resume-2016-08-22
    ✓ └─ resume-2016-11-28
        └─ eddie.md
        └─ john.md
        └─ kao.md
        └─ mary.md
        └─ sherly.md
        └─ tracy.md
    > └─ resume-bak
    > └─ resume-for-5xruby
```

雖然用肉眼就可以知道每個「版本」的用途，但，你能一下子就講出在 `resume-2016-05-08` 這個目錄裡的那兩個修改過的檔案改了什麼內容嗎？`resume-2016-08-22` 跟 `resume-2016-11-28` 這兩個目錄有什麼不一樣的地方嗎？又，那個 `resume-bak` 跟其它的目錄有什麼不同？最麻煩的是，如果這個目錄是跟其它人一起共同，不管是有心無意，要是檔案被其它同事覆蓋掉了，也完全不知道該怎麼處理。

如果你在乎這些問題的答案，那使用「版本控制系統」就是一個很正確的選擇。透過這樣的系統，可以清楚的記錄每個檔案是誰在什麼時候加進來、什麼時候被修改或刪除。Git 就是一種版本控制系統，也是目前業界最流行的版本控制系統，沒有之一。

出社會工作，有 Git 幫你保留這些歷史紀錄跟證據，萬一出事的時候你就能知道是從什麼時候開始就有問題，以及知道該找誰負責，再也不用自己背黑鍋了！

Git 的優點

所以，到底 Git 有什麼厲害的地方，會讓這麼多人選擇它呢：

1. 免費、開源

Git 是由 Linux 核心的作者 Linus Torvalds 在 2005 年為了管理 Linux 核程式碼，僅花了 10 天所開發出來的，至目前已十幾年的歷史了。除了可免費使用外，整個 Git 的原始程式碼也可在網路上取得（當然 Git 的原始程式碼也是用 Git 在做版本控制的）。

2. 速度快、檔案體積小

如果你是使用前面提到的「複製、貼上大法」，這些備份的目錄會很佔空間。而其它大部份的版控系統大多是記錄每個版本之間的差異，而不是完整的備份整個目錄，所以整個目錄的大小就不會快速的增加。

而 Git 特別的設計，在於它並不是記錄版本的差異，而是記錄檔案內容的「快照」(snapshot)，它可以讓 Git 在非常快速的切換版本。至於什麼是「快照」，在後面的章節會有更仔細的介紹。

3. 分散式系統

對我來說，這個可能是最大的優點了。在以往其它的版本控制系統，例如 [CVS](#) 或是 [SVN](#) 之類的集中式的版控系統 (Centralized Version Control)，都需要有一台專用的伺服器，所有的更新都需要跟這台伺服器溝通。也就是說，萬一這台伺服器壞了，或是沒有網路連線的環境，版本控制功能就沒辦法使用。

而 Git 是一款分散式的版控系統 (Distributed Version Control)，雖然通常也會有共同的伺服器，但即使在沒有伺服器或是沒有網路的環境，依舊可以使用 Git 來進行版控，待伺服器恢復正常運作或是在有網路的環境後再進行同步，不會受影響。而且，事實上在使用 Git 的過程中，大多的 Git 操作也都是在自己電腦本機就可以完成。

Git 的缺點

硬是要說 Git 的缺點，大概就是易學難精吧。雖然 Git 的指令有非常多，而且有的指令有點複雜，但還好平常會用到的指令不太多，根據「80/20 法則」，大概 20% 的指令就足以應付平日 80% 的工作。

除了在終端機（或命令提示字元）環境的 Git 指令外，也有許多厲害的圖形介面工具，可以讓使用者不用敲打複雜的指令就可以享用 Git 強大的功能。本書也將會使用終端機指令解述概念並以圖形介面工具（例如 SourceTree）輔助說明 Git 是怎麼運作的。

Git 教學：新手常見問題

等等...我是設計師，我也可以用 Git 嗎？

基本上，Git 只關心檔案的「內容」，所以只要是檔案，其實都可以使用 Git 來管理。只是，設計師工作產出的檔案大多是 PhotoShop 的 PSD 或是 Illustrator 的 AI 檔，雖然 Git 也可以管理這些檔案，但因為這些檔案（二進位檔）不像一般文字檔可以一行一行的被檢視，所以就沒辦法那麼精準的知道什麼人在什麼時候改了哪些字。但整體來說，使用 Git 還是可以幫得上設計師的忙，至少當檔案不小心被覆蓋或刪除的時候，還可以救回舊的版本的檔案。

Git 我知道，我有看過那個 GitHub 什麼的...

這也是很多新手容易有的誤會，以為 Git 就是 GitHub（或是反過來以為 GitHub 就是 Git），甚至也曾看過在公司徵人的職缺上寫著「會使用 GitHub」這項技能。事實上 Git 是一款版本控制軟體，而 GitHub 是一個商業網站，GitHub 的本體是一個 Git 伺服器，但這個網站上的應用程式讓大家可以透過 Web 介面來操作一些原本需要複雜的 Git 指令才能做到的事。

本書後面也有介紹到如何使用 GitHub 與其它人共同協作，雖然 GitHub 很好用，但別忘了 Git 才是本體喔。

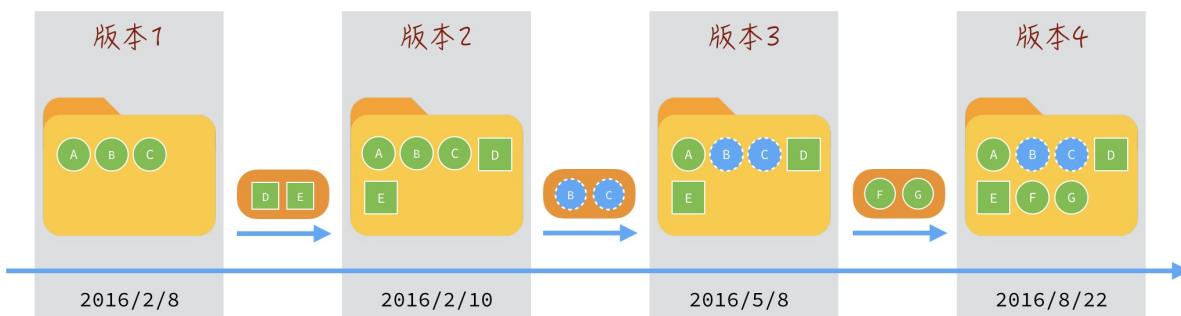
與其它版本控制系統的差異

分散式系統

跟其它的版控系統（例如 CVS 或 SVN）相比，在上一章有提到，Git 是一種分散式的版本控制系統，就算在深山裡或飛機上沒有網路可使用，也可正常的使用 Git，待有網路的時候再與其它人同步即可。Git 大部份的操作都是在自己電腦上就可完成，而且不管是遠端的伺服器或是自己的電腦，在同步之後大家都會有一份完整的檔案及歷史紀錄。

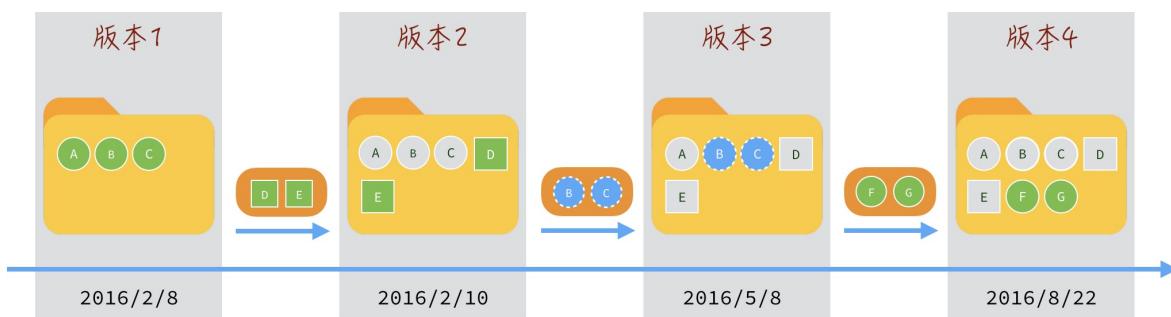
處理檔案的方式

Git 與其它版控系統最大的差異，是在於處理檔案的方式。其它的版控系統，大多是記錄每個版本之間的「異動」：



例如從版本 1 到版本 2，新增了 `D` 跟 `E` 這兩個檔案，版本 2 到版本 3 則是修改了 `B` 跟 `C` 這兩個檔案，以此類推。由於記錄了每個版本異動的片段內容，所以版控系統就可以依據這些「片段」，拼湊成完整的檔案。

但是 Git 就不同了：



對 Git 來說：

1. 版本 1 到版本 2：雖然新增了 `D` 跟 `E` 這兩個檔案，但原本的 `A`、`B` 以及 `C` 檔案都沒

有異動，所以版本 2 除了記錄了新增加的檔案外，其餘的 A、B、C 這三個檔案會指向版本 1 的 A、B、C 那三個檔案（以灰色表示）。

2. 版本 2 到版本 3：因為只修改了 `B` 跟 `C` 兩個檔案，`D` 跟 `E` 這兩個沒有更改的檔案就是指向版本 2 的 `D` 跟 `E`，而 `A` 則是指向版本 1 的 `A`。
3. 版本 3 到版本 4：新增了 `F` 跟 `G` 兩個檔案，但 `A` 一直維持不變，所以它一直都還是指向版本 1 的 `A` 檔案；`B` 跟 `C` 檔案會指向版本 3 的 `B`、`C`，而 `D` 跟 `E` 則是指向版本 2 的 `D` 跟 `E`。

用白話文來說，就是其它的版控系統在版本變化的時候記錄每次不一樣的地方，而 Git 則是有點像拍照（snapshot）一樣，在每次版本變化的時候，Git 會更新並記錄整個目錄跟檔案的樹狀結構。但如果各位覺得上面這段過程還是太難理解也沒關係，先大概有個印象，在後面章節也都還有更詳細的說明。

其實，在 Git 裡有一些比較複雜一點的物件結構，分別是 `Blob`、`Tree`、`Commit` 以及 `Tag` 等等物件，它們都藏在一個名為 `.git` 的目錄裡面，這些都會在之後的「在 `.git` 目錄裡有什麼東西？」單元有更詳細的介紹。只要你能了解這些物件的用途，就更能理解 Git 本身的運作原理，就會發現其實 Git 真的是一款相當簡單而且有效率的工具。

安裝在 Windows 作業系統

要在 Windows 作業系統上安裝 Git，請到官方網站下載合適的版本：



Your download is starting...

You are downloading the latest (2.21.0) 32-bit version of Git for Windows. This is the most recent maintained build. It was released 3 months ago, on 2019-02-26.

If your download hasn't started, [click here to download manually](#).

Other Git for Windows downloads

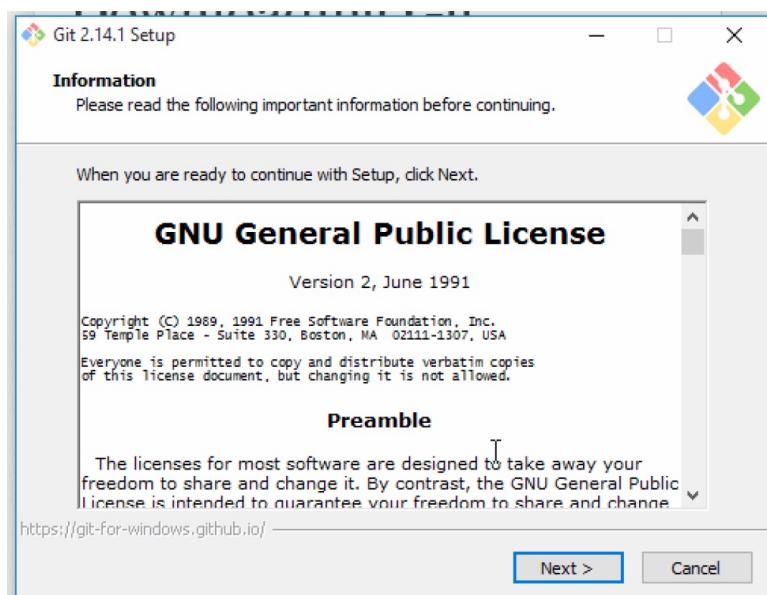
[Git for Windows Setup](#)
[32-bit Git for Windows Setup](#).
[64-bit Git for Windows Setup](#).

[Git for Windows Portable \("thumbdrive edition"\)](#)
[32-bit Git for Windows Portable](#).
[64-bit Git for Windows Portable](#).

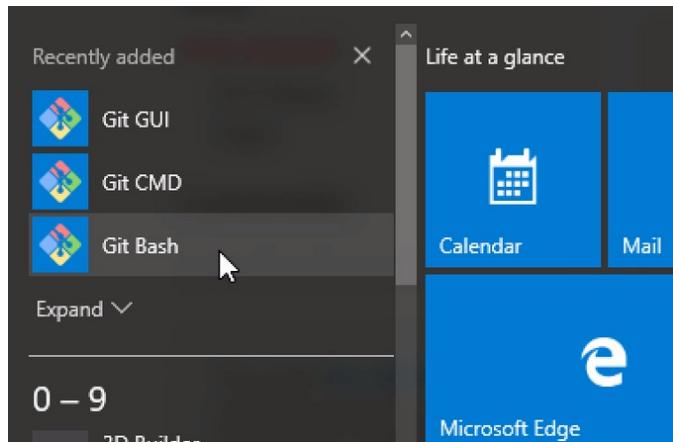
The current source code release is version 2.21.0. If you want the newer version, you can build it from [the source code](#).

網址：<https://git-scm.com/download/win>

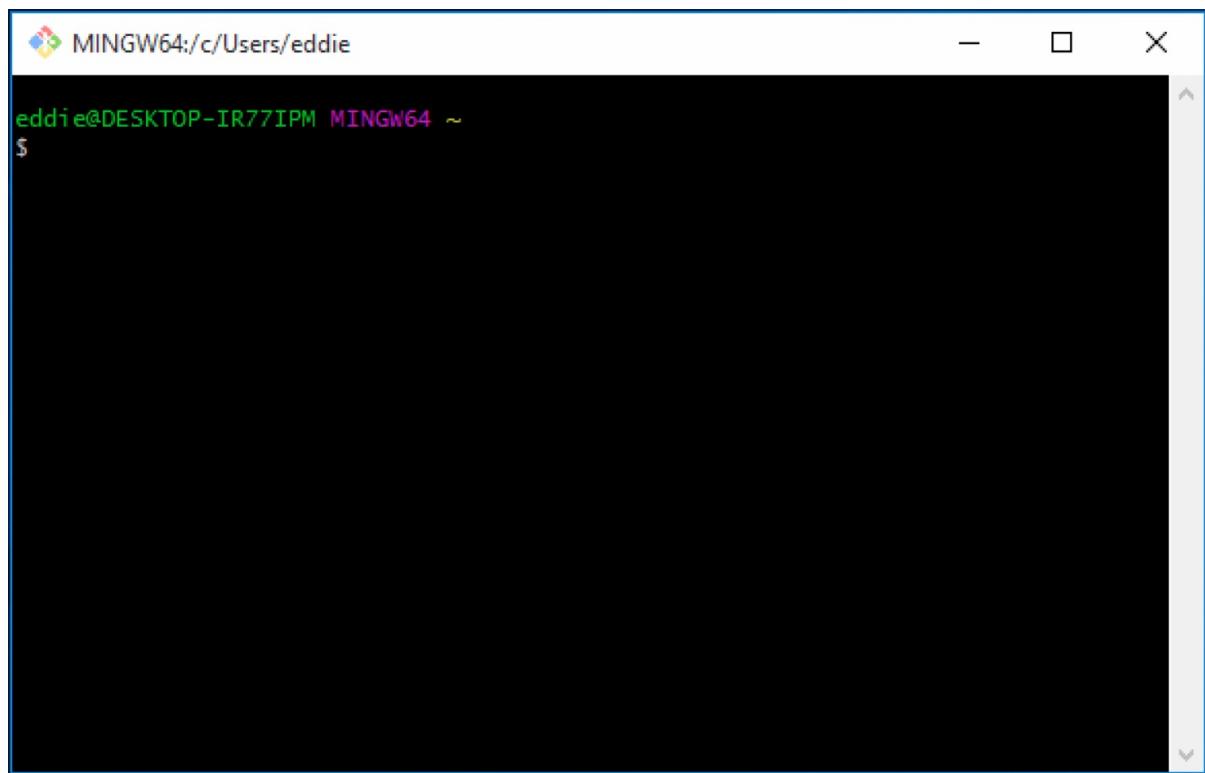
選擇好版本後，安裝也是相當無腦的可以一路按下一步按到底：



安裝完成之後，請找到「Git Bash」這個應用程式：



進入 Git Bash 後，雖然一樣都是黑黑的視窗，但這個跟 Windows 內建的「命令提示字元」不太一樣，它本身模擬了一個在 Linux 的世界還滿有知名度的軟體（其實不能算是一般的應用軟體）叫做 Bash：



到這裡，你可以在那個黑黑的視窗試試輸入指令，驗證一下 Git 是不是有安裝起來，以及版本資訊對不對：

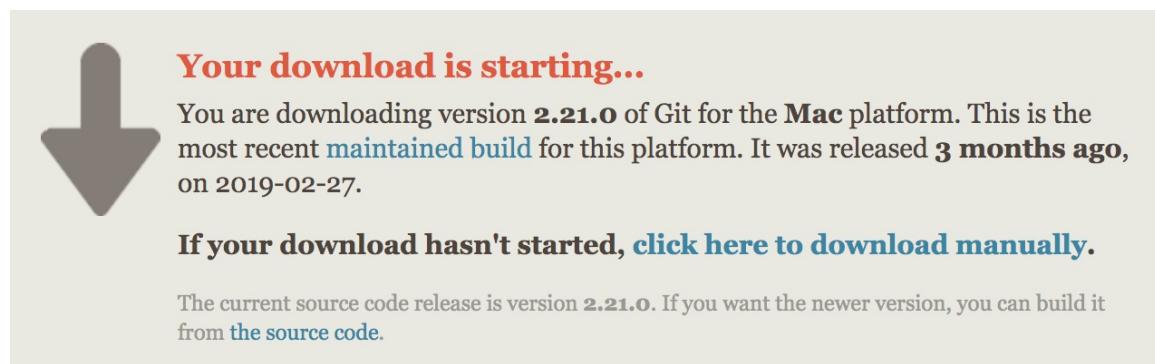
```
$ which git  
/mingw64/bin/git  
  
$ git --version  
git version 2.21.0.windows.1
```

如果看到類似的訊息，就是安裝成功了。

安裝在 Mac OSX 作業系統

在 Mac 上安裝 Git 有幾種做法：

1. 從官方網站下載 Mac 專屬版本的 Git



網站：<https://git-scm.com/download/mac>

只要下載檔案、點開、執行，基本上這樣就可順利完成了。

2. 使用 Homebrew 軟體來安裝 Git



雖然 Mac 作業系統出廠的時候已經有很多軟體或工具了，但對開發者來說很多工具還是得自己想辦法下載甚至得從原始程式碼進行編譯、安裝。而 Homebrew 這個工具就是用來補足這個缺口，它有點像在 Linux 的 `apt-get` 之類的安裝工具，通常只要一行指令就可完成下載、編譯、安裝。如果您本身也是開發者，建議可考慮使用 Homebrew 來安裝軟體。

網址：https://brew.sh/index_zh-tw.html

安裝方式很簡單，就是只要照著它網站上的那行複製起來：

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

在終端機視窗貼上並執行就可以了。

Homebrew 安裝完成後，請一樣在終端機下執行這行指令：

```
$ brew install git
```

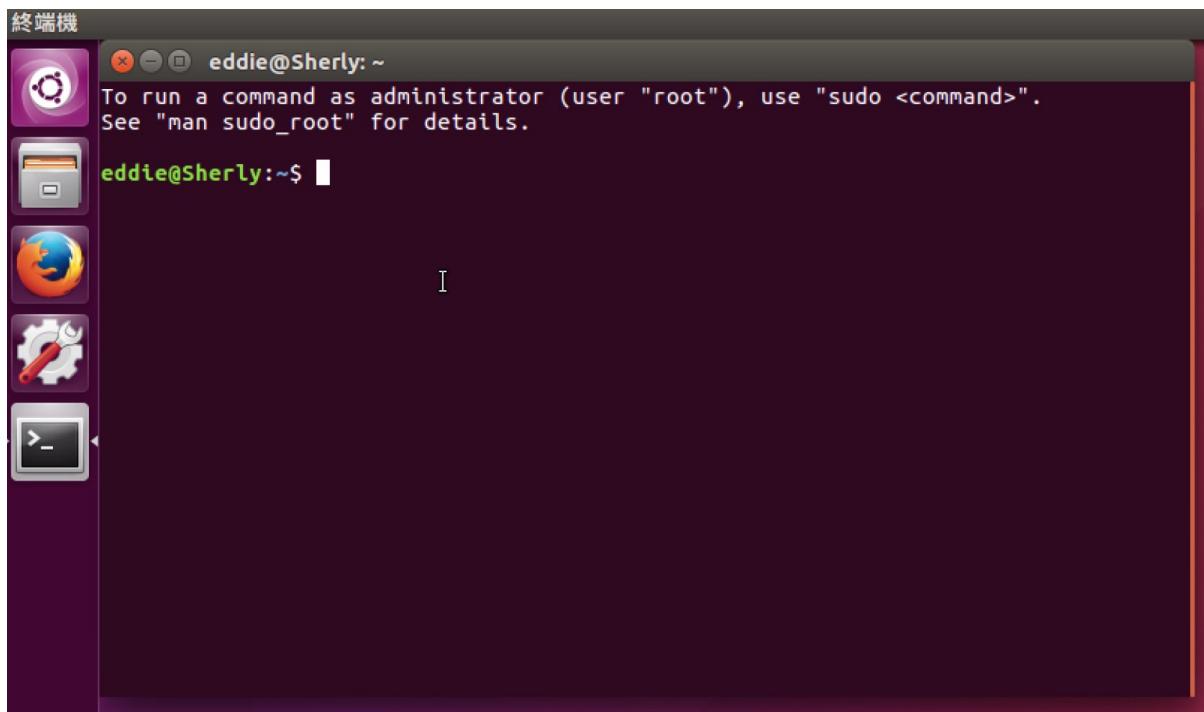
意思就是請 Homebrew 幫你安裝 Git 這個軟體，按下 Enter 鍵之後，剩下的就交給 Homebrew 去煩惱了。

安裝在 Linux 作業系統

在 Linux 安裝 Git，可能算是最簡單的，因為只要一行 `apt-get` 的指令就搞定了。首先，先叫出終端機視窗：



就是這個看起來黑黑的視窗啦：



接著只要輸入一行指令：

```
$ apt-get install git
```

然後就會看到這個錯誤訊息：

```
eddie@Sherly:~$ apt-get install git
E: 無法開啟鎖定檔 /var/lib/dpkg/lock - open (13: 拒絕不符權限的操作)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
eddie@Sherly:~$
```

這意思是說目前這個帳號的權限不足。在 Linux 的世界，不是每個人都可以隨便裝軟體的，需要有足夠的權限才行。不過還好，可以藉由 `sudo` 這個指令，暫時提高權限，讓你可以順利安裝：

```
$ sudo apt-get install git
```

接著就會自動連上網路、下載回來安裝了。

圖形化介面工具

很多人在接觸 Git 都會覺得它不容易學，除了不了解它運作的原理之外，另一個原因就是大部份的人都還是比較習慣圖形介面工具（GUI, Graphic User Interface），對終端機（或命令提示字元）以及 Git 指令操作相對的不熟悉。

GUI 工具的確是比較方便沒錯，不過在不懂原理的前提下，按下 GUI 工具上的按鈕也不知道到底發生什麼事，遇到問題也比較不知道該怎麼處理。

所以，雖然本書是以指令為主，但仍會使用 GUI 工具輔助說明。

在 Git 官方網站上有建議許多款 GUI 工具，有的是商業軟體，有的是免費軟體：

The screenshot shows the official Git website at <https://git-scm.com/downloads/guis>. The main heading is "GUI Clients". Below it, a sub-section titled "SourceTree" is shown with a screenshot of its interface. Another section titled "GitHub Desktop" is also visible.

網址：<https://git-scm.com/downloads/guis>

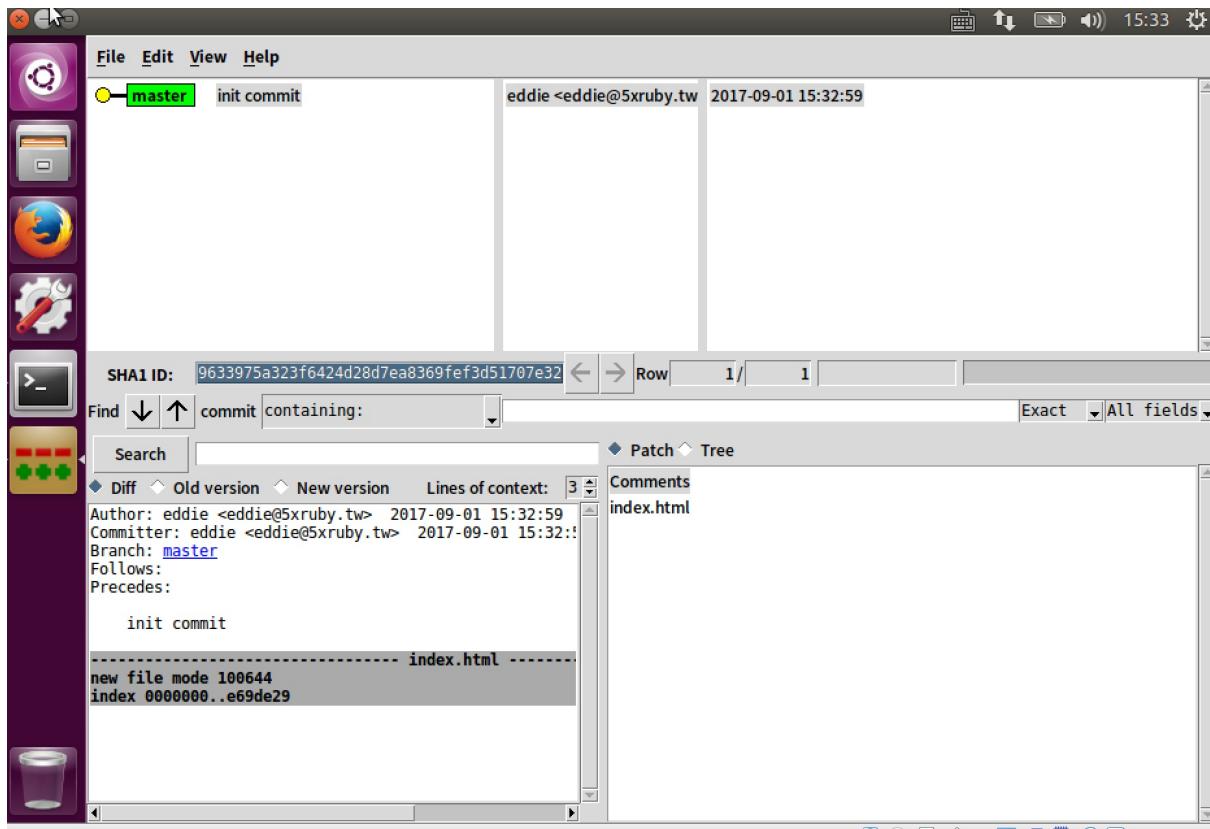
我使用過 SourceTree 以及 GitHub Desktop 這兩款，不僅在 Mac 以及 Windows 作業系統都有，功能都很完整，也都可免費使用，很推薦給大家。

本書將使用 SourceTree 這個軟體做為範例。

至於在 Linux/Ubuntu 上沒有 SourceTree 可以安裝，不過也沒關係，就用陽春一點的 gitk 也可以，安裝一樣只要一行：

```
$ sudo apt-get install gitk
```

gitk 實際運作的畫面像這樣：



gitk 的畫面跟 SourceTree 比起來雖然有點陽春，但該有的功能也是都有的喔！

Git 教學：終端機及常用指令介紹

什麼是終端機

終端機（Terminal）光看名字可能猜不出來什麼意思，其實這是一個來自上一個世代的名詞。在那個時代的電腦很貴，不像現在幾乎每個人都有電腦（可能還不只一台）可以用。公司內部通常會有一台大型的電腦主機，其它人要使用這部電腦就是自己拿螢幕跟鍵盤去接，然後在上面做事情，這些末端的操作設備便統稱之「終端機」。終端機本身通常不是一部電腦，它沒有運算能力，僅用來顯示資料及輸入資料所有的計算都是在主機上處理的。

而現在我們所稱的終端機，大多是電腦工程師手邊那個黑色會一直跑程式碼的畫面：



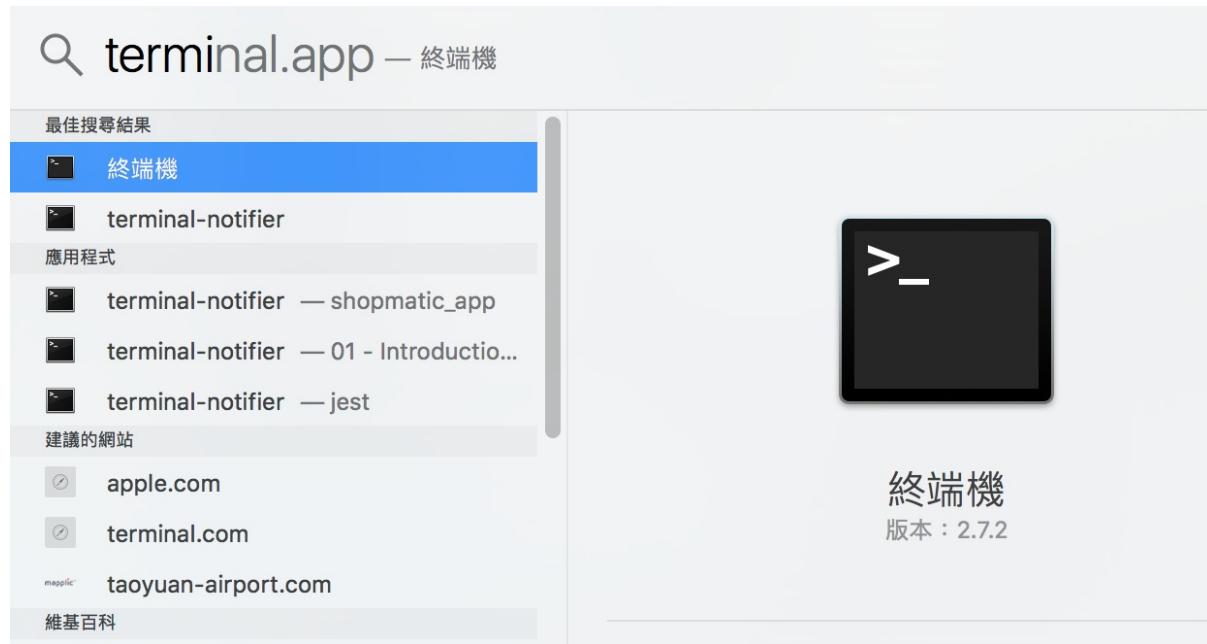
```
2. eddie@gaojiande-MacBook: /tmp/git-practice (zsh)

● 高 見龍 ⌂ cat ✓ 1                               git-practice
total 0
drwxr-xr-x  6 eddie  wheel  204 Aug 15 16:20 .
drwxrwxrwt 28 root   wheel  952 Aug 15 16:51 ..
drwxr-xr-x 14 eddie  wheel  476 Aug 15 17:06 .git
-rw-r--r--  1 eddie  wheel     0 Aug 15 16:20 cat1.html
-rw-r--r--  1 eddie  wheel     0 Aug 15 16:18 index.html
-rw-r--r--  1 eddie  wheel     0 Aug 15 16:19 mam.html

● 高 見龍 ⌂ cat ✓                                git-practice
```

其實就是讓使用者可以輸入指令、跟電腦進行互動。大家看到的畫面應該跟我的不一樣，因為這是我自己客制化過的終端機設定。

在 MacOS 開啟終端機的方式，可以點擊從右上角的「放大鏡」功能，搜尋「terminal」：



就能找到並啟動它。

而在 Windows 作業系統，比較簡單的方式就是按下 `Windows 鍵 + X`，此時會跳出一個選單，選擇中間的「命令提示字元」即可進入一個類似終端機的視窗。

不過如果在 Windows 作業系統使用 Git，在安裝完「Git for Windows」套件之後，便會有一個 Git CMD 或 Git Bash 可以使用，可以讓你比較順利的使用 Git 指令。

使用 Git 前必知：常用終端機命令列指令

在學習 Git 的過程中，許多指令都是在終端機（Terminal）環境下操作。大部份的初學者較習慣圖形介面工具，不熟悉指令該怎麼輸入，或是輸入的指令是什麼意思，這是讓新手覺得容易挫折的地方。以下介紹幾個在終端機環境常會用到的系統指令。

Windows	MacOS / Linux	說明
cd	cd	切換目錄
cd	pwd	取得目前所在的位置
dir	ls	列出目前的檔案列表
mkdir	mkdir	建立新的目錄
無	touch	建立檔案
copy	cp	複製檔案
move	mv	移動檔案
del	rm	刪除檔案
cls	clear	清除畫面上的內容

不同的作業系統，指令也會不太一樣。

使用 Git 前必知終端機指令：目錄切換及顯示

在使用 Git 時，指令要在正確的目錄下才能正常運作，所以學會目錄的切換是很重要的。

```
# 切換到 /tmp 目錄（絕對路徑）
$ cd /tmp

# 切換到 my_project 目錄（相對路徑）
$ cd my_project

# 往上一層目錄移動
$ cd ..

# 切換到使用者的 home 目錄中的 project 裡的 namecards 目錄
# 這個 "~" 符號表示 home 目錄
$ cd ~/project/namecards/

# 顯示目前所在目錄
$ pwd
/tmp
```

如果是 Windows：

```
# 切換到 D 槽的 5xruby 目錄（絕對路徑）
C:\> cd D:\5xruby

# 切換到 5xruby 目錄（相對路徑）
D:\> cd 5xruby

# 往上一層目錄移動
D:\5xruby> cd ..

# 顯示目前所在目錄
D:\5xruby> cd
D:\5xruby
```

使用 Git 前必知終端機指令：檔案列表

`ls` 指令可列出在目前目錄所有的檔案及目錄，後面接的 `-al` 參數，`a` 是指連小數點開頭的檔案（例如`.gitignore`）也會顯示，`l` 則是完整檔案的權限、擁有者以及建立、修改時間：

```
$ ls -al
total 56
drwxr-xr-x 18 user  wheel  612 Dec 18 02:20 .
drwxrwxrwt 24 root  wheel  816 Dec 18 02:19 ..
-rw-r--r--  1 user  wheel   543 Dec 18 02:19 .gitignore
-rw-r--r--  1 user  wheel  1729 Dec 18 02:19 Gemfile
-rw-r--r--  1 user  wheel  4331 Dec 18 02:20 Gemfile.lock
```

```
-rw-r--r--  1 user  wheel  374 Dec 18 02:19 README.md
-rw-r--r--  1 user  wheel  227 Dec 18 02:19 Rakefile
drwxr-xr-x 10 user  wheel  340 Dec 18 02:19 app
drwxr-xr-x  8 user  wheel  272 Dec 18 02:20 bin
drwxr-xr-x 14 user  wheel  476 Dec 18 02:19 config
-rw-r--r--  1 user  wheel  130 Dec 18 02:19 config.ru
drwxr-xr-x  4 user  wheel  136 Dec 18 02:41 db
drwxr-xr-x  4 user  wheel  136 Dec 18 02:19 lib
drwxr-xr-x  4 user  wheel  136 Dec 18 02:23 log
drwxr-xr-x  9 user  wheel  306 Dec 18 02:19 public
drwxr-xr-x  9 user  wheel  306 Dec 18 02:19 test
drwxr-xr-x  7 user  wheel  238 Dec 18 02:23 tmp
drwxr-xr-x  3 user  wheel  102 Dec 18 02:19 vendor
```

使用 Git 前必知終端機指令：如何建立檔案、目錄

```
$ touch index.html
```

如果 `index.html` 這個檔案本來不存在，`touch` 指令會建立一個名為 `index.html` 的空白檔案；如果本來就已經存在，則只會改變這個檔案的最後修改時間，不會變更原本這個檔案的內容。

```
$ mkdir demo
```

`mkdir` 指令會在目前所在目錄，建立一個名為 `demo` 的目錄。

使用 Git 前必知終端機指令：檔案操作

把檔案 `index.html` 複製一份成 `about.html`：

```
$ cp index.html about.html
```

把檔案 `index.html` 更名成 `info.html`：

```
$ mv index.html info.html
```

刪除檔案 `index.html`：

```
$ rm index.html
```

刪除在這個目錄裡所有的 `html` 檔：

```
$ rm *.html
```

在 Windows 作業系統則是把 `cp` 、`mv` 以及 `rm` 指令分別換成 `copy` 、`move` 以及 `del` 。

不要害怕指令、不要害怕錯誤

這些看起來好像很難的指令，請不要太擔心，在學習過程用到的 Git 指令其實都不會太複雜，多試幾次就能上手，不要因為指令輸入錯誤而造成挫折。

另外，在執行指令後，不管成功或失敗，通常都會有訊息顯示在畫面上，這些訊息請多花幾秒鐘仔細的閱讀（最好把它唸出來）。很多的新手以為看到訊息就等於是指令執行成功，但事實上可能是錯誤訊息。

不要害怕輸入指令，不要害怕錯誤訊息，加油！

超簡明 Vim 操作介紹

Vi 是一款非常有年紀的軟體，它從 1976 年出生到現已經超過 40 年的歷史，我們現在用的大多是強化過的版本 Vim (Vi IMproved)，第一個 Vim 版本也超過 25 年了。

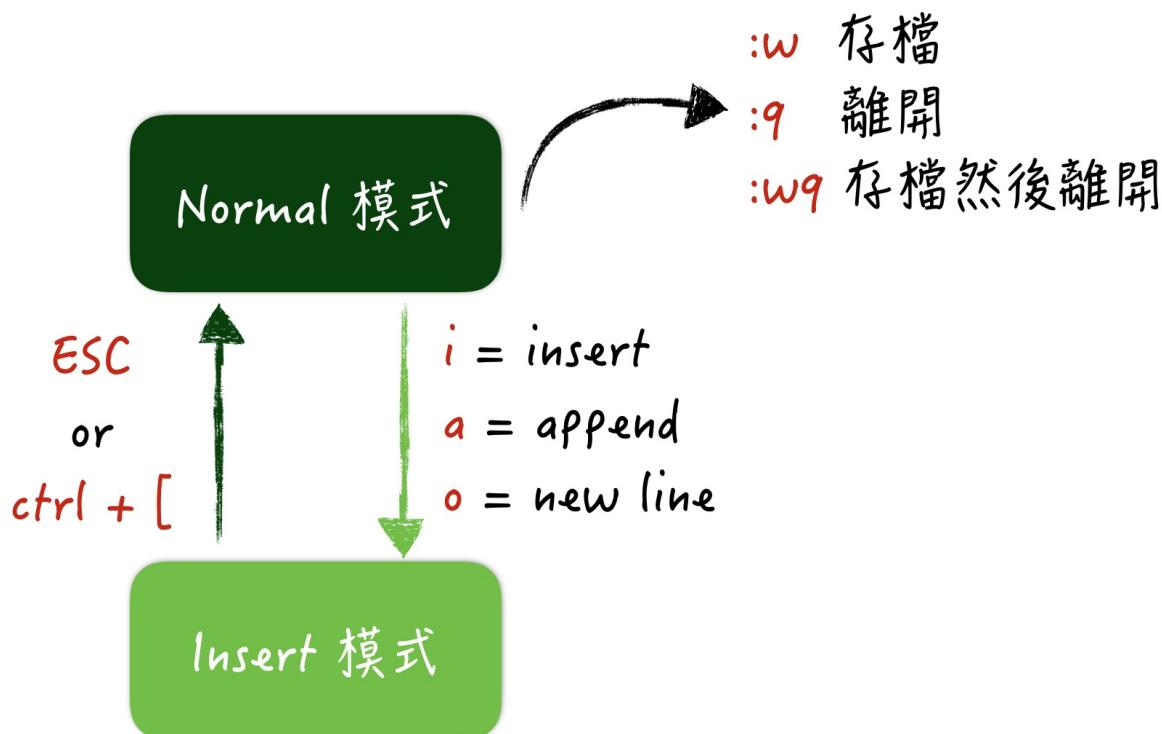
很多人打開它，會發現不知道怎麼開始打字、輸入；好不容易開始輸入了，會不知道該怎麼存檔；好不容易會存檔了，卻會不知道該怎麼離開，簡直可以說是數位版的密室脫逃。

由於 Vim 是 Git 的預設編輯器，不少人是在使用 Git 的過程中意外進入 Vim 編輯器之後不知道如何編輯內容，甚至知名的 Stack Overflow 網站上問到關於「如何退出 Vim？」的問題有超過百萬人瀏覽。

喜歡用它而且用習慣的人，會覺得它非常好用，但這款軟體的入門門檻有點高，我們也沒打算在這裡介紹它完整的功能，本章將僅介紹它的基本使用方法，可以順利的輸入、存檔、離開。

模式切換

Vim 主要是使用模式的切換來進行輸入、移動游標、選取、複製及貼上等操作。在 Vim 主要常用的有幾個模式：Normal 模式以及 Insert 模式：



說明如下：

1. Normal 模式，又稱命令模式，在這個模式下，無法輸入文字，僅能進行複製、貼上、存檔

或離開動作。

2. 要開始輸入文字，需要先按下 `i`、`a` 或 `o` 這三個鍵其中一個進入 Insert 模式，便能開始打字。其中，`i` 表示 `insert`，`a` 表示 `append`，而 `o` 則是表示會新增一行並開始輸入。
3. 在 Insert 模式下，按下 `ESC` 鍵或是 `Ctrl + [` 組合鍵，可退回至 Normal 模式。
4. 在 Normal 模式下，按下 `:w` 會進行存檔，按下 `:q` 會關閉這個檔案（但若未存檔會提示先存檔再離開），而 `:wq` 則是存檔完成後直接關閉這個檔案。

Vim 的指令還非常多，但就以在 Git 會遇到的狀況來說（主要是編輯 Commit 訊息），上述這些指令應該已經足夠使用。

使用者設定

要開始使用 Git，首先要做的第一件事（應該也只要做一次就好），就是設定使用者的 Email 信箱以及使用者名稱。請打開你的終端機視窗，輸入下面這兩行指令：

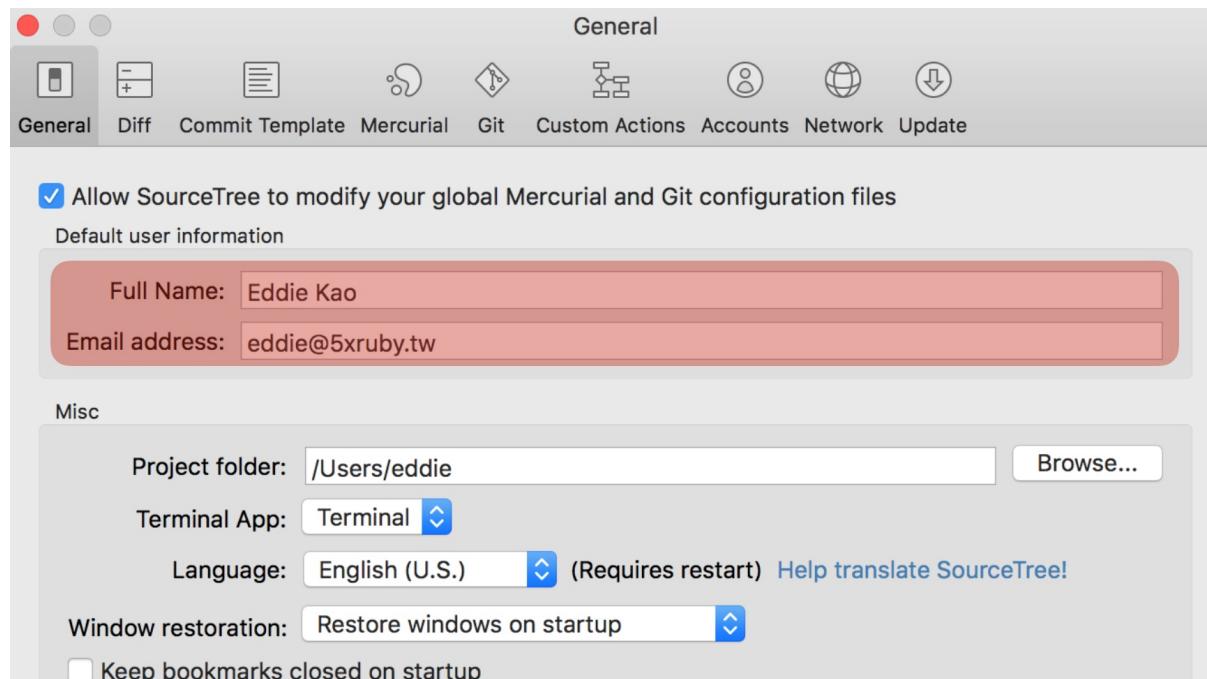
```
$ git config --global user.name "Eddie Kao"
$ git config --global user.email "eddie@5xruby.tw"
```

輸入完成之後，可以再檢視一下目前的設定：

```
$ git config --list
user.name=Eddie Kao
user.email=eddie@5xruby.tw
```

如果你有安裝過 Git 相關的圖形化介面工具，可能 `git config --list` 這個指令還會輸出其它額外的設定，但至少我們剛剛的這兩行設定會新增 `user.name` 跟 `user.email` 這兩個設定。

如果你不喜歡或不習慣敲打終端機指令，也可透過 SourceTree 來設定：



會有一樣的效果。

設定檔的位置

不管是透過終端機指令或是圖形介面工具做的設定，所有 Git 相關的設定，預設會在自己帳號下的 `.gitconfig` 這個檔案裡，所以使用一般的文字編輯器，直接手動修改這個檔案也會有一樣的效果：

檔案：`~/.gitconfig`

```
[user]
  name = Eddie Kao
  email = eddie@digik.com.tw
[core]
  excludesfile = /Users/eddie/.gitignore_global
# ...略...
```

【狀況題】可以每個專案設定不同的作者嗎？

你可能有注意到剛剛前面在做設定的時候有多加了一個 `--global` 的參數，意思是要做全域 (Global) 的設定。但偶爾會遇到一些特別的狀況，例如是要幫特定的專案設定不同的作者 (不要問我什麼時候會需要做這件事)，可以在該專案目錄下進行 Git 設定的時候，加上 `--local` 參數：

```
$ git config --local user.name Sherly
$ git config --local user.email sherly@5xruby.tw
```

這樣一來，在這個專案進行操作的時候，就會使用特別幫這個專案設定的使用者名稱及 Email 來進行操作。離開這個專案後的 Git 操作，就還是會使用預設的 Global 設定。

其它方便的設定

工程師大多有著懶惰的美德，討厭重複的工作，所以常會幫自己設定一些方便的設定，以下是一些我自己在 Git 上常會做的方便設定。

更換編輯器

一般來說，Vim 是 Git 預設的編輯器，所以在終端機下使用 Git，對新手的麻煩之一就是或多或少都會遇到 Vim 這個編輯器。雖然我們在[超精簡 Vim 操作介紹](#)章節介紹了這個神奇的編輯器大概怎麼使用，但對平常沒在用，或用不習慣的人來說還是會覺得很麻煩。

不過還好，不一定要用 Vim，你可以在終端機下執行這個指令：

```
$ git config --global core.editor emacs
```

這樣就可以把預設的 Vim 編輯器換成 Emacs 了（嘆？好像沒比較好？）

其實除了 Vim 或 Emacs 之外，也可以使用 Sublime Text、Atom 或是 Visual Studio Code 比較現代的文字編輯器。只是你需要先 Google 一下怎麼從終端機啟動這些應用程式，然後就可以用一樣的方法把 Vim 換掉了。

如果在操作 Git 的過程中跳出 Vim 這個問題對你一直很困擾，那建議可以搭配圖形介面軟體使用。

懶得打字，或常打錯字

雖然 Git 指令不長，但有時候就懶得打那麼多字（例如 `checkout` 指令就有 8 個字母），或是有些指令就是常會打錯（例如 `status` 指令我就常打成 `state`）。

遇到這種狀況，我們可以在 Git 裡設定一些「縮寫」，然後就可以少打幾個字。只要在終端機下執行：

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.st status
```

設定之後，輸入 `git co` 指令就跟 `git checkout` 會有一樣的效果，`git st` 就有 `git status` 的效果，一下就省了很多個鍵。

而且還可以再加一些參數進去，例如每次在看 log 的時候，為了看比較精簡的資訊，都得要輸入 `git log --oneline --graph` 這麼長的指令，改用 Alias 設定的話：

```
$ git config --global alias.l "log --oneline --graph"
```

這樣之後只要輸入 `git l` 就可以有原來 `--oneline --graph` 的效果了：

```
$ git l
* cc200b5 (HEAD -> master) Merge branch 'cat'
| \
| * 0d1d15d (cat) add cat 2
| * 0d392fb add cat 1
| /
* 657fce7 (origin/master, origin/HEAD) add container
* abb4f43 update index page
* cef6e40 create index page
* cc797cd init commit
```

或是甚至可以再把格式弄複雜一點，把 Commit 時間的人跟時間都加進來：

```
$ git config --global alias.ls 'log --graph --pretty=format:"%h <%an> %ar %s"'
```

尾巴那段看起來有點複雜的 `format` 的參數就是印出 Commit 的各別資訊，參數代表的意思請執行 `git help log` 後查閱關於 `format` 有關的段落。用起來的樣子就會像這樣：

```
$ git ls
* cc200b5 <Eddie Kao> 9 seconds ago Merge branch 'cat'
| \
| * 0d1d15d <Eddie Kao> 18 seconds ago add cat 2
| * 0d392fb <Eddie Kao> 20 seconds ago add cat 1
| /
* 657fce7 <Eddie Kao> 13 days ago add container
* abb4f43 <Eddie Kao> 13 days ago update index page
* cef6e40 <Eddie Kao> 2 weeks ago create index page
* cc797cd <Eddie Kao> 2 weeks ago init commit
```

這樣甚至不需要圖形介面工具也可以輕鬆的看 log。上面這些 Alias 的設定，也可直接到 `~/.gitconfig` 裡修改：

```
[alias]
co = checkout
br = branch
st = status
l = log --oneline --graph
ls = log --graph --pretty=format:"%h <%an> %ar %s"
```

不要小看這幾個字的差別，每次少敲幾次鍵盤，長期累積下來也是很驚人的。

新增、初始 Repository

終於，我們要開始使用 Git 了，喔耶！

如果是全新的開始...

我假設這是你第一次使用 Git，所以就先從建立一個全新的目錄開始來練練手感吧。請打開你的終端機視窗，並試著操作以下指令。在指令後面的 `#` 符號是說明，不需要跟著輸入：

```
$ cd /tmp          # 切換至 /tmp 目錄  
$ mkdir git-practice # 建立 git-practice 目錄  
$ cd git-practice    # 切換至 git-practice 目錄  
$ git init          # 初始化這個目錄，讓 Git 對這個目錄開始進行版控  
Initialized empty Git repository in /private/tmp/git-practice/.git/
```

在上面這個範例中主要做了幾件事：

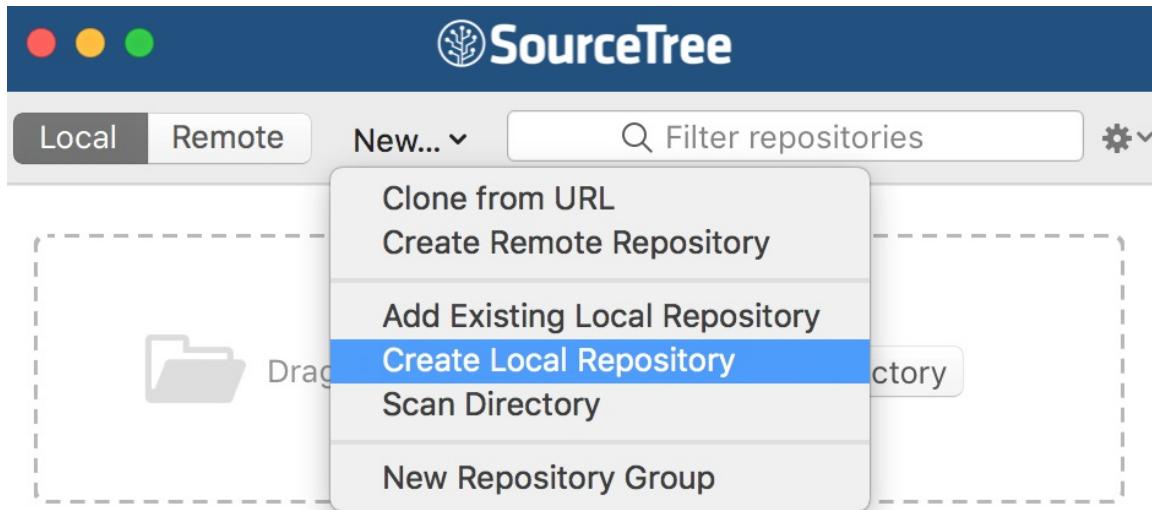
1. 使用 `mkdir` 指令建立了 `git-practice` 目錄。
2. 使用 `cd` 指令切換到剛剛建立的 `git-practice` 目錄。
3. 使用 `git init` 指令初始化這個目錄，主要目的是要讓 Git 開始對這個目錄進行版本控制。

其中，第 3 步這個指令會在這個目錄裡建立一個 `.git` 目錄，整個 Git 的精華就都是在這個目錄裡了。如果各位有興趣可以先看一下這個目錄裡面的內容，不過現在並不打算介紹裡面的細節，請各位先體會一下使用 Git 的手感，待後面的章節會再慢慢介紹這個目錄裡到底在賣什麼藥。

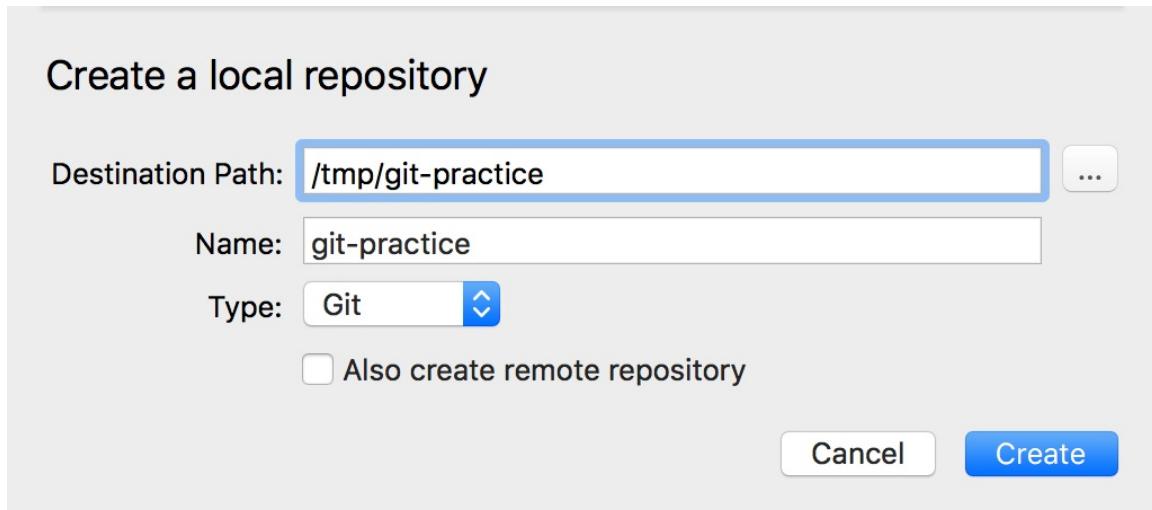
小數點開頭的目錄或檔案名稱（例如 `.git`），在一些作業系統中預設是隱藏的，可能會需要開啟檢視隱藏檔之類的設定才看得到。

如果你是使用 SourceTree，也可以這樣做：

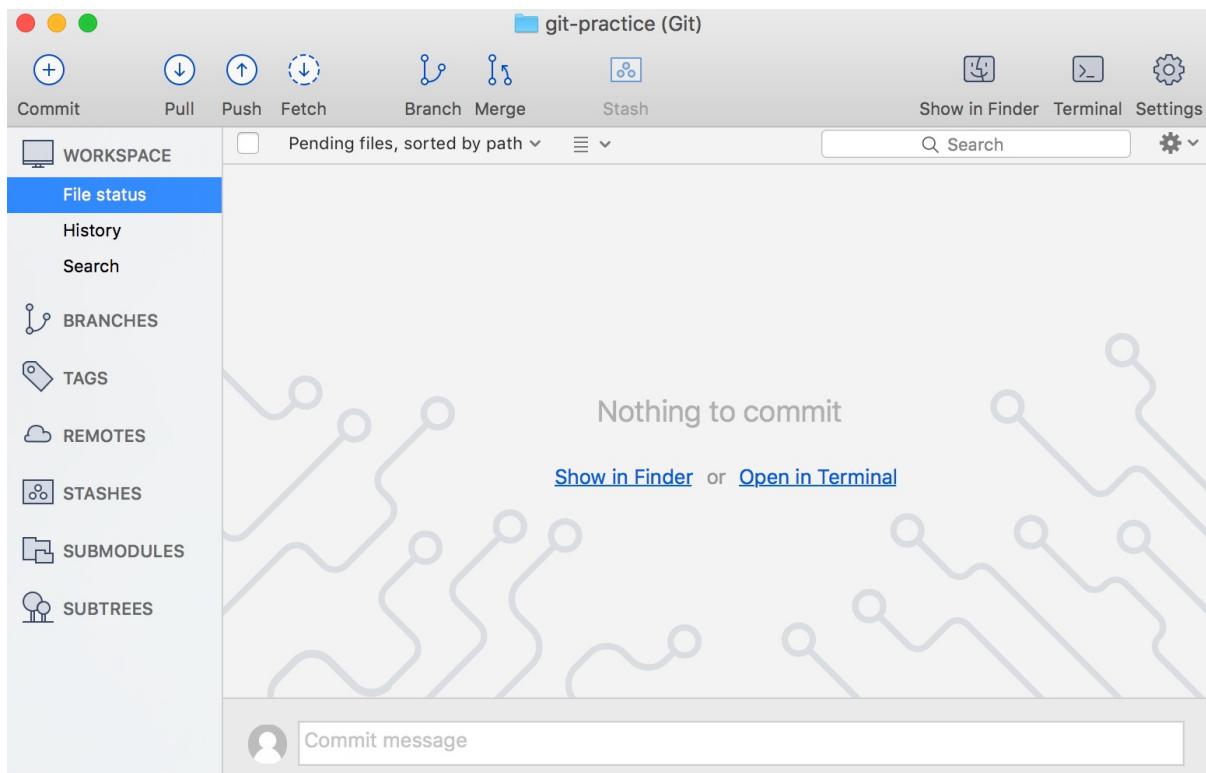
選擇 `New -> Create Local Repository`：



填寫路徑，並選擇使用 Git：



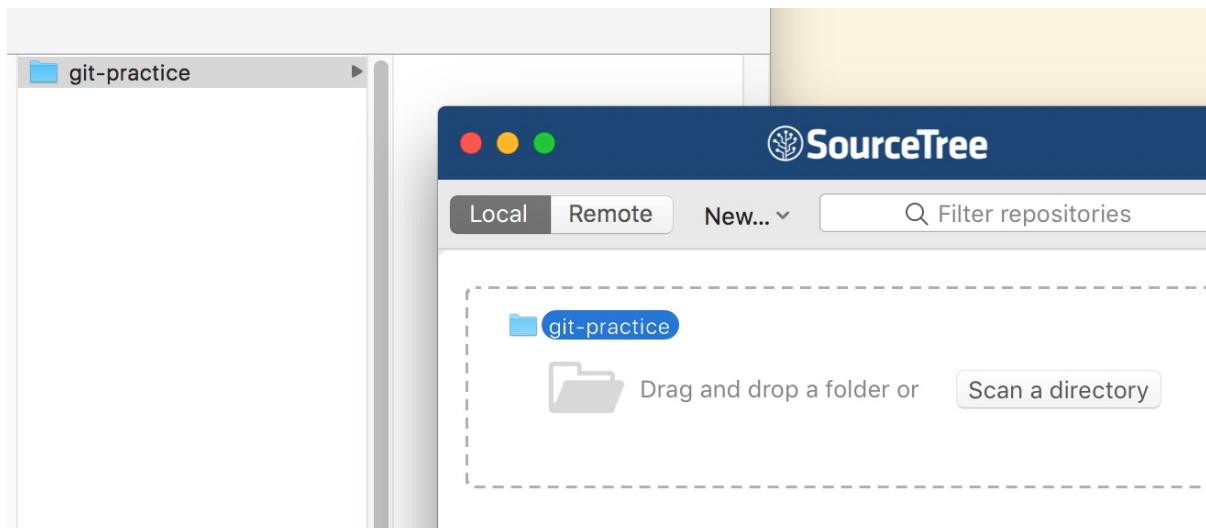
然後就會在 /tmp 目錄下建立一個 git-practice 目錄，最後的畫面會長得像這樣：



這樣一來就可以做出跟指令差不多的效果了。

如果本來就有的目錄...

如果想針對本來就存在的目錄進行版控，其實只要到那個目錄下執行 `git init` 指令即可。如果是使用 SourceTree，則是只要把那個目錄拖拉到 SourceTree 介面即可：



如果這個目錄不想再被 Git 控制？

其實 Git 的版控很單純，全部就是只靠那個 `.git` 目錄在做事而已，所以如果這個目錄不想被版控，或是只想給客戶不含版控紀錄的內容的話，只要把那個 `.git` 目錄移除，Git 就對這個目錄失去控制權了。

也就是說，整個專案目錄裡，什麼檔案或目錄刪了都救得回來，但 `.git` 目錄只要刪了就沒辦法了。

為什麼一直用 `/tmp` 目錄，其它目錄可以嗎？

在本書的範例中，我常會在 `/tmp` 目錄下進行練習，那是因為在 MacOS 系列的作業系統，`/tmp` 目錄裡的東西在下次電腦重新開機（或當機）的時候就全部被清空，所以不需要再手動整理。這算是我在練習的時候的個人喜好，你要使用其它目錄也是可以的，但如果是重要的檔案，千萬不要跟我一樣放在這個地方喔。

把檔案交給 Git 控管

在上個章節我們對目錄進行了 Git 的初始化，讓 Git 開始可以管理這個目錄，接下來，我們就來看看 Git 是怎麼操作的。

把檔案交給 Git

在開始之前，我想先介紹 `git status` 這個指令。這個指令的用途是用來查詢現在這個目錄的「狀態」，先在剛剛建立的 `git-practice` 目錄下執行這個指令：

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

在這個目錄裡，現在除了 Git 幫你產生的那個 `.git` 隱藏目錄外什麼都沒有，所以上面這段訊息就是要跟你說「現在沒東西可以提交 (nothing to commit)」。接下來，在這個目錄裡透過系統指令建立一個內容為 "hello, git" 並命名為 `welcome.html` 的檔案：

```
$ echo "hello, git" > welcome.html
```

這個步驟要用一般的文字編輯器或檔案管理員來完成也沒關係，總之就是在這個目錄裡建立一個叫做 `welcome.html` 的檔案就行了。接著，我們再次使用 `git status` 指令，來看一下這個目錄的狀態：

```
$ git status
On branch master

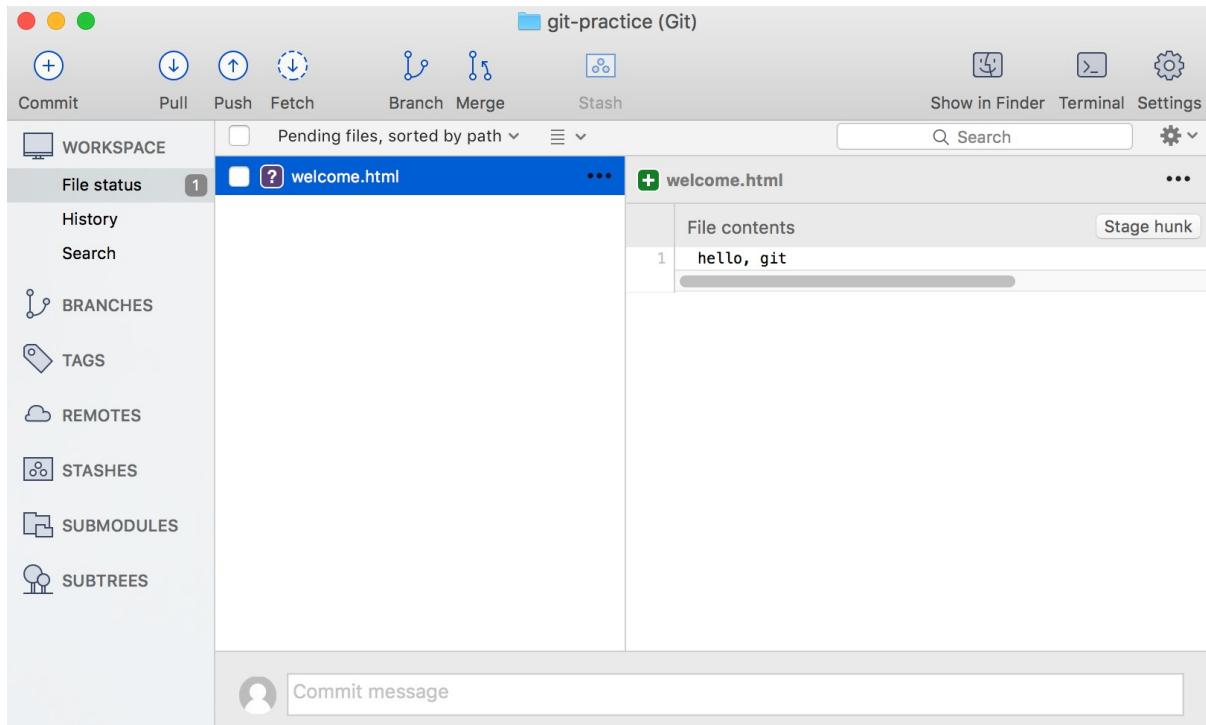
Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    welcome.html

nothing added to commit but untracked files present (use "git add" to track)
```

或是使用 SourceTree 來看：



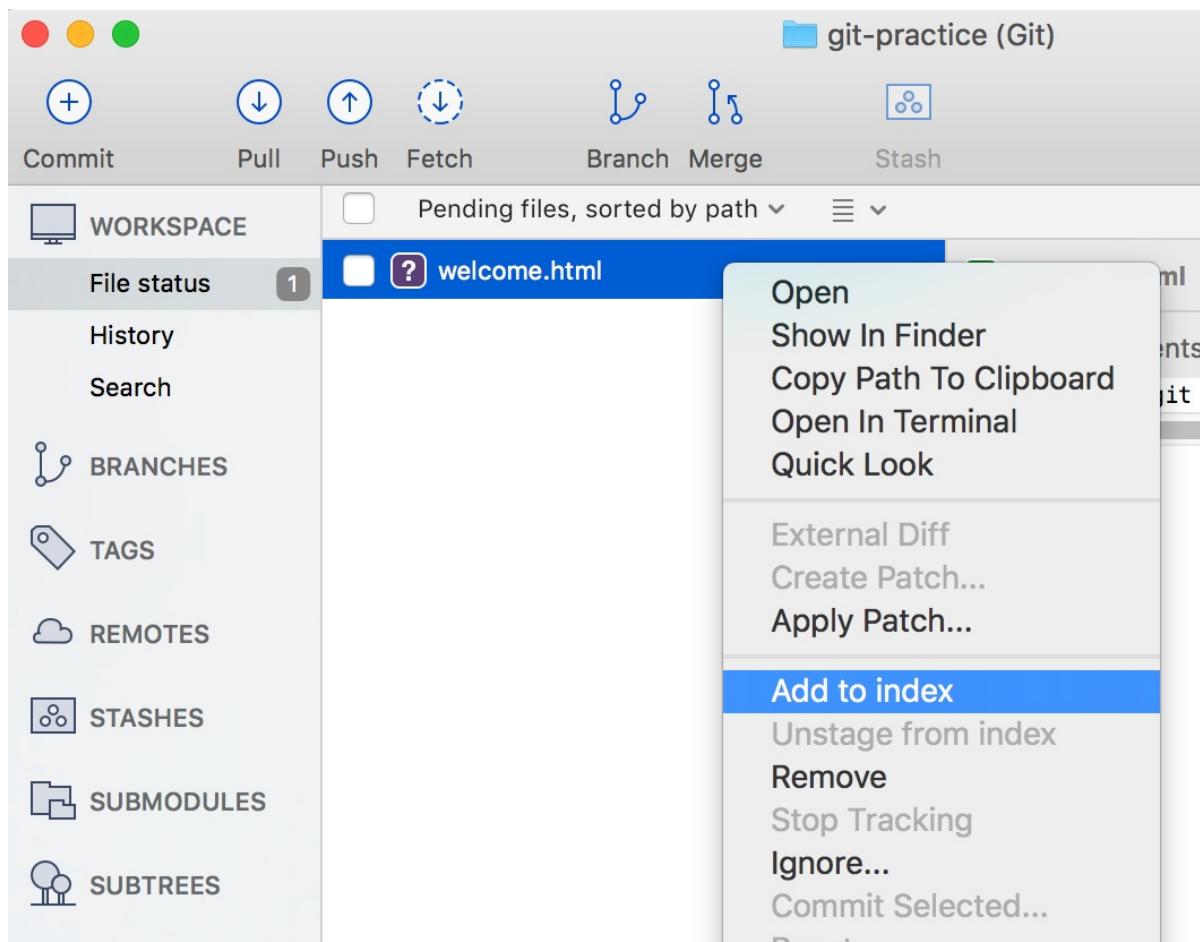
這時候的狀態跟剛才一開始不太一樣了，現在可以看到這個 `welcome.html` 檔案目前的狀態是 `Untracked files`，意思是這個檔案尚未被加到 Git 版控系統裡，還沒開始正式被 Git 「追蹤」，它只是剛剛才加入這個目錄而已。

把檔案交給 Git

既然目前這個檔案是 `Untracked`，接下來就是要把 `welcome.html` 這個檔案交給 Git，讓 Git 開始「追蹤」它，使用的指令是 `git add` 後面加上 檔案名稱：

```
$ git add welcome.html
```

在終端機執行這個指令不會有任何的輸出結果，如果是使用 SourceTree，可以在那個檔案上按滑鼠右鍵，然後選擇「Add to index」：



就可以把這個檔案交給 Git 來管控了。再次使用 `git status` 指令看一下目前的狀態：

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   welcome.html
```

有注意到嗎？剛才那個檔案從 `Untracked` 變成 `new file` 狀態了。這個表示這個檔案已經被安置到暫存區（Staging Area），等待稍後跟其它檔案一起被存到儲存庫裡。

而這個暫存區也稱之索引（index），所以在上圖中，SourceTree 的選單才會用「Add to index」字樣。基本上暫存區跟索引是一樣的意思。為了讓大家比較方便理解，本書將統一使用「暫存區」的用法。至於為什麼要有這樣的設計，請見下一章「[工作區、暫存區與儲存庫](#)」的介紹。

小提示

其實這個 `add` 指令看起來很簡單，但背後 Git 實在幫你做了不少事，詳情請參閱「[【超冷知識】在 .git 目錄裡有什麼東西？Part 1](#)」章節說明。

如果你覺得 `git add welcome.html` 這樣一次只加一個檔案有點麻煩，你也可以使用萬用字元 (wildcard character)：

```
$ git add *.html
```

就可把所有附檔名是 `html` 的檔案全部都加到暫存區。而如果想要一口氣把全部的檔案加到暫存區，可直接使用 `--all` 參數：

```
$ git add --all
```

【狀況題】如果在 `git add` 之後又修改了那個檔案的內容？

想像一下這個情境：

1. 你新增了一個檔案叫做 `abc.txt`。
2. 然後，執行 `git add abc.txt` 把檔案加至暫存區。
3. 接著編輯 `abc.txt` 檔案。

完成編輯後，接著你可能會想要進行 Commit，把剛剛修改的內容存下來。這是新手可能會犯的錯誤之一，以為 Commit 指令就會把所有的異動都存下來，事實上這樣的想法是不太正確的。

執行一下 `git status` 指令，看一下目前的狀態：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   abc.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   abc.txt
```

咦？你發現 `abc.txt` 這個檔案變成 2 個了嗎？其實你在第 2 步的確有把 `abc.txt` 加到暫存區沒錯，但你在第 3 步又編輯了那個檔案，對 Git 來說，編輯的內容並沒有再次被加到暫存區，所以這時候在暫存區裡的資料還是你在第 2 步時候加進來的那個檔案。

如果確定這個修改是你要的，就請再次使用 `git add abc.txt` 指令，再把 `abc.txt` 加至暫存區。

豆知識

其實這樣的動作會產生一些 Unreachable 的沒人愛邊緣物件，這部份算是比較進階的內容，有興趣可參閱「[【冷知識】你知道 Git 有資源回收機制嗎？](#)」章節內容。

【冷知識】"--all" 跟 " ." 參數有什麼不一樣？

有時候會看到別人說「`git add .` 指令也可以把所有的檔案全部加到暫存區喔！」，這樣的說法其實不完全正確，這得看以下幾種情況而決定：

1. Git 版本

在比較舊版本（1.x 版）的 Git，`git add .` 這個指令會把「新增的檔案」（也就是 `Untracked` 狀態的檔案）以及有「修改過的檔案」加到暫存區沒錯，但是不會處理「刪除檔案」的行為。讓我畫個表格簡單說明一下：

使用參數	新增檔案	修改檔案	刪除檔案
--all	○	○	○
.	○	○	✗

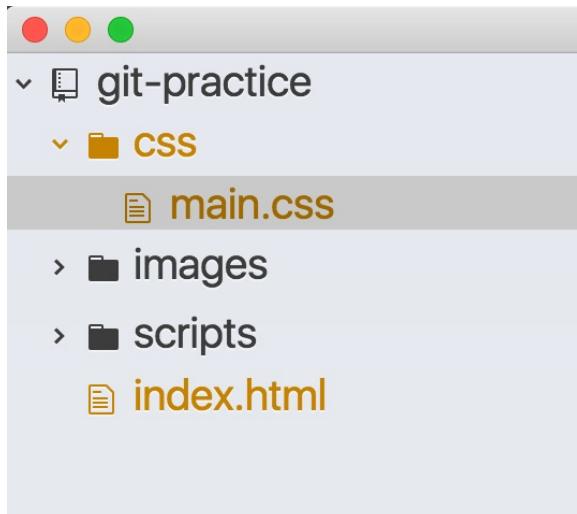
不過，在 Git 2.x 版之後變成這樣：

使用參數	新增檔案	修改檔案	刪除檔案
--all	○	○	○
.	○	○	○

也就是說，在 Git 2.x 之後，這兩個參數在功能上就沒什麼差別了。

2. 執行指令時候的目錄位置

舉個例子來說：



在專案的根目錄的 `index.html` 以及在 `css` 目錄的 `main.css` 都有修改，如果是在根目錄執行 `git add .`，這兩個檔案都會被加進暫存區，但如果是在 `css` 目錄下執行，僅會加入 `main.css`，`index.html` 的狀態不會改變。

那是因為 `git add .` 這個指令會把目前當下這個目錄，以及它的子目錄、子子目錄、子子子目錄...裡的異動全部加到暫存區，但在這個目錄的以外的就不歸它管了。而 `git add --all` 指令就沒這個問題，這個指令不管在專案的哪一層目錄執行，效果都是一樣的，在這個專案裡所有的異動都會被加至暫存區。

所以，回到原來的問題「`--all` 跟 `.` 參數有什麼不一樣？」，答案會跟所使用的 Git 版本不同以及執行指令時的目錄而有所差異。

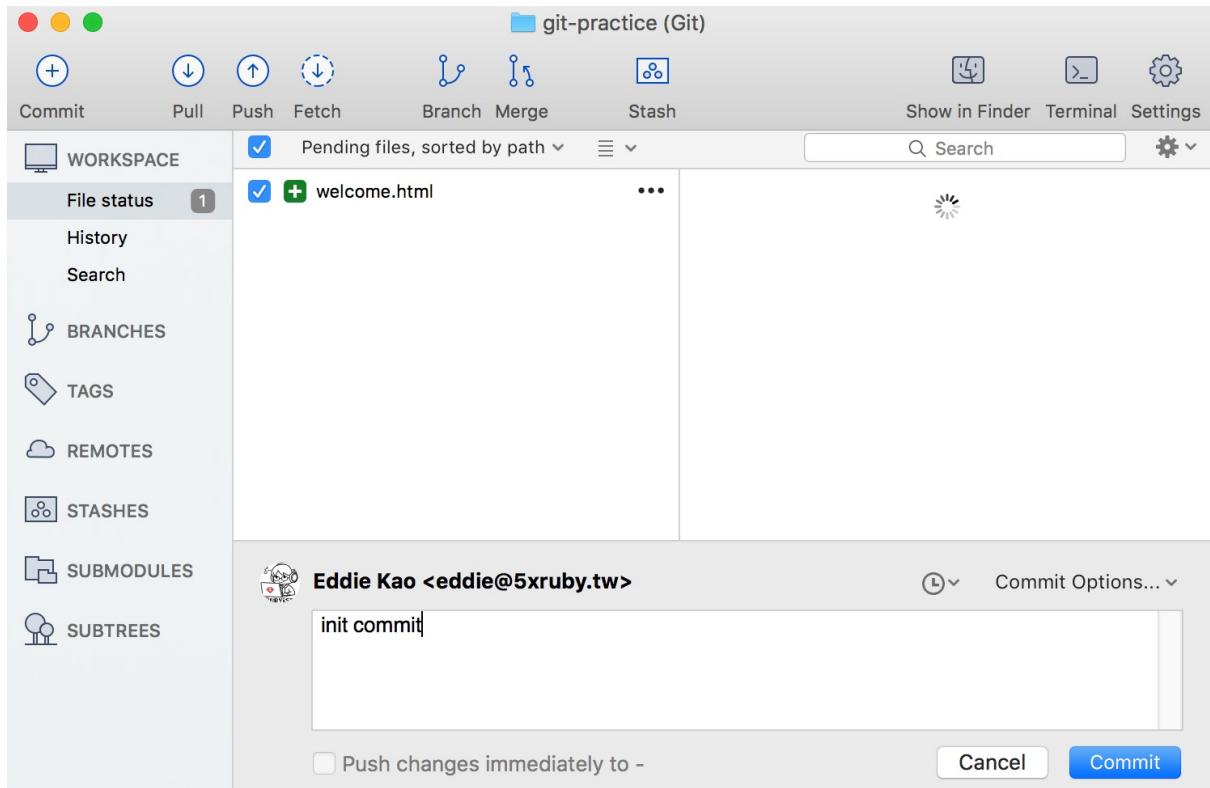
把暫存區的內容提交到倉庫裡存檔

如果僅是透過 `git add` 指令把異動加到暫存區是不夠的，這樣還不算是完成整個流程。要讓暫存區的內容永久的存下來的話，使用的是 `git commit` 指令：

```
$ git commit -m "init commit"
[master (root-commit) dfccf0c] init commit
 1 file changed, 1 insertion(+)
 create mode 100644 welcome.html
```

在後面加上的 `-m "init commit"` 是指要要說明「你在這次的 Commit 做了什麼事」，只要使用簡單、清楚的文字說明就好，中、英文都可，重點是清楚，讓不久之後的你或是你的同事能很快的明白就行了。

如果是使用 SourceTree，可點擊左上角的「Commit」按鈕，就可以開始輸入訊息：



輸入完訊息，按下右下角的「Commit」按鈕，即可完成這次的 Commit。

當完成了這個動作後，對 Git 來說就是「把暫存區的東西存放到儲存庫（Repository）裡」，翻譯成白話文就是「我完成一個存檔（或備份）的動作了」，也是建立了一個我們在第一個章節所提到的「版本」。關於儲存庫，我們會在下個章節「[工作區、暫存區與儲存庫](#)」介紹。

注意！

要完成 Commit 指令才算是完成整個流程喔！

到底 Commit 了哪些東西？

請先記住一個很重要的觀念：「Git 每次的 Commit 都只會處理暫存區（Staging Area）裡的內容」。也就是說，如果在執行 `git commit` 指令的時候，那些還沒被加到暫存區裡的檔案，就不會被 Commit 到儲存庫裡。

舉例來說，如果你新增了一個檔案，但卻沒有執行 `git add` 指令把這個檔案加至暫存區的話，在執行 `git commit` 指令的時候那個檔案就會被無視喔。

那個訊息是什麼？很重要嗎？

對，很重要！很重要！很重要！（因為重要所以要說三次）

在 Commit 的時候，如果沒有輸入這個訊息，Git 預設是不會讓你完成 Commit 這件事的。它最主要的目的就是告訴你自己以及其它人「這次的修改做了什麼」。以下是幾點關於訊息的建議：

1. 儘量不要使用太過情緒性的字眼（我知道開發者有時候工作會有低潮或遇到澳洲來的客人），避免不必要的問題。
2. 英文、中文都沒關係，重點是要簡單、清楚。
3. 儘量不要使用像 `bug fixed` 這樣模糊的描述，因為沒人知道你修了什麼 bug。但如果有搭配其它的系統使用，則可使用 `#34 bug fixed`，因為這樣可以知道這次的 Commit 修改了第 34 號的 bug。

等等，我怎麼跳出一個奇怪的視窗了

在執行 `commit` 指令時，如果沒有在後面加上訊息參數，預設會跳出一個黑黑的畫面，那個就是傳說中的編輯器 - Vim。那是個對新手不太友善的編輯器，各位可參考「[超精簡 Vim 操作介紹](#)」章節的介紹，或是直接使用 SourceTree 之類的圖形介面軟體來處理輸入提交訊息的問題。

【冷知識】一定要有東西才能 Commit 嗎？

只要加上 `--allow-empty` 參數，沒東西也是可以 Commit 的：

```
$ git commit --allow-empty -m "空的"
[master 76a5b84] 空的

$ git commit --allow-empty -m "空的"
[master f4f568c] 空的

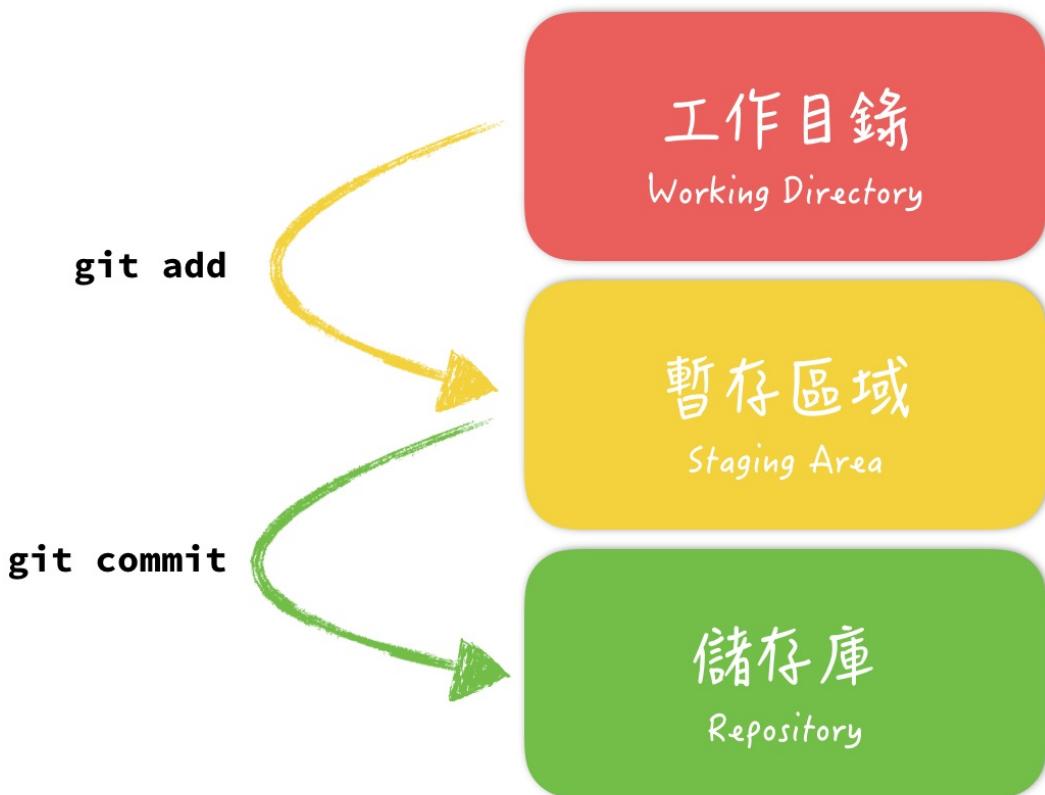
$ git commit --allow-empty -m "空的"
[master 7653117] 空的
```

這樣就做了 3 個空的 Commit 出來，基本上這沒什麼意義，但有時候我們在上 Git 課的時候就會滿方便的，可以不用新增檔案就快速產生 Commit 來練習合併。

工作區、暫存區與儲存庫

在上個章節「[把檔案交給 Git 控管](#)」介紹到可以使用 `git add` 指令把檔案加至暫存區（Staging Area），然後可使用 `git commit` 指令把暫存區的內容移往儲存庫（Repository）。

在 Git 裡，主要可以分成「工作目錄（Working Directory）」、「暫存區（Staging Area）」以及「儲存庫（Repository）」三個區塊，透過不同的 Git 指令，可以把檔案移往不同的區域：



1. `git add` 指令把檔案從工作目錄移至暫存區（或索引）。
2. `git commit` 指令把暫存區的內容移至儲存庫。

請記得，要執行 Commit 指令，也就是要存放到 Repository 才算是完成整個流程喔。

基本上，只要記得這三個區塊怎麼操作，在本機端的 Git 操作應該就沒太大的問題了。不管是用指令操作，或是用圖形介面工具操作，請一定要親自操作、熟悉這個流程。

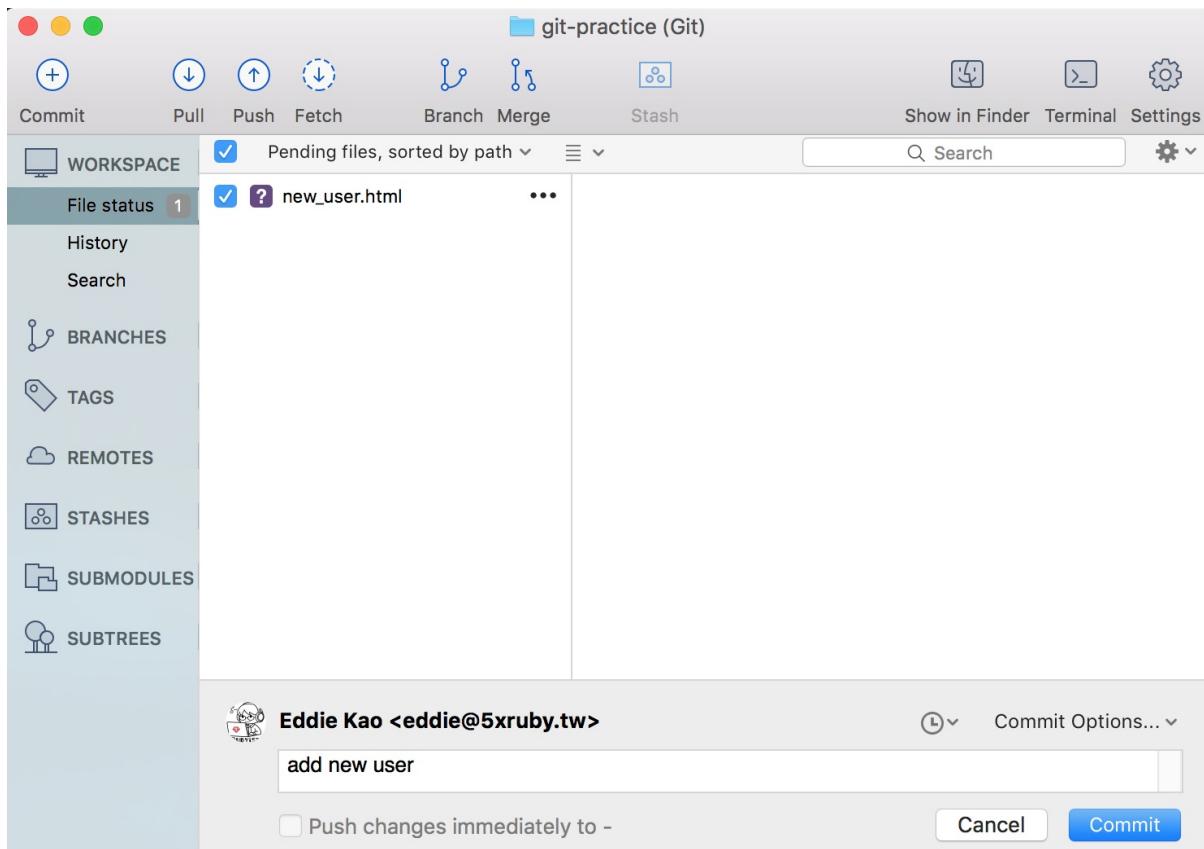
一定要二段式嗎？

會覺得要先 `add` 再 `commit` 有點囉嗦嗎？你可以在 Commit 的時候多加一個 `-a` 的參數，縮短這個流程：

```
$ git commit -a -m "update content"
```

這樣即使沒有先 `add` 也可以完成 Commit。但要注意的是這個 `-a` 參數只對已經存在 Repository 的檔案有效，對還是新加入的檔案（也就是 Untracked file）是無效的。

或是，如果你是使用像是 SourceTree 之類的圖形介面工具，可以勾一勾檔案，填寫 Commit 訊息後，按下 Commit 按鈕就可完成提交的流程：



雖然圖形介面工具可以讓你省一些麻煩，它實際上的運作原理也是一樣的，這也是為什麼在本書一直強調的，要學好 Git 一定要了解它的運作原理。

為什麼要二段式這麼麻煩？

先 `add` 再 `commit` 這樣二段式感覺可能有點麻煩，但也是有好處的。你可以想像你有一個倉庫，在倉庫門口有個小廣場，這個廣場的概念就像跟暫存區一樣，你把要存放到倉庫的貨物先放到這邊（`git add`），然後等收集的差不多了就可以打開倉庫門，把在廣場上的貨物送進倉庫裡（`git commit`），並且記錄下來這批貨是什麼用途的、誰送來的。

當然，你也可以每來一件貨物就開倉門存一次、紀錄一次，但這樣一來開倉庫的次數就會非常多。老實說這樣也沒什麼過錯，只是因為太過零碎的 Commit，有時候要查閱紀錄可能會有點不太方便；另外，過於零碎的 Commit 也可能給你的同事帶來一些困擾，例如要進行 Code

Review 的時候，比較有整理的 Commit 可以一次看到比較完整的內容，而不需要一個一個 Commit 慢慢看。

所以，什麼時候要 Commit？

這個問題沒有標準答案，你可以很多檔案修改好再一口氣全部一起 Commit，也可只改一個字就 Commit。常見的 Commit 的時間點有：

1. 完成一個「任務」的時候：不管是大到完成一整個電子商務的金流系統，還是小至只加了一個頁面甚至只是改幾個字，都算是「任務」。
2. 下班的時候：雖然可能還沒完全搞定任務，但至少先 Commit 今天的進度，除了備份之外，也讓公司知道你今天有在努力工作。（然後帶回家繼續苦命的做？）
3. 你想要 Commit 的時候就可以 Commit。

檢視紀錄

接下來，我們來看看怎麼檢視之前 Commit 的紀錄吧。

因為目前我們只有一次的 Commit，所以這裡我很快的再新增另一次的 Commit，順便也複習一下上一章介紹的內容：

```
$ touch index.html          # 建立檔案 index.html
$ git add index.html        # 把 index.html 加至暫存區
$ git commit -m "create index page" # 進行 Commit
```

使用 Git 指令

檢視 Git 紀錄，使用的是 `git log` 指令。這個指令執行後的結果看起來會像這樣：

```
$ git log
commit cef6e4017eb1a16a7bb3434f12d9008ff83a821a (HEAD -> master)
Author: Eddie Kao <eddie@5xruby.tw>
Date:   Wed Aug 2 03:02:37 2017 +0800

    create index page

commit cc797cdb7c7a337824a25075e0dbe0bc7c703a1e
Author: Eddie Kao <eddie@5xruby.tw>
Date:   Sun Jul 30 05:04:05 2017 +0800

    init commit
```

越新的資訊會在越上面，從這段訊息，大概可以看得出來：

1. Commit 作者是誰。 (人是誰殺的)
2. 什麼時候 Commit 的。 (什麼時候殺的)
3. 每次的 Commit 大概做了些什麼事。 (怎麼殺的)

至於那個看起來像亂碼的 `cef6e4017eb1a16a7bb3434f12d9008ff83a821a`，其實是有著特殊用意的。在 Git 裡每串這種看起來像亂碼的文字，都是使用 SHA-1 (Secure Hash Algorithm 1) 演算法所計算的結果。計算的方式會在「[【冷知識】那個長得很像亂碼 SHA-1 是怎麼算出來的？](#)」章節有更詳細的介紹，現在可先把它當做一種重複機率非常非常非~~~常低的一段文字（比連中好幾期樂透的機率還低）。Git 使用這樣的字串做為識別。每個 Commit 都有一個這樣的值，你可把它想像成是每個 Commit 的身份證字號一樣，不會重複。

注意！

這邊用身份證字號舉例其實不太精準，因為在台灣身份證字號是會重複的，而且重複的機會還滿高的，遠比 SHA-1 重複的機會高得多。

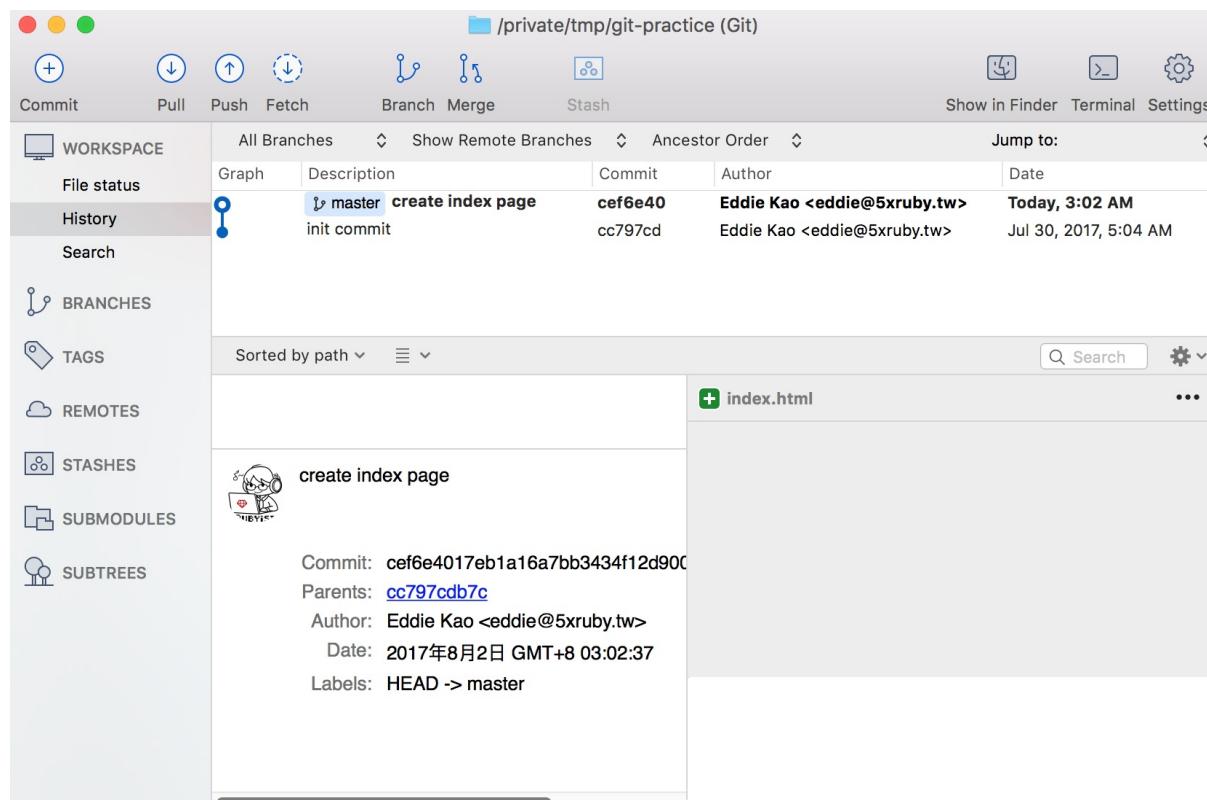
在使用 `git log` 指令時，如果加上額外參數，可以看到不一樣的輸出格式，例如可加上 `--oneline` 跟 `--graph`：

```
$ git log --oneline --graph
* cef6e40 create index page
* cc797cd init commit
```

輸出的結果就會更為精簡，可以一次看到更多的 Commit。

使用圖形介面（推薦！）

如果使用 SourceTree，點擊左邊的「History」選項就可以看到所有的歷史紀錄：



我想這應該沒什麼好爭論的，使用圖形介面工具，比在終端機視窗使用 Git 指令清楚多了。所以我常會建議初學者（甚至我自己也是），在觀看紀錄的時候，使用圖形介面工具不僅可少打一些字，而且可以顯示更完整的資訊。

再更仔細的研究一下畫面上的資訊：

Push	Fetch	Branch	Merge	Stash
All Branches		Show Remote Branches		Ancestor Order
Graph	Description	Commit	Author	
	↳ master create index page init commit	cef6e40 cc797cd	Eddie Kao <eddie@5xruby.tw> Eddie Kao <eddie@5xruby.tw>	

1. 越新的 Commit 會在越上面。
2. 在 Description 欄位除了 Commit 訊息之外，還有一個「master」字樣，這是在 Git 裡預設的分支的名字。關於分支會在「使用分支」章節介紹。
3. 在這個區塊，這兩次的 Commit 分別在畫面上是以藍色的實心小圓圈呈現，而空心的小圓圈則是表示 HEAD 的位置。HEAD 通常是指向現在這個分支的最前端的地方，更多關於 HEAD 的說明，可參閱「[【冷知識】HEAD 是什麼東西？](#)」章節說明。
4. 在 Commit 欄位，可以看到 `cef6e40` 跟 `cc797cd`，它其實就是 `cef6e4017eb1a16a7bb3434f12d9008ff83a821a` 跟 `cc797cdb7c7a337824a25075e0dbe0bc7c703a1e` 這兩串文字的「縮寫」。對 Git 來說，其實只要這 6 ~ 8 碼的 Commit 資訊，就足以識別了。那如果這個前 6 ~ 8 碼的縮寫真的重複了怎麼辦？別擔心，Git 會很貼心的告訴你「這個資訊沒辦法識別，請再提供更多位數的資訊以供識別」。

狀況題

以下是一些在使用 Git 查詢歷史紀錄的時候可能遇到的問題，這是目前的歷史紀錄：

Graph	Description	Commit	Author
	↳ master add fish	db3bbec	Eddie Kao <eddie@5xruby.tw>
	add pig	930feb3	Sherly <sherly@5xruby.tw>
	add lion and tiger	51d54ff	Sherly <sherly@5xruby.tw>
	add dog 2	27f6ed6	Eddie Kao <eddie@5xruby.tw>
	add dog 1	2bab3e7	Eddie Kao <eddie@5xruby.tw>
	add 2 cats	ca40fc9	Eddie Kao <eddie@5xruby.tw>
	add cat 2	1de2076	Eddie Kao <eddie@5xruby.tw>
	add cat 1	cd82f29	Eddie Kao <eddie@5xruby.tw>
	add database settings	382a2a5	Eddie Kao <eddie@5xruby.tw>
	init commit	bb0c9c2	Eddie Kao <eddie@5xruby.tw>

【狀況題】我想要找某個人或某些人的 Commit...

例如我只想找一位叫做 Sherly 的作者的 Commit：

```
$ git log --oneline --author="Sherly"  
930feb3 add pig  
51d54ff add lion and tiger
```

只會查到這兩個。同時可以再用 `|` (中文「或者」的意思) 來查詢「Sherly 以及 Eddie 這兩個人的 Commit 紀錄」：

```
$ git log --oneline --author="Sherly\|Eddie"
```

【狀況題】我想要找 Commit 訊息裡面有在罵髒話的...

使用 `--grep` 參數，可以從 Commit 訊息裡面搜尋符合字樣的內容：

```
$ git log --oneline --grep="WTF"
```

還好目前看起來沒有！

【狀況題】我要怎麼找到哪些 Commit 的檔案內容有提到 "Ruby" 這個字？

使用 `-S` 參數，可以搜尋在所有 Commit 的檔案中，哪些符合特定條件的：

```
$ git log -S "Ruby"
```

【狀況題】主管：「你再混嘛！我看看你今天早上 Commit 了什麼！」

在查歷史資料的時候，可以搭配 `--since` 跟 `--until` 參數查詢：

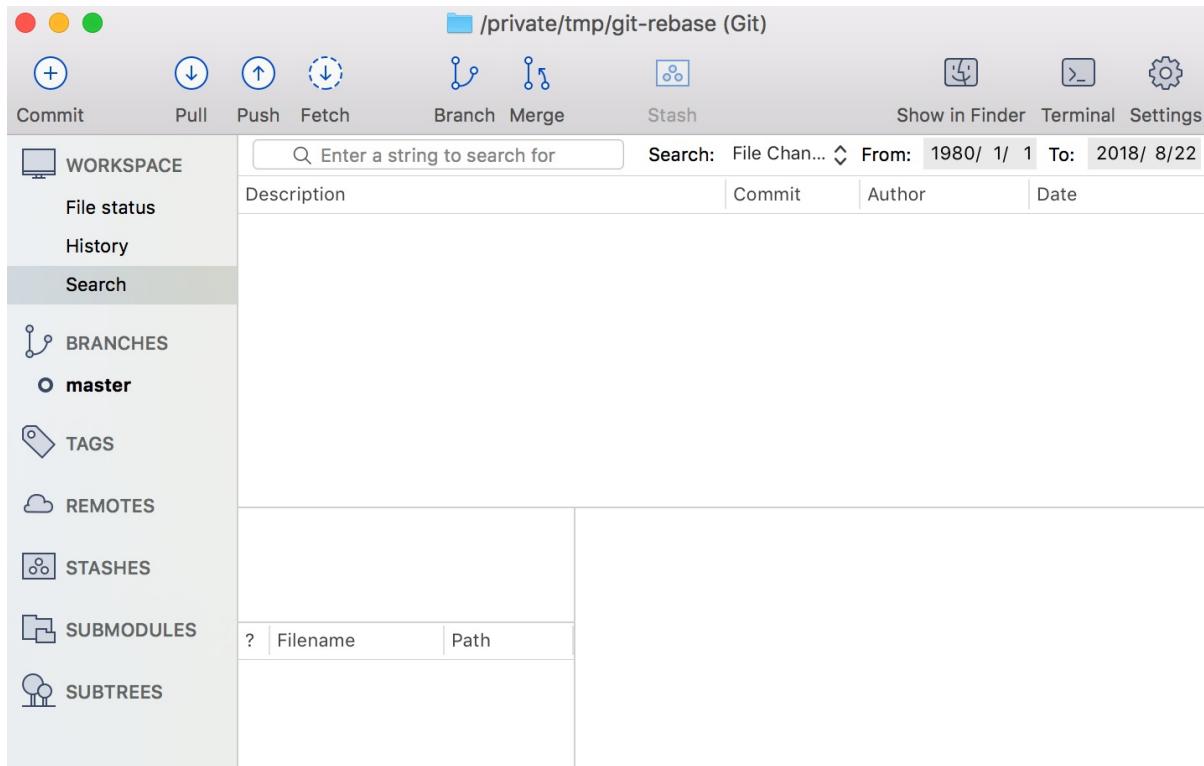
```
$ git log --oneline --since="9am" --until="12am"
```

這樣就可以找出「今天早上 9 點到 12 點之間所有的 Commit」。還可以再加一個 `after`：

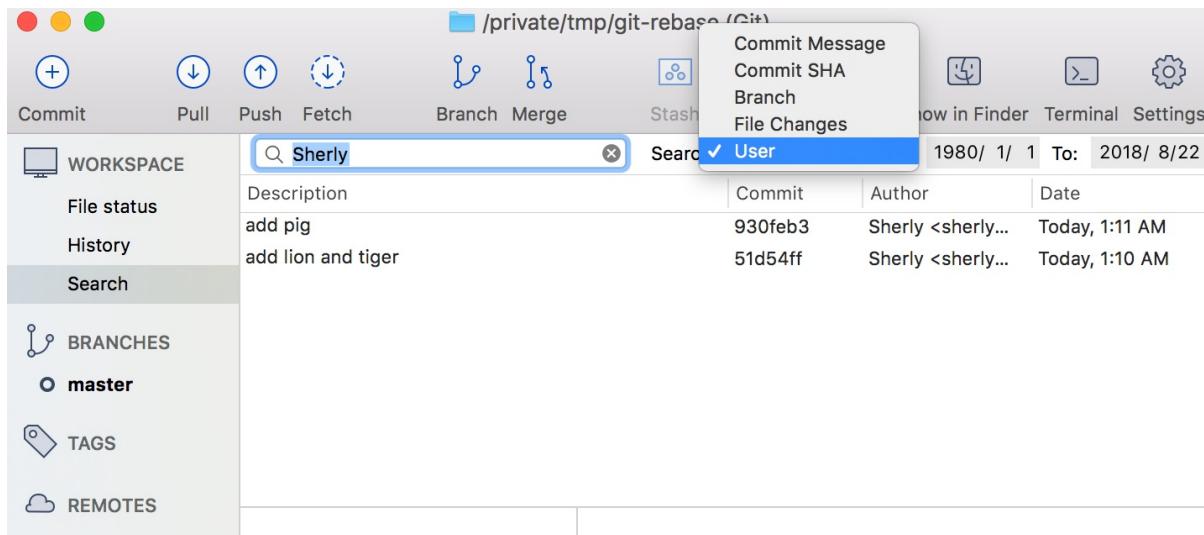
```
$ git log --oneline --since="9am" --until="12am" --after="2017-01"
```

這樣可以找到「從 2017 年 1 月之後，每天早上 9 點到 12 點的 Commit」

以上這些指令，在 SourceTree 都可以很方便的完成。點選左邊的側邊選單，在 WORKSPACE 選單下有個「Search」功能，即可進入搜尋畫面：



然後便可輸入關鍵字或是條件，例如我想找作者叫「Sherly」的人：



很快就可以查到想要查詢的資料了，相當方便吧！

【狀況題】如何在 Git 裡刪除檔案或變更檔名？

在 Git 裡，不管是刪除檔案或是變更檔名，對 Git 來說都是一種「修改」。

刪除檔案

直接砍

你可以使用系統指令 `rm` 或是檔案總管之類的工具來刪除檔案，例如：

```
$ rm welcome.html          # 刪除檔案 welcome.html
```

然後看一下狀態：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:    welcome.html

no changes added to commit (use "git add" and/or "git commit -a")
```

可以看到 `welcome.html` 這個檔案目前的狀態是 `deleted`。如果你確定這是你想做的，就可以把這次的「修改」加到暫存區：

```
$ git add welcome.html
```

再看一下目前的狀態：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    welcome.html
```

它現在的狀態是 `deleted`，而且已被加至暫存區，所以接下來就可以進行 Commit 了。我知道「把刪除檔案加到暫存區」這件事感覺有點不太直覺，就把「刪除檔案」也當做是一種「修改」看待就行了。

請 Git 幫你砍

像這樣先 `rm` 刪除然後再 `git add` 加入暫存區的兩段式動作，事實上可以直接使用 `git rm` 指令：

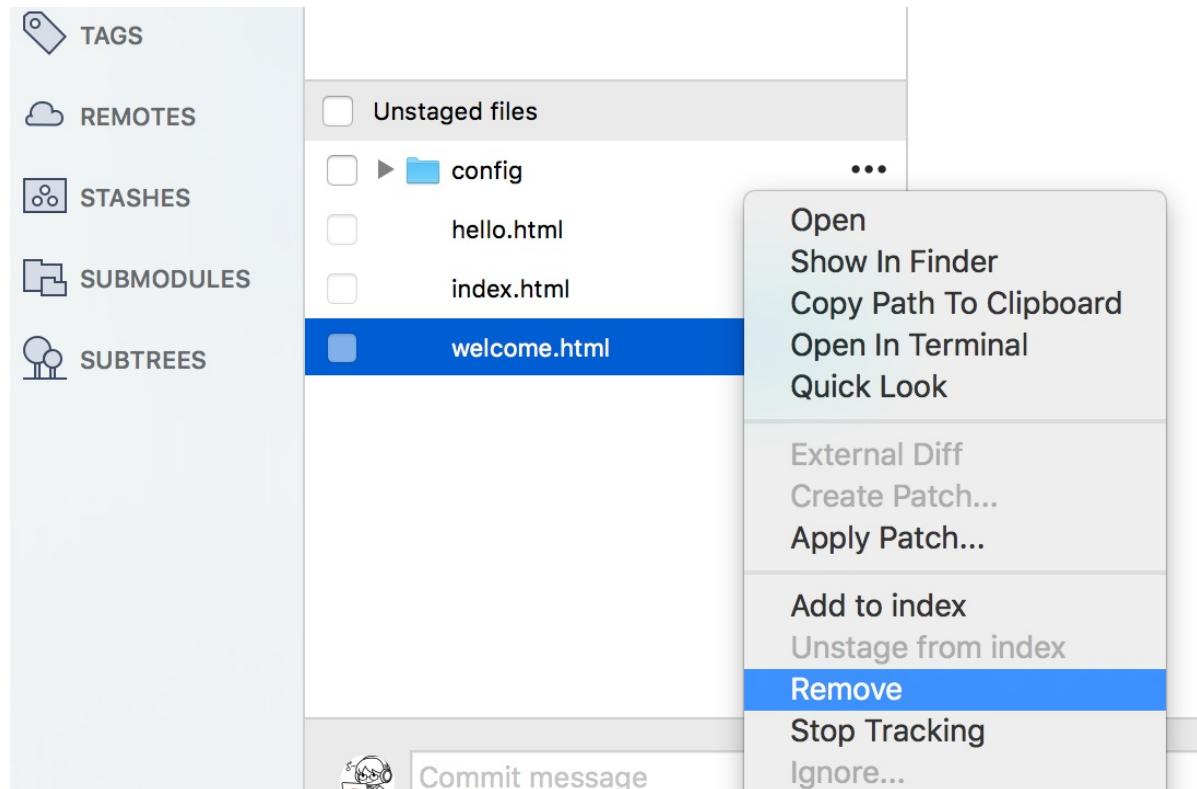
```
$ git rm welcome.html  
rm 'welcome.html'
```

這時候看狀態會發現：

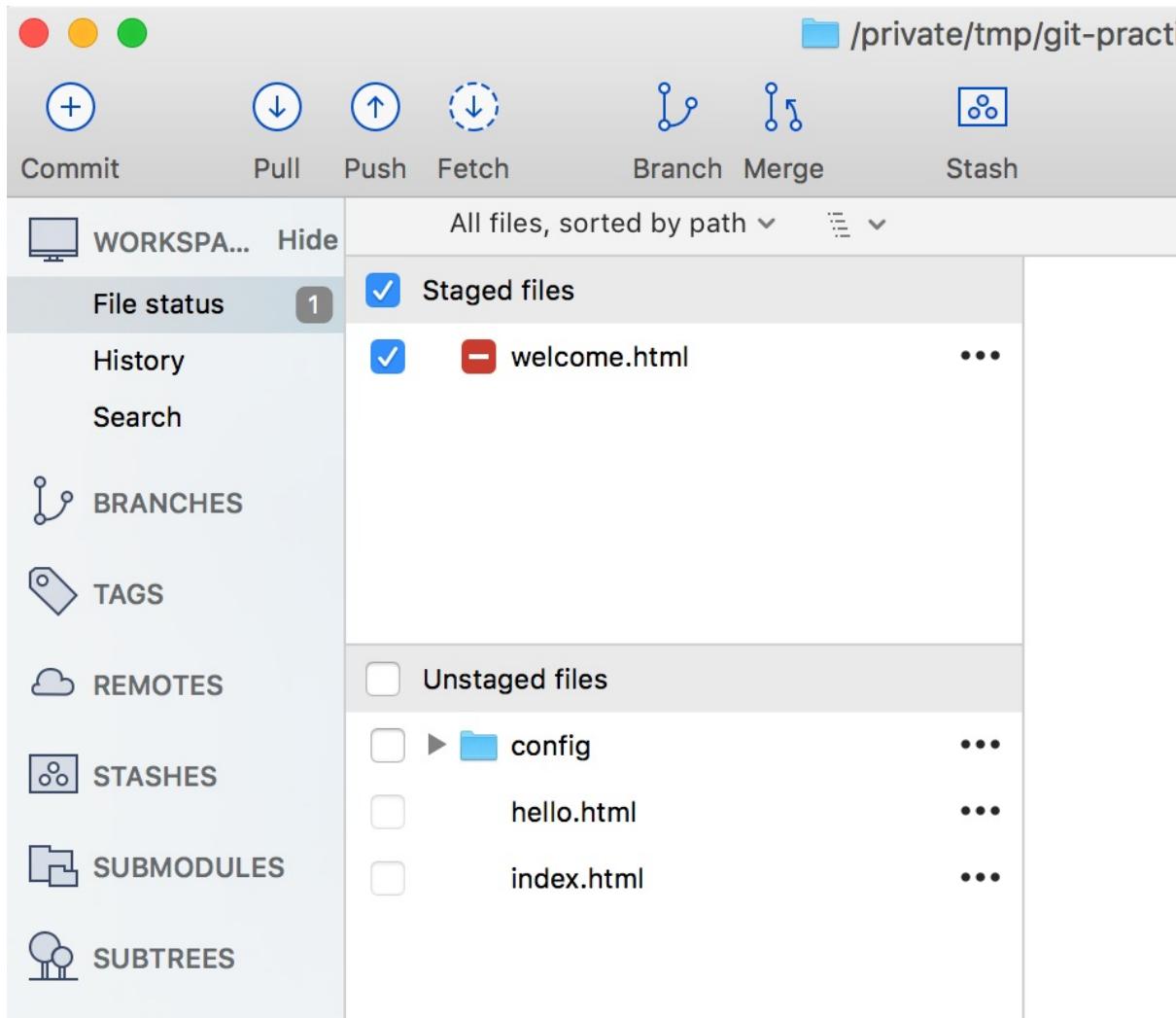
```
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
      deleted:    welcome.html
```

它就直接在暫存區了，不需要再自己 add 一次，可以少做一個步驟。

使用 SourceTree 來做這件事也是很輕鬆的，只要在檔案上按右鍵，選擇「Remove」即可：



即可有跟 `git rm` 同樣的效果，`welcome.html` 檔案已被標記為刪除記號並且放置在暫存區：



加上 `--cached` 參數

不管是系統的 `rm` 或是 `git rm` 指令，都會真的把這個檔案從工作目錄裡刪掉，但如果只是「我不是真的想把這個檔案刪掉，只是不想讓這個檔案再被 Git 控管了」的話，可以加上 `--cached` 參數：

```
$ git rm welcome.html --cached  
rm 'welcome.html'
```

別擔心，這個訊息並不是真的把檔案刪了，而僅是把檔案從 Git 裡移掉而已。這時候的狀態會變成：

```
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)
```

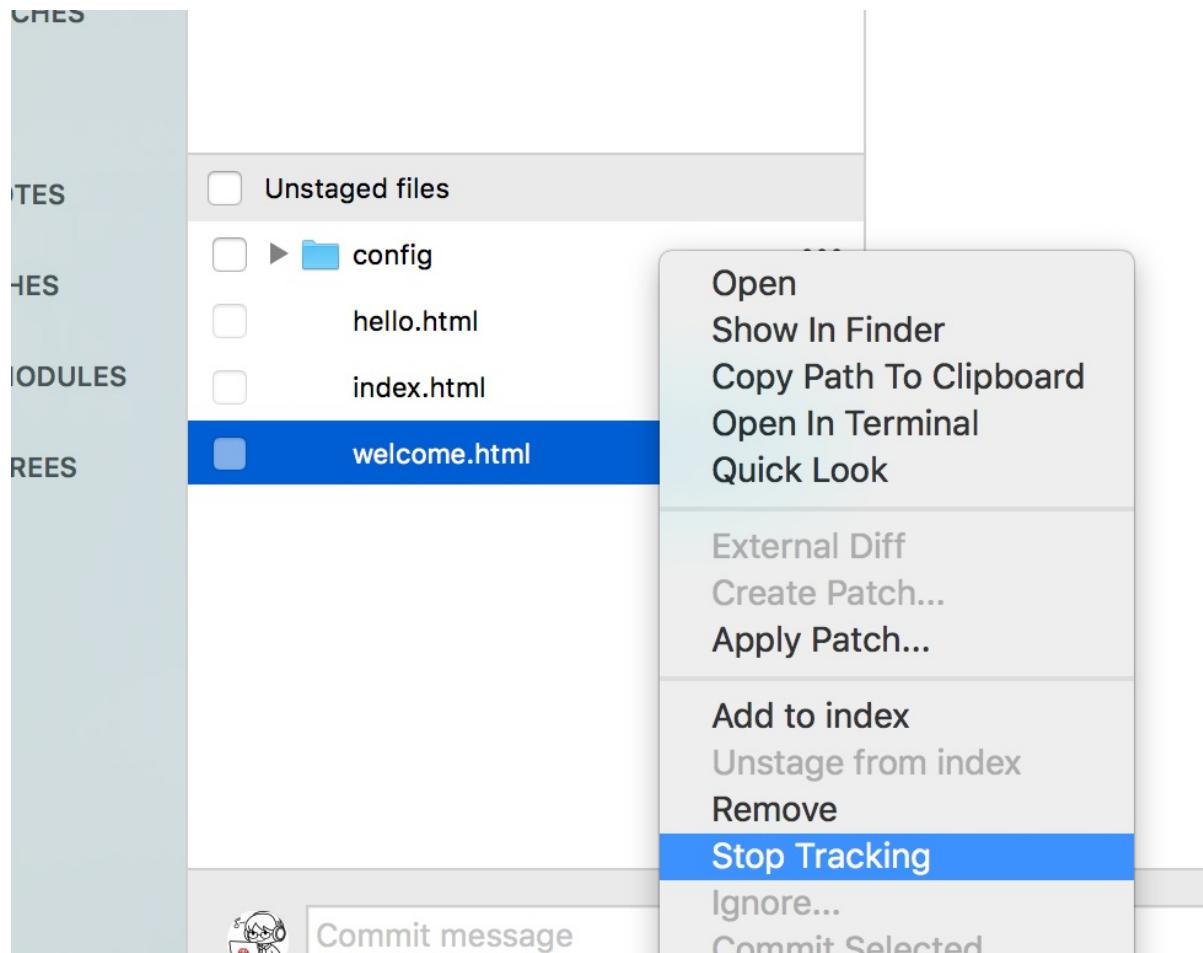
```
deleted:    welcome.html

Untracked files:
(use "git add <file>..." to include in what will be committed)

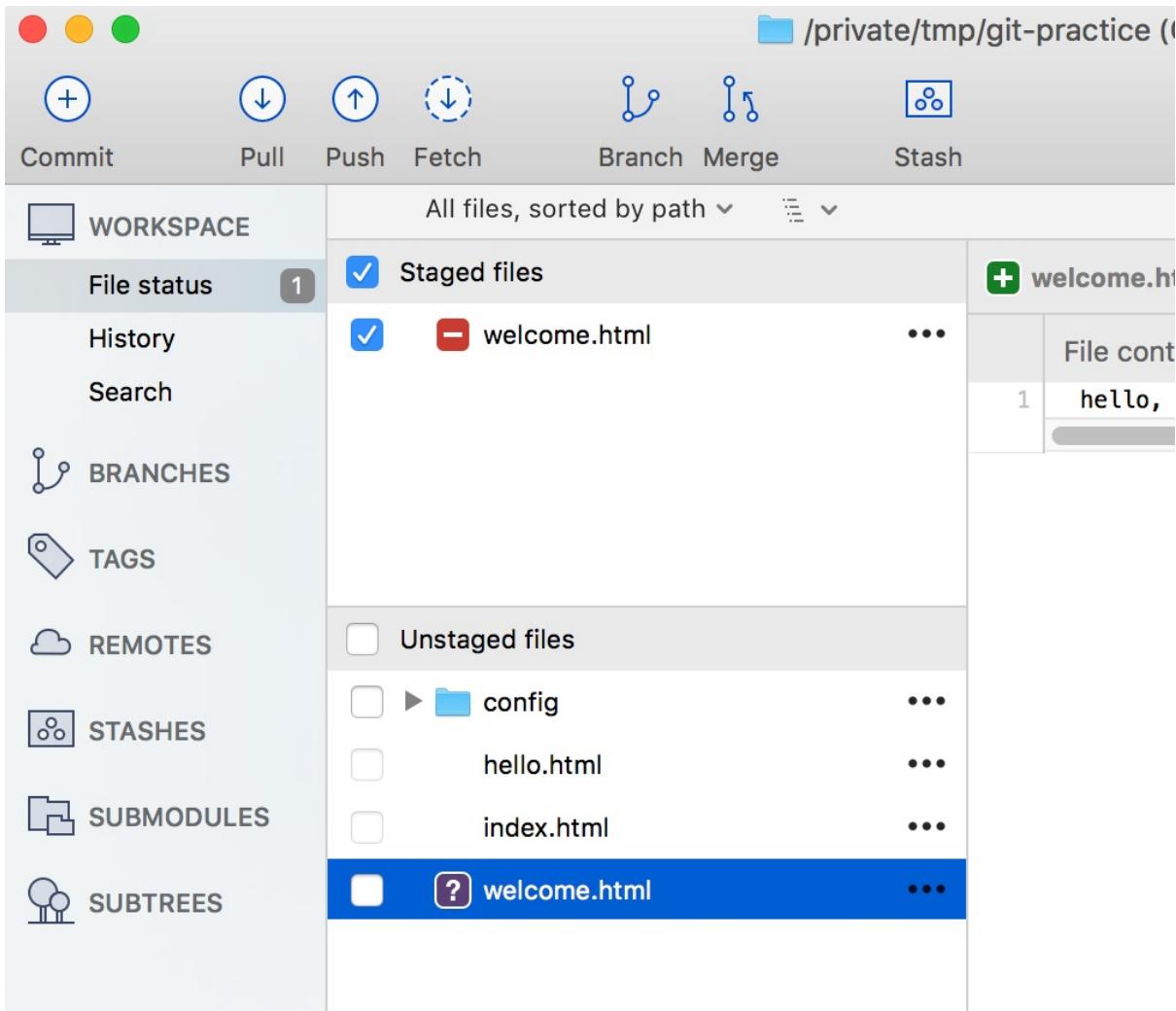
welcome.html
```

welcome.html 的狀態從原本已經在 Git 目錄裡的 tracked 變成 untracked 了。

若使用 SourceTree 則是只要在檔案上按右鍵，選擇「Stop Tracking」：



這樣就等同 `git rm --cached` 的效果，可以看到在暫存區有刪除檔案的標記，同時因為檔案已變成 untracked 狀態，所以在檔案前面打上了一個問號標記：



變更檔名

直接改名

跟刪除檔案一樣，變更檔名也是一種「修改」，所以操作上其實也是差不多的：

```
$ mv hello.html world.html      # 把 hello.html 改成 world.html
```

這時候看一下狀態，會看到兩筆狀態的改變：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:    hello.html
```

```
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
    world.html  
  
no changes added to commit (use "git add" and/or "git commit -a")
```

雖然只是改檔名，但對 Git 來說會被認為是兩個動作，一個是刪除 hello.html 檔案，一個是新增 world.html 檔案（變成 Untracked 狀態）。接著繼續使用 `git add` 指令把這些異動加至暫存區：

```
$ git add --all  
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
    renamed:    hello.html -> world.html
```

因為檔案的內容沒有改變，Git 猜得出來這個只是單純的改名字，所以現在它的狀態變成 `renamed` 了。

請 Git 幫你改名

跟前面的 `git rm` 一樣，Git 也有提供類似的指令可以讓你少做一步：

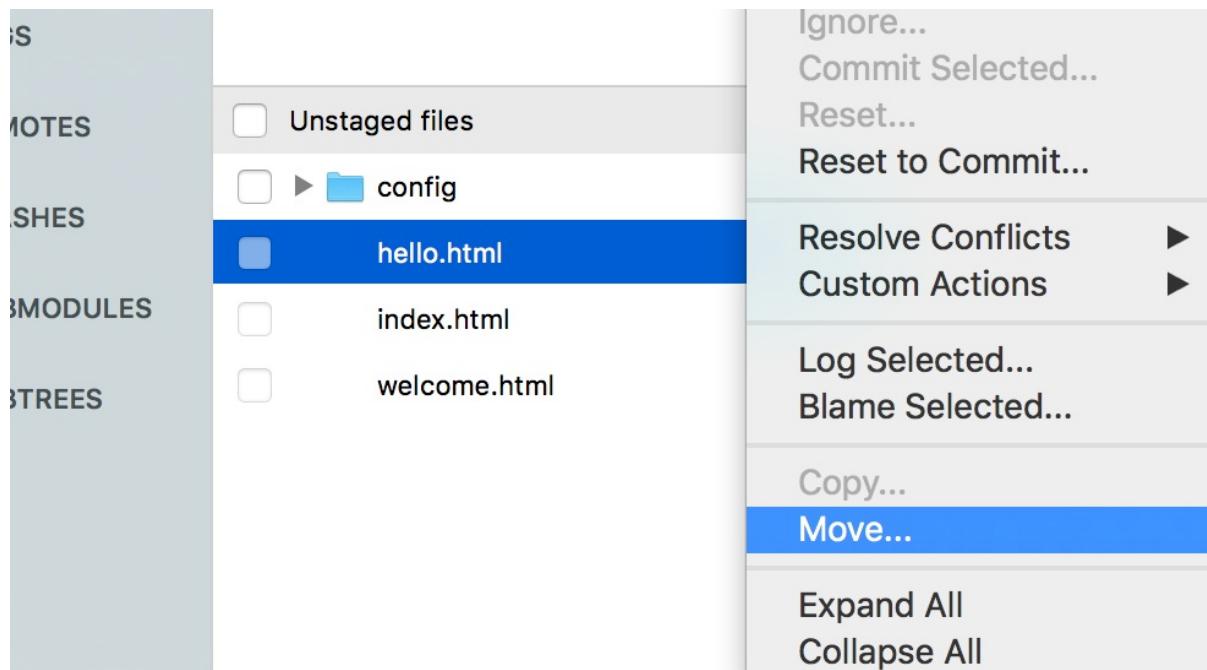
```
$ git mv hello.html world.html
```

看一下狀態：

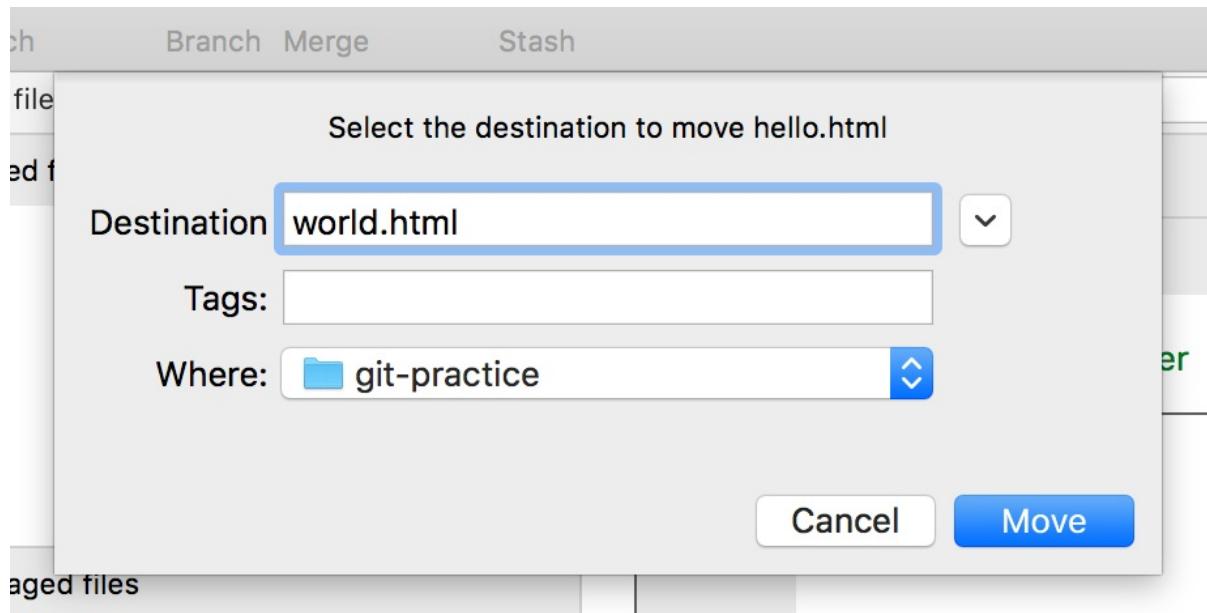
```
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
    renamed:    hello.html -> world.html
```

它的狀態會直接變成 `renamed` 了。

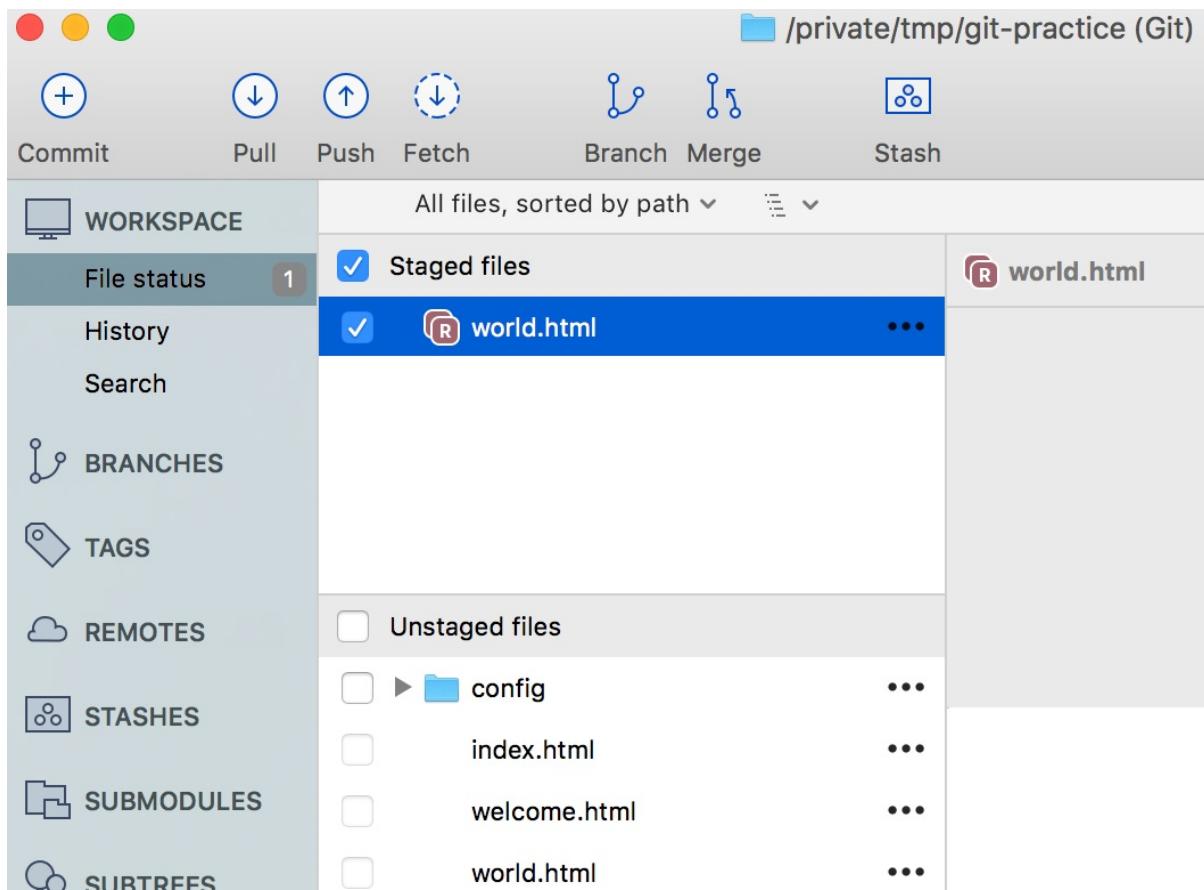
使用 SourceTree 的話，同樣是在檔案上按右鍵，選擇「Move」功能：



接著便會跳出一個對話框，輸入要改的檔名：



完成後，會在前面標記一個 R 字樣：



【冷知識】其實 Git 根本不在乎你的檔案叫什麼名字！

Git 是根據檔案的「內容」去算出那個 SHA-1 的值，所以 Git 不是很在乎你的檔案叫什麼名字，只在乎檔案的內容是什麼。所以當你進行更改檔名的時候，Git 並沒有為此做出一個新的 Blob 物件，而僅是指向原來舊的那顆 Blob 物件。但因為檔名變了，所以會為此做出一顆新的 Tree 物件喔。

如果這些 Git 物件還不清楚它們是幹嘛的，在後面的「[【超冷知識】在 .git 目錄裡有什麼東西？Part 1](#)」章節會有詳細的說明。

【狀況題】修改 Commit 紀錄

遇到澳洲來的客人，做得很不爽，因為心情不好，不小心在 Commit 訊息罵了客戶了，要怎麼消掉？

身為工程師，偶爾總是會遇上不太順心的客人或專案，心情不好的時候，在程式碼或 Commit 訊息裡「抒發」一下情緒也是很常見的，只是這要是讓客人看見了總是不好解釋。

要修改 Commit 紀錄有好幾種方法：

1. 把 `.git` 目錄整個刪除（誤）。
2. 使用 `git rebase` 來修改歷史。
3. 先把 Commit 用 `git reset` 拆掉，整理後再重新 Commit。
4. 使用 `--amend` 參數來修改最後一次的 Commit。

這裡我們將使用第 4 種方式來修改最後一次的 Commit 訊息，第 2 跟第 3 種方式會在後面的章節陸續介紹到，至於第 1 種方式等於是砍掉重練，因為那會把該專案所有的 Git 紀錄全部清掉，除非必要，不要輕易使用。

使用 `--amend` 參數來進行 Commit

例如原來的紀錄是這個樣子：

```
$ git log --oneline
4879515 WTF
7dbc437 add hello.html
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

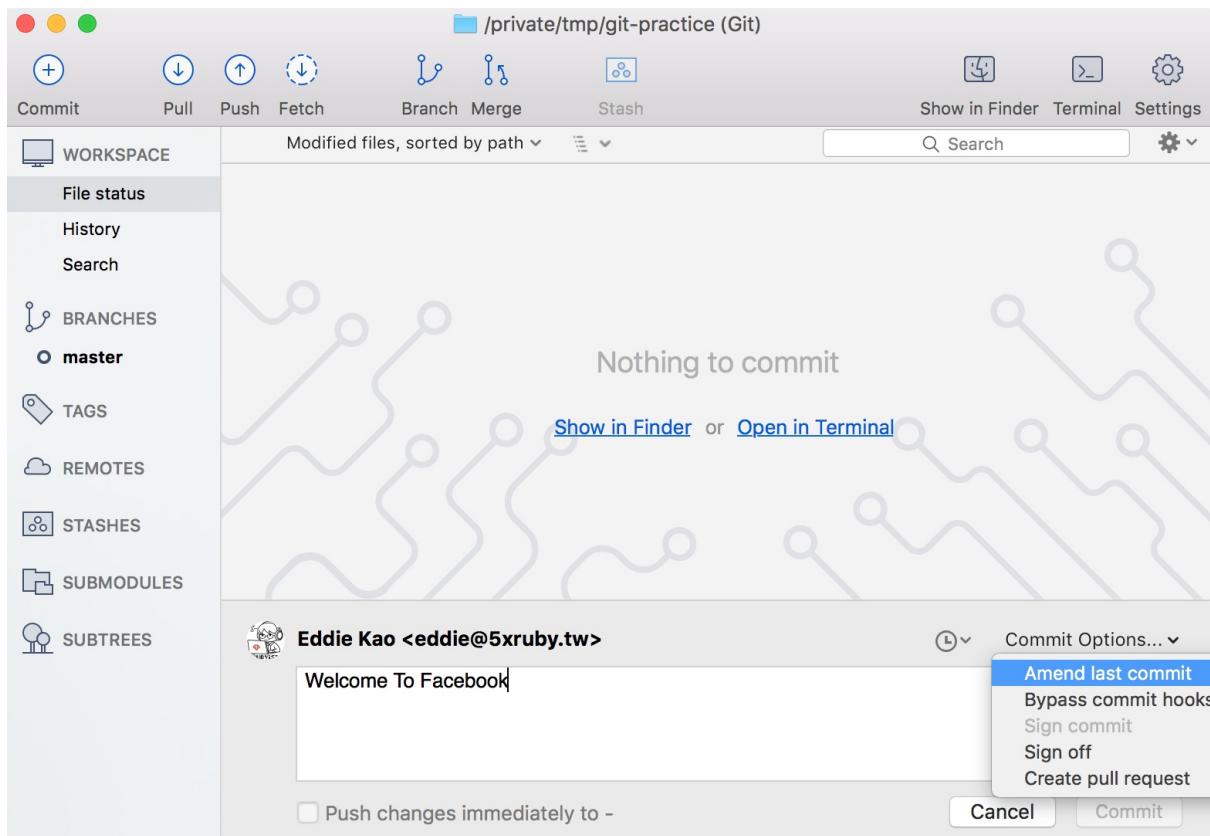
那個 "WTF" 訊息有點糟糕！要修改最後一次的 Commit 訊息，只要直接在 Commit 指令後面加上 `--amend` 參數即可：

```
$ git commit --amend -m "Welcome To Facebook"
[master 614a90c] Welcome To Facebook
Date: Wed Aug 16 05:42:56 2017 +0800
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 config/database.yml
```

如果沒有加上 `-m` 參數提供要修改的訊息，就會跳出 Vim 視窗讓你編輯訊息。再回來看紀錄，"WTF" 就被改成 "Welcome To Facebook" 了：

```
$ git log --oneline
614a90c Welcome To Facebook
7dbc437 add hello.html
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

如果是使用 SourceTree，請先點選左上角的 Commit 按鈕進行 Commit 畫面，然後在右上角的「Commit Options」選擇「Amend last commit」：



在下方的空格填寫好訊息，按下右下的 Commit 按鈕後就可修改。

你注意到了嗎？

雖然只是改紀錄的訊息，其它什麼都沒有改，但對 Git 來說因為「Commit 的內容」改變了，所以 Git 會重新計算並產生一顆新的 Commit 物件，也就是這其實是一次全新的 Commit（只是看起來不像新的）。以上面這個例子為例，修改前的 Commit 物件的 SHA-1 值是 4879515，但改完訊息之後 SHA-1 值變成 614a90c。雖然 Commit 的時間跟檔案的內容看起來並沒有被修改，但它仍是一次全新的 Commit。

那可以修改更早的紀錄嗎？

可以的，只是就得使用 Rebase 指令來處理了，`--amend` 參數只能處理最後一次而已。Rebase 指令的使用方式請參閱「[【狀況題】修改歷史訊息](#)」章節說明。

平行時空

請記得，雖然這只是改訊息，不管如何它就是修改了一次的歷史，所以請儘量不要在已經 Push 出去之後再修改，否則可能會造成其它人的困擾。

【狀況題】追加檔案到最近一次的 Commit

剛剛完成 Commit，但發現有一個檔案忘了加到，又不想為了這個檔案重新再發一次 Commit...

像上述這個情況，雖然為了這個單一檔案再加送一次 Commit 也不是不行，但有些人有 Commit 的潔癖（例如我），希望每個 Commit 不要太大也不要太小，就剛剛好做到這個 Commit 應該做的事就好。所以如果不想因此再發一次 Commit，想把這個檔案併入最近一次 Commit，可以這樣做：

1. 使用 `git reset` 把最後一次的 Commit 拆掉，加入新檔案後再重新 Commit。
2. 使用 `--amend` 參數進行 Commit。

這裡先介紹第二種方式，第一種方式會在「[【狀況題】剛才的 Commit 後悔了，想要撤掉重做...](#)」章節進行說明。

例如，我有一個檔案叫做 `cinderella.html`，想把它加到最後一次的 Commit 裡：

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    cinderella.html

nothing added to commit but untracked files present (use "git add" to track)
```

假設它剛加進來，它的狀態還是 `Untracked`。流程上一樣，還是使用 `git add` 先把檔案加到暫存區：

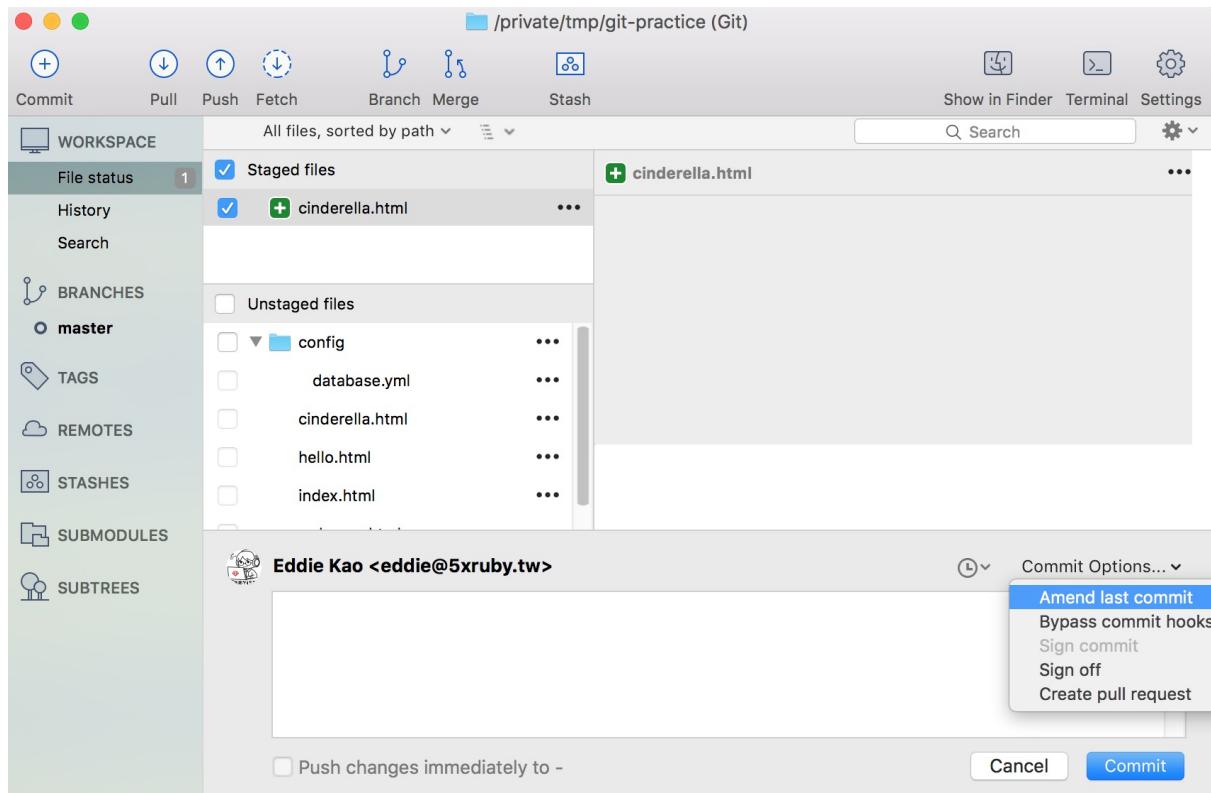
```
$ git add cinderella.html
```

接著在 Commit 的時候加上 `--amend` 參數：

```
$ git commit --amend --no-edit
[master 3128d00] update story
Date: Wed Aug 16 05:42:56 2017 +0800
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 cinderella.html
create mode 100644 config/database.yml
```

這樣就可以把檔案併入最後一次的 Commit。而最後面那個 `--no-edit` 參數的意思是指「我不需要編輯 Commit 訊息」，所以就不會跳出 Vim 編輯器的視窗。

如果使用 SourceTree，一樣也是先把 `cinderella.html` 檔案勾選至暫存區，在 Commit 的時候，右下角的下拉選單有一個「Amend last commit」選項



Commit 訊息要改不改都可，按下右下角按鈕後就會完成 Commit，這個檔案就會被併到最近的一次 Commit 裡了。

修改歷史

跟前個章節一樣的提醒，像這樣的修改歷史，請儘量不要使用在已經 Push 出去的 Commit 上。

【狀況題】新增目錄？

我剛剛新增了一個 images 目錄，但卻發現這個目錄好像沒辦法被加到 Git 裡面？

舉例來說：

```
$ git status  
On branch master  
nothing to commit, working tree clean
```

現在的狀態是剛 Commit 完，工作目錄也沒有任何修改。接著我建立一個 images 目錄：

```
$ mkdir images
```

再看一次狀態，會發現 Git 的狀態依舊沒有變化：

```
$ git status  
On branch master  
nothing to commit, working tree clean
```

請記得一件很重要的觀念，就是 Git 在計算、產生物件的時候，是根據「檔案的內容」去做計算的，所以光是新增一個目錄，Git 是沒辦法處理它的。

注意！

空的目錄無法被提交！

那怎麼辦？其實很簡單，就只要在那個空目錄裡隨便放一個檔案就行了。如果目前還沒東西可以放，或是不知道該放什麼檔案比較好，慣例上可以放一個名為 ".keep" 或 ".gitkeep" 的空檔案，讓 Git 能「感應」到這個目錄的存在：

```
$ touch images/.keep
```

然後再查看一下狀態：

```
$ git status  
On branch master  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
images/  
  
nothing added to commit but untracked files present (use "git add" to track)
```

就可以發現 Git 已經能感知到這個目錄的存在了（其實是感應到裡面那個 .keep 檔案的存在），接下來就照一般的流程來進行 add 跟 commit 就行了。

【狀況題】有些檔案我不想放在 Git 裡面...

有些比較機密的檔案不想放在 Git 裡面一起備份，例如資料庫的存取密碼或是 AWS 伺服器的存取金鑰...

不只是比較機密的檔案，有時候一些程式編譯的中間檔或暫存檔，因為每次只要一編譯就等於產生一次新的檔案，對專案來說通常沒有實質的利用價值，像這樣的檔案其實也不會想讓它進到 Git 裡。

要做到這件事，只要在專案目錄裡放一個 `.gitignore` 檔案，並且設定想要忽略的規則就行了。如果這個檔案不存在，就手動新增它：

```
$ touch .gitignore
```

然後編輯這個檔案的內容：

```
# 檔案名稱 .gitignore

# 忽略 secret.yml 檔案
secret.yml

# 忽略 config 目錄下的 database.yml 檔案
config/database.yml

# 忽略所有 db 目錄下附檔名是 .sqlite3 的檔案
/db/*.sqlite3

# 忽略所有附檔名是 .tmp 的檔案
*.tmp

# 當然你要忽略自己也可以，只是通常不會這麼做
# .gitignore
```

只要 `.gitignore` 這個檔案存在，即使這個檔案沒被 Commit 或是沒被 Push 上 Git Server 就會有效果。但這個檔案會建議 Commit 進專案並且推上 Git Server，這樣一來整個專案一起開發的人可以共享相同的設定。

在新增檔案的時候，一旦只要符合 `.gitignore` 檔案裡的規定，這個檔案就會被忽視。例如，現在的狀態是剛剛 Commit 完，暫存區跟工作目錄都是乾淨的：

```
$ git status
On branch master
nothing to commit, working tree clean
```

這時候，故意加入會被忽略的檔案 `secret.yml`：

```
$ touch secret.yml
```

再看一下狀態：

```
$ git status  
On branch master  
nothing to commit, working tree clean
```

這個檔案雖然現在確實存在這個目錄裡，但 Git 已經感應不到它了，也就是說它被 Git 無視了。

如果你不知道你在使用的工具或程式語言通常會忽略哪些檔案，GitHub 上有整理了一份各種程式語言常見的 `.gitignore` 檔案。

網址：<https://github.com/github/gitignore>

可以忽略這個忽略嗎？

雖然 `.gitignore` 這個檔案有列了一些忽略的規則，但其實也是可以忽略這個忽略的規則。只要在 `git add` 的時候再加上 `-f` 的參數：

```
$ git add -f 檔案名稱
```

就可以無視規則，強迫闖關。

咦？怎麼沒效果？

以上面這個例子來說，這個專案裡剛好有個 `config` 目錄，裡面剛好有個 `database.yml` 檔，完全符合被忽略的規則。所以照理說這個修改應該要被無視，但編輯 `database.yml` 之後卻會發現：

```
$ git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)  
  
    modified:   config/database.yml
```

狀態竟然變成 `modified` 了！不是說會無視嗎？

這是因為這個 `config/database.yml` 檔案，在 `.gitignore` 之前就存在了。`.gitignore` 檔案設定的規則，只對在規則設定之後的有效，那些已經存在的檔案就像既得利益者一樣，這些規則是對他們沒有效果的。

如果想套用 `.gitignore` 的規則，就必須先使用 `git rm --cached` 指令把這些既得利益者請出 Git，移出 Git 控管之後，它就會開始會被忽略了。

Git 教學：該如何清除忽略的檔案？

如果想要一口氣清除那些已經被忽略的檔案，可以使用 `git clean` 指令並配合 `-x` 參數：

```
$ git clean -fx
```

那個額外加上的 `-f` 參數是指強制刪除的意思，這樣一來就可清除那些被忽略的檔案。

【狀況題】檢視特定檔案的 Commit 紀錄

雖然 `git log` 可以檢視整個專案的 Commit 紀錄，但如果只想檢視單一檔案的紀錄，只要在 `git log` 後面接上那個檔名：

```
$ git log welcome.html
commit 688fef0c50004c12fe85aa139e2bf1b1aca4a38f
Author: Eddie Kao <eddie@5xruby.tw>
Date:   Thu Aug 17 03:44:58 2017 +0800

    update welcome

commit cc797cdb7c7a337824a25075e0dbe0bc7c703a1e
Author: Eddie Kao <eddie@5xruby.tw>
Date:   Sun Jul 30 05:04:05 2017 +0800

    init commit
```

這樣就能看到這個檔案 Commit 的歷史紀錄。如果想看這個檔案到底每次的 Commit 做了什麼修改，可以再給它一個 `-p` 參數：

```
$ git log -p welcome.html
commit 688fef0c50004c12fe85aa139e2bf1b1aca4a38f
Author: Eddie Kao <eddie@5xruby.tw>
Date:   Thu Aug 17 03:44:58 2017 +0800

    update welcome

diff --git a/welcome.html b/welcome.html
index 94bab17..edc805c 100644
--- a/welcome.html
+++ b/welcome.html
@@ -1,3 @@
hello, git
+
+Welcome to Git

commit cc797cdb7c7a337824a25075e0dbe0bc7c703a1e
Author: Eddie Kao <eddie@5xruby.tw>
Date:   Sun Jul 30 05:04:05 2017 +0800

    init commit

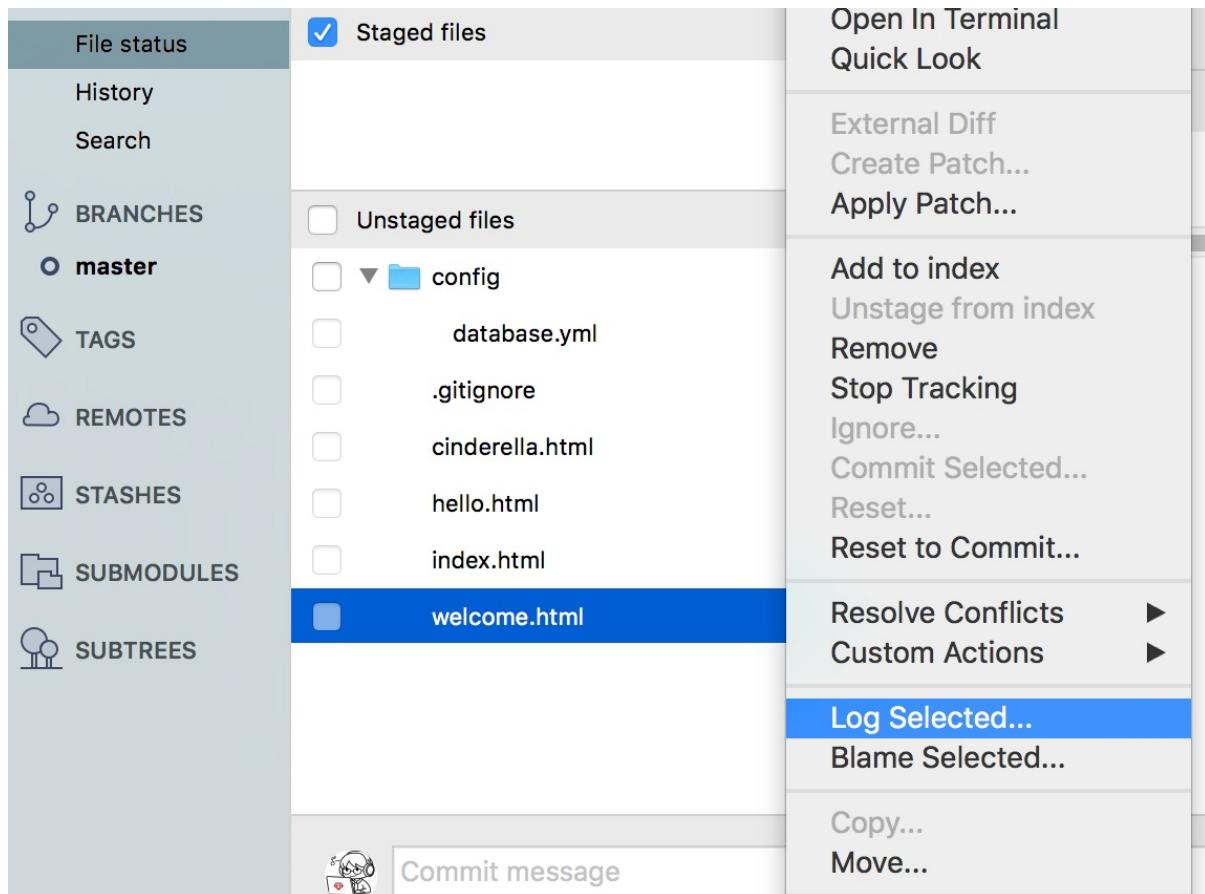
diff --git a/welcome.html b/welcome.html
new file mode 100644
index 0000000..94bab17
--- /dev/null
+++ b/welcome.html
@@ -0,0 +1 @@
+hello, git
```

格式可能看起來有點複雜，但大概可以看得出來在 "init commit" 那次的 Commit 只有加一行 "hello, git"，而 "update welcome" 那次 Commit 則是再新增了一行 "Welcome to Git"。

小提示

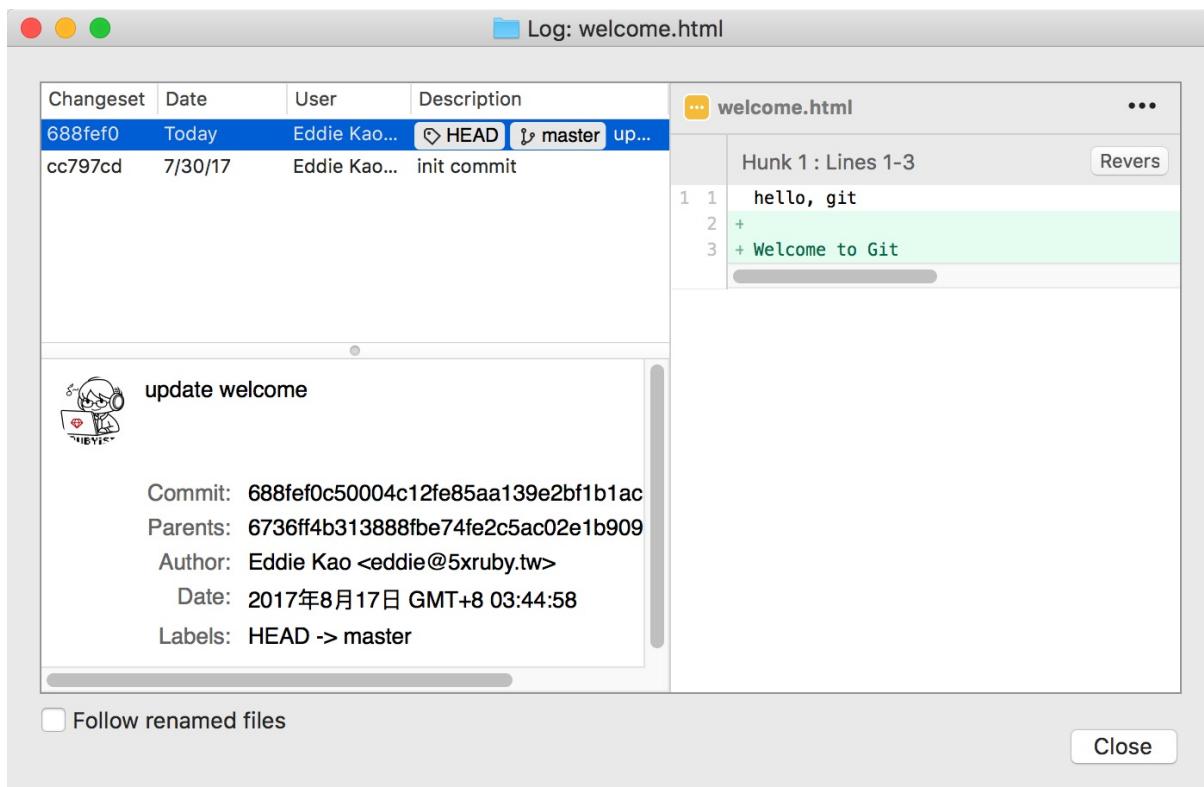
前面的加號 + 表示是新增的內容，如果是減號 - 表示原本的內容被刪除

使用 SourceTree 的話，可在指定的檔案上按右鍵，選擇「Log Selected」功能：



即可看到這個單一檔案的 Commit 紀錄：

【狀況題】檢視特定檔案的 Commit 紀錄



每次 Commit 修改了什麼在右邊的小視窗也都看得到。

【狀況題】等等，這行程式誰寫的？

啊！網站怎麼掛了？！這行程式碼是誰寫的？！

這種狀況應該常常發生，想要知道某個檔案的某一行是誰寫的嗎？在 Git 可使用 `git blame` 指令幫你抓出兇手：

```
$ git blame index.html
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 1) <!DOCTYPE html>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 2) <html>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 3)   <head>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 4)     <meta charset="utf-8">
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 5)     <title>首頁</title>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 6)   </head>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 7)   <body>
7dc9302e (Eddie Kao 2019-05-28 16:00:49 +0800 8)     <div class="container">
7dc9302e (Eddie Kao 2019-05-28 16:00:49 +0800 9)   </div>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 10)  </body>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 11) </html>
```

這樣可以很清楚的看得出來哪一行是誰在什麼時候寫的，而最前面看起來像亂碼的文字，正是每次 Commit 的識別代碼。表示這一行是在哪一次的 Commit 裡。以這個例子來說，除了第 8 跟第 9 行之外（`7dc9302e`），其它的都是在同一個 Commit 裡加進來的（`6783cc21`）。

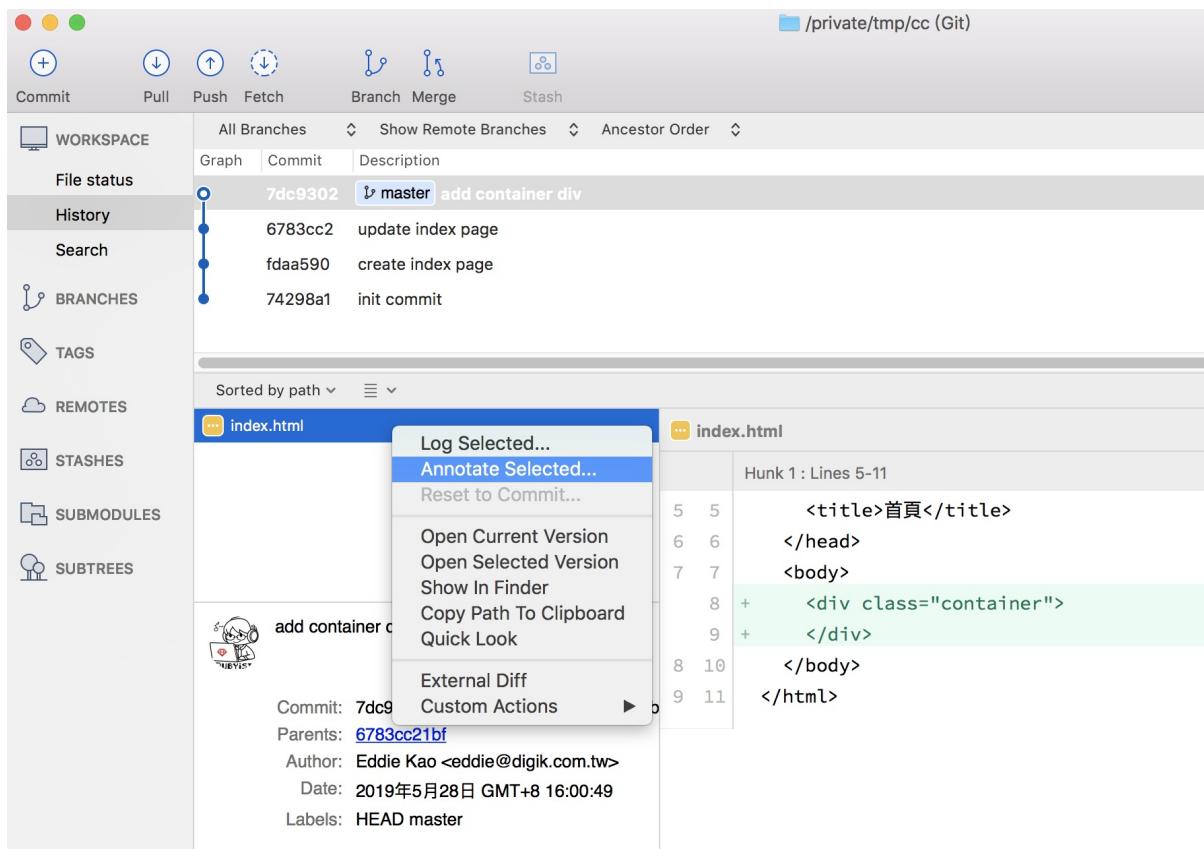
如果檔案太大，也可加上 `-L` 參數，只顯示指定行數的內容：

```
$ git blame -L 5,10 index.html
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 5)   <title>首頁</title>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 6)   </head>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 7)   <body>
7dc9302e (Eddie Kao 2019-05-28 16:00:49 +0800 8)     <div class="container">
7dc9302e (Eddie Kao 2019-05-28 16:00:49 +0800 9)   </div>
6783cc21 (Eddie Kao 2019-05-28 16:00:35 +0800 10) </body>
```

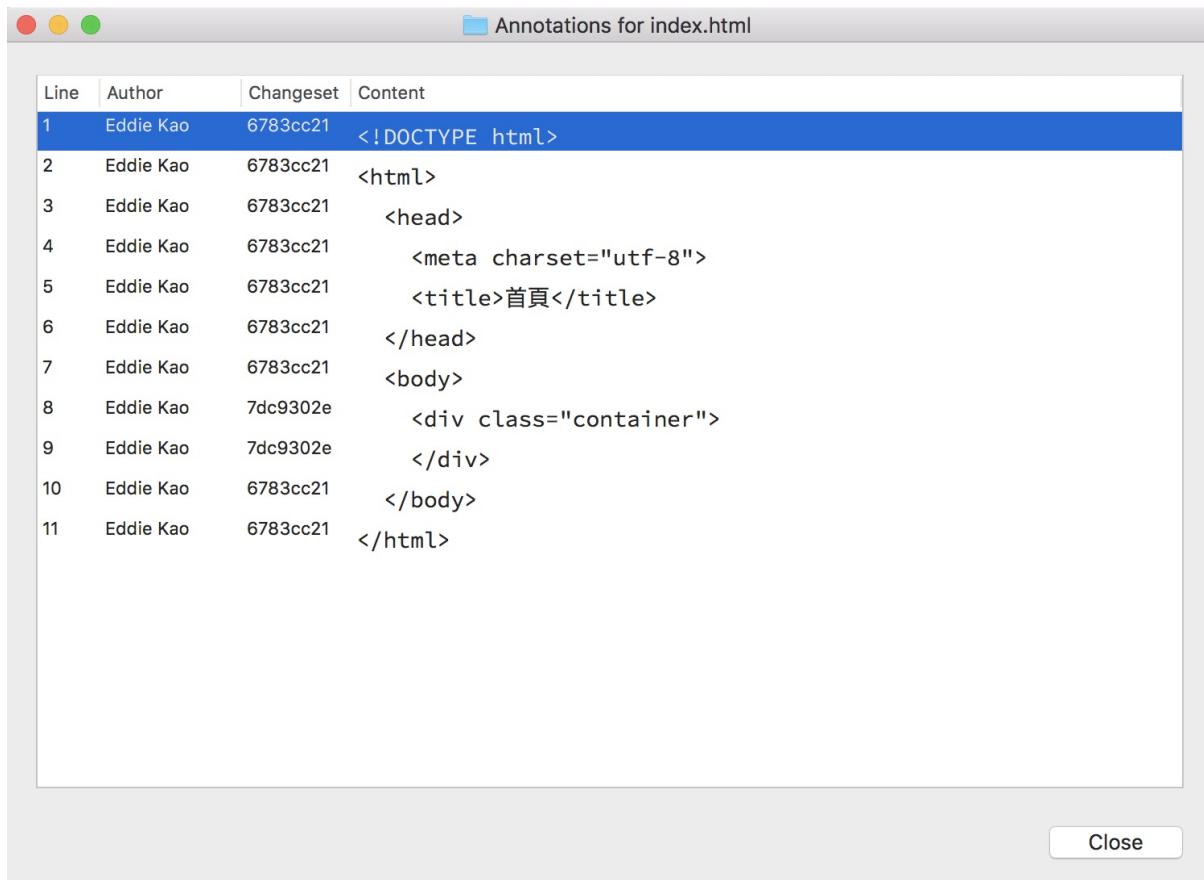
這樣就只會顯示第 5 ~ 10 行的資訊。

如果使用 SourceTree，可在你想檢視的那個檔案上按滑鼠右鍵：

【狀況題】等等，這行程式誰寫的？



選擇「Annotate Selected」，就可以看到這個畫面：



基本上看到的資訊跟使用指令差不多。

謎之音

很多時候，`git blame` 抓到的兇手大多都是自己！

【狀況題】啊！不小心把檔案或目錄刪掉了...

貼心小提示

本章節用到的 `rm` 指令使用請小心，請不要在精神不好的時候使用，以免造成連 Git 都救不了你的悲劇。

人有失蹄馬有失手，總是有不小心或沒睡飽精神不好的時候。不管是有心或無心，在 Git 裡面如果不小心把檔案或目錄刪掉是救得回來的，這也是我們之所以使用版本控制系統最主要原因之一。這裡我先故意使用 `rm` 指令，把專案裡所有的 HTML 檔刪掉：

```
$ rm *.html

$ ls -al
total 8
drwxr-xr-x  6 eddie  wheel  204 Aug 17 04:38 .
drwxrwxrwt  65 root   wheel  2210 Aug 17 03:38 ..
drwxr-xr-x  15 eddie  wheel  510 Aug 17 04:44 .git
-rw-r--r--  1 eddie  wheel  232 Aug 16 17:14 .gitignore
drwxr-xr-x  3 eddie  wheel  102 Aug 16 16:45 config
drwxr-xr-x  2 eddie  wheel   68 Aug 16 17:19 images
```

的確可以看到所有的 HTML 檔都不見了。接著看一下目前的 Git 狀態：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:    cinderella.html
    deleted:    index.html
    deleted:    welcome.html
    deleted:    world.html

no changes added to commit (use "git add" and/or "git commit -a")
```

看得出來目前這四個檔案都是處於被刪除（`deleted`）的狀態。

別擔心，這時如果要把 `cinderella.html` 救回來，可以使用 `git checkout` 指令：

```
$ git checkout cinderella.html
```

看一下檔案列表：

```
$ ls -al
total 8
drwxr-xr-x  7 eddie  wheel   238 Aug 17 04:46 .
drwxrwxrwt  65 root   wheel  2210 Aug 17 04:45 ..
drwxr-xr-x  15 eddie  wheel   510 Aug 17 04:46 .git
-rw-r--r--   1 eddie  wheel    232 Aug 16 17:14 .gitignore
-rw-r--r--   1 eddie  wheel     0 Aug 17 04:46 cinderella.html
drwxr-xr-x   3 eddie  wheel   102 Aug 16 16:45 config
drwxr-xr-x   2 eddie  wheel    68 Aug 16 17:19 images
```

那個檔案回來了！如果想一口氣把所有被刪掉的檔案救回來：

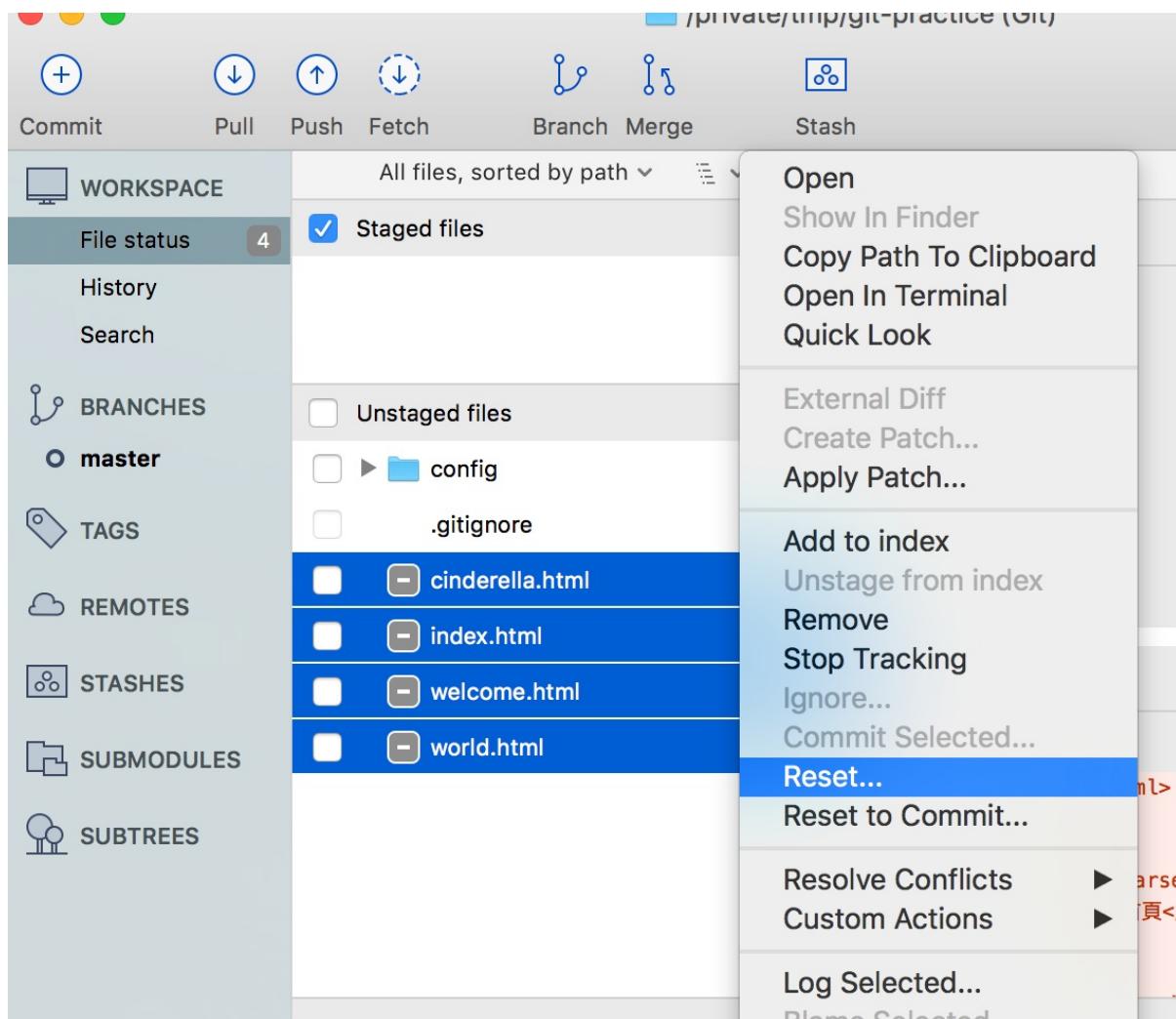
```
$ git checkout .
```

看一下檔案列表：

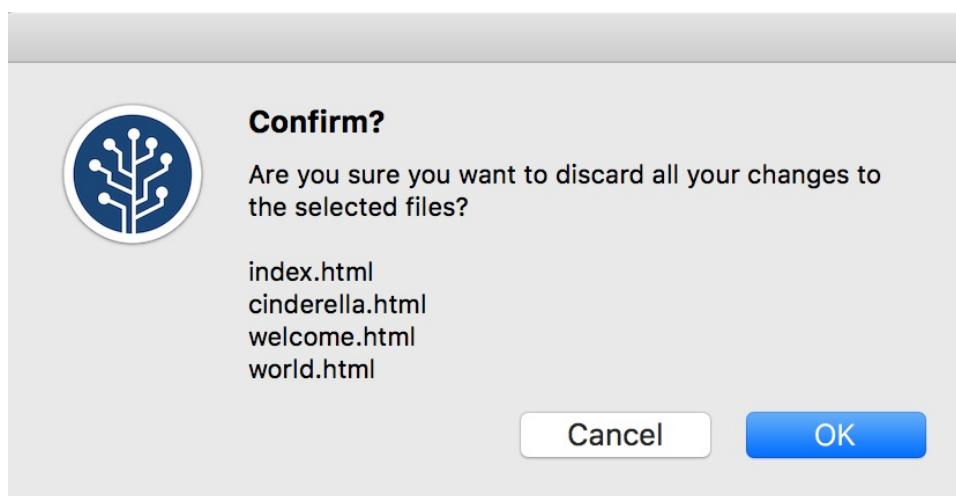
```
$ ls -al
total 24
drwxr-xr-x 10 eddie  wheel   340 Aug 17 05:34 .
drwxrwxrwt 65 root   wheel  2210 Aug 17 04:45 ..
drwxr-xr-x 15 eddie  wheel   510 Aug 17 05:34 .git
-rw-r--r--  1 eddie  wheel    232 Aug 16 17:14 .gitignore
-rw-r--r--  1 eddie  wheel     0 Aug 17 04:46 cinderella.html
drwxr-xr-x  3 eddie  wheel   102 Aug 16 16:45 config
drwxr-xr-x  2 eddie  wheel    68 Aug 16 17:19 images
-rw-r--r--  1 eddie  wheel   161 Aug 17 05:34 index.html
-rw-r--r--  1 eddie  wheel    27 Aug 17 05:34 welcome.html
-rw-r--r--  1 eddie  wheel     0 Aug 17 05:34 world.html
```

你可以發現，不只六師弟回來了，剛剛被刪掉的所有檔案也都回來了。

使用 SourceTree 的話，這些被刪除的檔案前面會被標記一個減號。可以選取一個或多個檔案後，按滑鼠右鍵選擇「Reset...」功能：



然後會跳出一個確認對話框，問你是否確認：



按下 OK 鈕之後，剛剛刪除的那些檔案就救回來了。

這個技巧不只用在刪除的檔案可以救回來，當你修改某個檔案但後悔了，也可以用這個指令把檔案回復成它上一次 Commit 的狀態。

再怎麼樣都救得回來嗎？

也不能這麼說，因為整個 Git 的紀錄都是放在根目錄的 `.git` 目錄裡，所以如果這個目錄被刪了，也就是表示歷史紀錄也被刪了，那就沒得救了。

【冷知識】Git 是去哪裡把檔案救回來的？

其實是使用《火影忍者》忍術「穢土轉生」把檔案救回來的！

開玩笑的，當然不是用忍術。這裡用的 `git checkout` 指令，在後面提到關於分支的時候也會再度出現。當使用 `git checkout 分支名稱` 的時候，Git 會切換到指定的分支，但如果後面接的是檔名或路徑，Git 則不會切換分支，而是把檔案從 `.git` 目錄裡拉一份到目前的工作目錄。

更精準的來說，這個指令會把暫存區（Staging Area）裡的內容或檔案，拿來覆蓋工作目錄（Working Directory）的內容或檔案。所以當在上面執行 `git checkout welcome.html` 或 `git checkout .` 的時候，它會把 `welcome.html` 這個檔案，或是當下目錄所有檔案回復到上一次 Commit 的狀態。

如果在執行這個指令的時候這樣做多加了一個參數：

```
$ git checkout HEAD~2 welcome.html
```

這樣就會拿距離現在兩個版本以前的那個 `welcome.html` 檔案來覆蓋現在的工作目錄裡的 `welcome.html` 檔案，但要注意的是，這同時也會更新暫存區的狀態喔。

如果能理解上面這個指令的意思：

```
$ git checkout HEAD~2 .
```

這個指令的意思就是「拿距離現在兩個版本以前的檔案來覆蓋現在工作目錄的檔案，同時也更新暫存區裡的狀態」。

【狀況題】剛才的 Commit 後悔了，想要拆掉重做...

這個狀況很常見，雖然使用的指令 `git reset` 看起來很簡單，但因不少人誤解 Reset 指令的意思，所以反而是很多學習 Git 的人容易卡關的地方。

退一步海闊天空

先看一下我們目前的 Git 紀錄：

```
$ git log --oneline
e12d8ef (HEAD -> master) add database.yml in config folder
85e7e30 add hello
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

如果想拆掉最後一次的 Commit，可以使用「相對」或「絕對」的做法。「相對」的做法可以這樣：

```
$ git reset e12d8ef^
```

在最後面的那個 `^` 符號，每一個 `^` 符號表示「前一次」的意思，所以 `e12d8ef^` 是指在 `e12d8ef` 這個 Commit 的「前一次」，如果是 `e12d8ef^^` 則是往前兩次，以此類推。不過如果要倒退五次，通常不會寫 `e12d8ef^^^^^`，而會寫成 `e12d8ef~5`。

而且因為剛好 `HEAD` 跟 `master` 目前都是指向 `e12d8ef` 這個 Commit，而且 `e12d8ef` 這個數字也不好記，所以上面這行，通常也會改寫成：

```
$ git reset master^
```

或

```
$ git reset HEAD^
```

以這個例子來說會得到一樣的結果。（分支以及什麼是 HEAD 將於後面的章節再做說明）

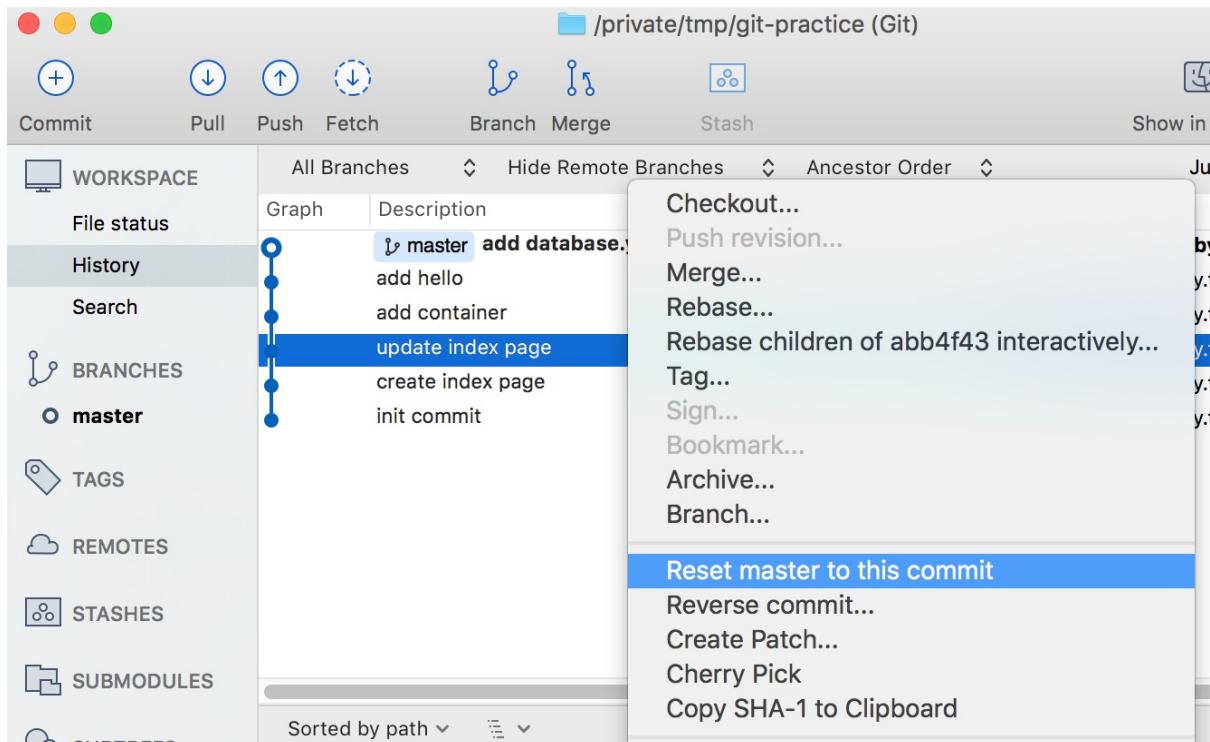
以上是「相對」的方式。如果你很清楚想要把目前的狀態退回到哪個 Commit，可以直接指明：

```
$ git reset 85e7e30
```

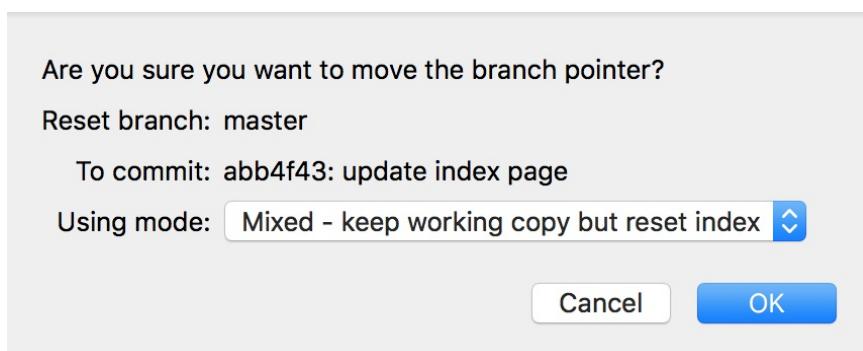
它就會切換到 `85e7e30` 這個 Commit 的狀態，因為 `85e7e30` 剛好就是 `e12d8ef` 的前一次 Commit，以這個例子來說也會達到跟「拆掉最後一次的 Commit」一樣的效果。

如何在 SourceTree ，Commit 後悔了，想要拆掉重做

如果使用 SourceTree，可以選擇你想要 reset 到的 Commit，例如想要一口氣退 3 個 Commit，就在想要去的 Commit 上按滑鼠右鍵，選擇「Reset master to this commit」：



接著會有個對話框要你選模式，這裡先選預設的「Mixed」：



按下 OK 鈕之後就一口氣退掉 3 個 Commit 了。

是說，Commit 拆掉了，那拆出來的那些檔案跑哪去了？這問題的答案跟接下來要介紹的 Reset 模式有關。

Reset 的模式

`git reset` 指令可以搭配參數使用，常見到的三種參數，分別是 `--mixed`、`--soft` 以及 `--hard`，不同的參數執行之後會有稍微不太一樣的結果。

mixed 模式

`--mixed` 是預設的參數，如果沒有特別加參數，`git reset` 指令將會使用 `--mixed` 模式。這個模式會把暫存區的檔案丟掉，但不會動到工作目錄的檔案，也就是說 Commit 拆出來的檔案會留在工作目錄，但不會留在暫存區。

soft 模式

這個模式下的 `reset`，工作目錄跟暫存區的檔案都不會被丟掉，所以看起來就只有 HEAD 的移動而已。也因此，Commit 拆出來的檔案會直接放在暫存區。

hard 模式

在這個模式下，不管是工作目錄以及暫存區的檔案都會丟掉。

畫個表格整理一下：

模式	mixed 模式	soft 模式	hard 模式
工作目錄	不變	不變	丟掉
暫存區	丟掉	不變	丟掉

如果上面的說明對你來說不容易想像到底發生什麼事，用白話一點的方式說明，你只要記這些不同的模式，將會決定「Commit 拆出來的那些檔案何去何從」，讓我再用另一個表格來解釋：

模式	mixed 模式（預設）	soft 模式	hard 模式
Commit 拆出來的檔案	丟回工作目錄	丟回暫存區	直接丟掉

但是 Commit 拆掉之後又後悔了，可以再撿回來嗎？

當然可以，甚至是使用 `--hard` 模式拆掉的也都能撿回來，我們在下一個章節「[【狀況題】不小心使用 hard 模式 Reset 了某個 Commit，救得回來嗎？](#)」就會介紹如何撗回來。

【觀念】不要被 Reset 這個字給誤導了！

Reset 這個英文單字的翻譯是「重新設定」，但事實上 Git 的 Reset 指令用中文來說比較像是「前往」或「變成」，也就是「go to」或「become」的概念。當執行這個指令的時候：

```
$ git reset HEAD~2
```

這個指令你原本可能會解讀成「請幫我拆掉最後兩次的 Commit」，但其實用「拆」這個動詞只是我們比較容易理解而已，事實上並沒有真的把這個 Commit 「拆掉」（放心，所有的 Commit 都還在）。

正確的說，上面這個指令應該要解讀成「我要前往兩個 Commit 之前的狀態」或是「我要變成兩個 Commit 之前的狀態」，而隨著使用不同的參數模式，原本的這些檔案就會丟去不同的區域。

因為實際上 `git reset` 指令也並不是真的刪除或是重新設定 Commit，只是「前往」到指定的 Commit，那些看起來好像不見的東西只是暫時看不到，但隨時都可以再撿回來。

Reset 是 Git 裡很常用的指令，所以一定要建立正確的觀念，操作 Git 才能真的達到隨心所欲的境界喔。

【狀況題】不小心使用 hard 模式 Reset 了某個 Commit，救得回來嗎？

借用一下上個章節的例子：

```
$ git log --oneline
e12d8ef (HEAD -> master) add database.yml in config folder
85e7e30 add hello
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

上面共計有六次 Commit。首先要先建立一個觀念，不管是用什麼模式進行 Reset，Commit 就是 Commit，並不會因為你 Reset 它然後就消失了。假設我們先用預設模式的 reset 指令倒退 2 步：

```
$ git reset HEAD~2
```

這時候 Commit 看起來就會少兩個，同時拆出來的檔案會被放置在工作目錄：

```
$ git log --oneline
657fce7 (HEAD -> master) add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

如果想要退回剛剛 Reset 的這個步驟，只要 Reset 回到一開始那個 Commit 的 SHA-1 `e12d8ef` 就行了：

```
$ git reset e12d8ef --hard
```

剛剛看起來拆掉的 Commit 就又接回來了。在這裡使用 `--hard` 參數可以強迫放棄 Reset 之後修改的檔案。

使用 Reflog

如果一開始沒有記下來 Commit 的 SHA-1 值也沒關係，Git 裡有個 `reflog` 指令有保留一些紀錄。再次借用上個章節的例子，但這次我改用 `--hard` 模式來進行 reset：

```
$ git reset HEAD~2 --hard  
HEAD is now at 657fce7 add container
```

不僅 Commit 看起來不見了，檔案也消失了。接著可使用 reflog 指令來看一下紀錄：

```
$ git reflog  
657fce7 (HEAD -> master) HEAD@{0}: reset: moving to HEAD~2  
e12d8ef (origin/master, origin/HEAD, cat) HEAD@{1}: checkout: moving from cat to master  
e12d8ef (origin/master, origin/HEAD, cat) HEAD@{2}: checkout: moving from master to cat
```

當 `HEAD` 有移動的時候（例如切換分支或是 `reset` 都會造成 `HEAD` 移動），Git 就會在 Reflog 裡記上一筆。從上面的這三筆記錄看起來大概可以猜得出最近三次 `HEAD` 的移動，而最後一次的動作就是 `Reset`。所以如果想要取消這次的 `Reset`，就是「`Reset` 到它 `Reset` 前的那個 Commit」（很像饒口令）。在這個例子就是 `e12d8ef`，所以只要這樣：

```
$ git reset e12d8ef --hard
```

就可以把剛剛 `hard reset` 的東西再次撿回來了。

小提示

`git log` 如果加上 `-g` 參數，也可以看到 Reflog 喔！

【冷知識】HEAD 是什麼東西？

HEAD 是一個指標，指向某一個分支，通常你可以把 HEAD 當做「目前所在分支」看待。在 .git 目錄裡有一個檔名為 HEAD 的檔案，就是記錄著 HEAD 的內容，來看一下這東西長什麼樣子：

```
$ cat .git/HEAD  
ref: refs/heads/master
```

從這個檔案看起來，HEAD 目前正指向著 master 分支。如果有興趣再深入看一下 refs/heads/master 的內容就會發現，其實所謂的 Master 分支也不過就是一個 40 個字元的檔案罷了：

```
$ cat .git/refs/heads/master  
e12d8ef0e8b9deae8bf115c5ce51dbc2e09c8904
```

切換分支的時候...

假設目前我這個專案包括 master 共有三個分支，而目前正在 master 分支上：

```
$ git branch  
cat  
dog  
* master
```

接下來我們試著切換分支到 cat 分支：

```
$ git checkout cat  
Switched to branch 'cat'
```

這時候來看一下剛剛那個 HEAD 檔案的內容：

```
$ cat .git/HEAD  
ref: refs/heads/cat
```

HEAD 的內容變成 refs/heads/cat 了。再試著切到 dog 分支看看：

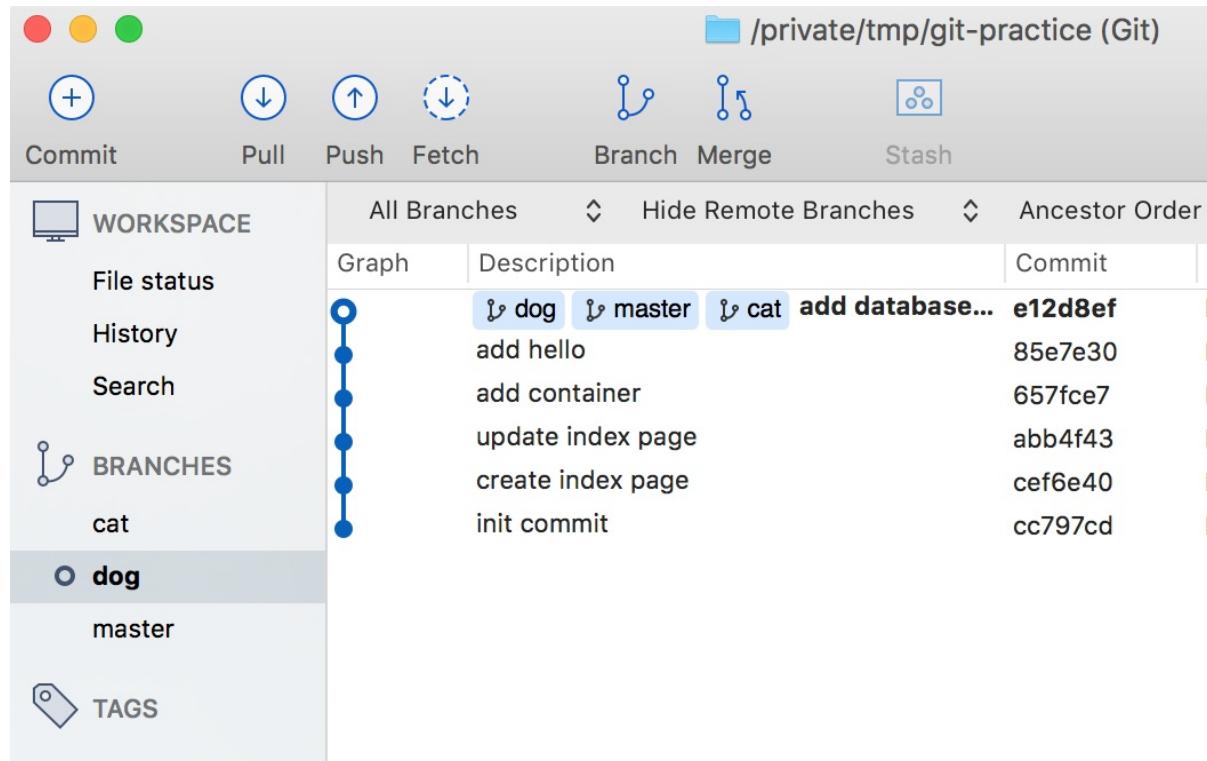
```
$ git checkout dog  
Switched to branch 'dog'
```

再確認一下 HEAD 的內容：

```
$ cat .git/HEAD  
ref: refs/heads/dog
```

它又改成指到 dog 分支了。也就是說，HEAD 通常會指向目前所在的分支。不過 HEAD 也不一定總是會指向某個分支，當 HEAD 沒有指向某個分支的時候便會造成「detached HEAD」的狀態，詳情請參閱「[【冷知識】斷頭\(detached HEAD\) 是怎麼一回事？](#)」章節說明。

而在 SourceTree 的介面裡，HEAD 是以一個「空心的小圈圈」圖示呈現：



在切換分支的同時，除了 HEAD 的內容會改變之外，在上個章節提到的 Reflog，當 HEAD 的內容改變的時候也會留下紀錄。

【狀況題】可以只 Commit 一個檔案的部份的內容嗎？

舉個例子來說，假設我的網站的首頁 index.html 的內容是這樣：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>首頁</title>
  </head>
  <body>
    <div class="container">
      <h1 id="heading">頭版消息</h1>
      <div>
        內文
        內文
        內文
        內文
      </div>
      <div id="footer">
        版權沒有，歡迎取用
      </div>
    </div>
  </body>
</html>
```

如果我因為某些龜毛的原因，footer 那個區塊還不想 Commit，在 Git 也可以先 Commit 其它的部份。

```
$ git add -p index.html
diff --git a/index.html b/index.html
index e90bdb3..2cac685 100644
--- a/index.html
+++ b/index.html
@@ -6,6 +6,16 @@
  </head>
  <body>
    <div class="container">
+      <h1 id="heading">頭版消息</h1>
+      <div>
+        內文
+        內文
+        內文
+        內文
+      </div>
+      <div id="footer">
+        版權沒有，歡迎取用
+      </div>
```

```
</div>
</body>
</html>
Stage this hunk [y,n,q,a,d/,e,?]?
```

當使用 `git add` 的時候加上 `-p` 參數後，Git 會問說是不是要把這個區塊（hunk）加到暫存區，如果選 `y` 就是整個檔案加進去。但因為我只想送出部份內容，所以選擇了 `e` 選項，接著就會出現編輯器顯示以下內容：

```
# Manual hunk edit mode -- see bottom for a quick guide.
@@ -6,6 +6,16 @@
</head>
<body>
  <div class="container">
+   <h1 id="heading">頭版消息</h1>
+   <div>
+     內文
+     內文
+     內文
+     內文
+   </div>
+   <div id="footer">
+     版權沒有，歡迎取用
+   </div>
  </div>
</body>
</html>
#
# To remove '-' lines, make them '' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging.
# If it does not apply cleanly, you will be given an opportunity to
# edit again. If all lines of the hunk are removed, then the edit is
# aborted and the hunk is left unchanged.
```

在這邊就可以編輯想要加到暫存區的區塊，因為 footer 的那個區塊我還不想加進去，所以我就把那三行刪掉，存檔並離開便可「把部份內容加到暫存區」。看一下目前的狀態：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

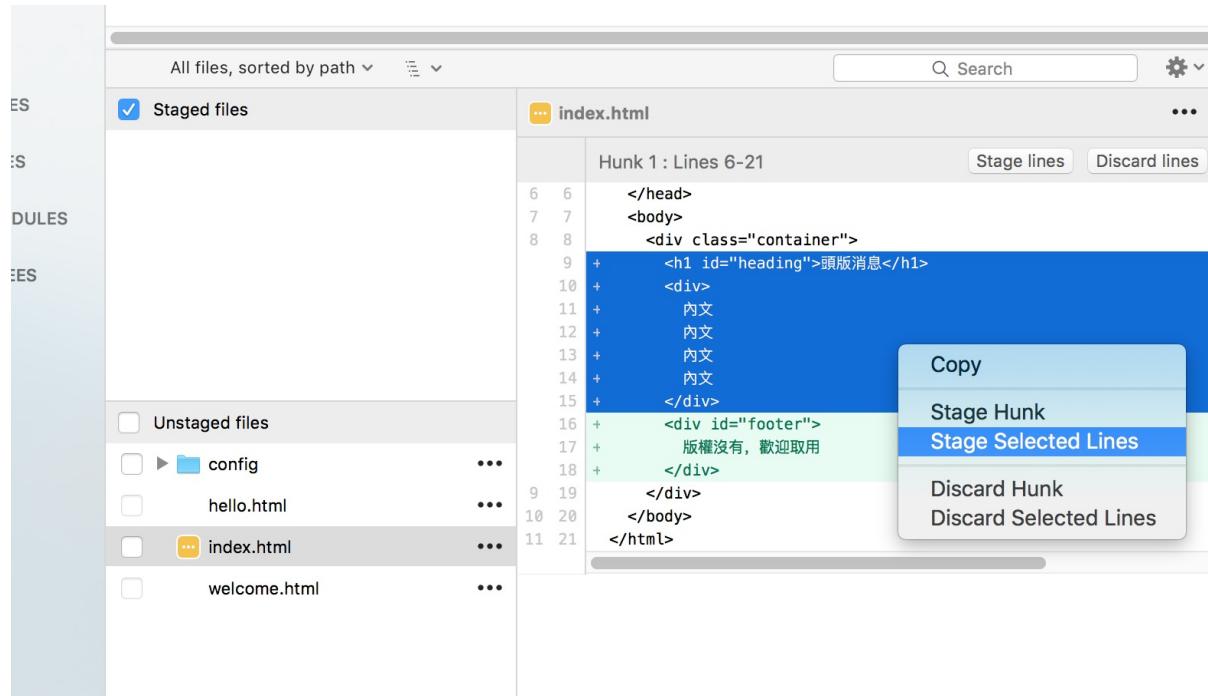
    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   index.html
```

`index.html` 這個檔案就有部份在暫存區，同時也有一份有包括那三行的版本放在工作目錄。這樣一來就可以只先 Commit 部份的內容了。

在 SourceTree 做這件事相對的比較簡單一些，用滑鼠選取想要加入的內容，按滑鼠右鍵選擇「Stage Selected Lines」，然後就搞定了。



【冷知識】那個長得很像亂碼 SHA-1 是怎麼算出來的？

SHA-1 是「Secure Hash Algorithm 1」的縮寫，它是一種雜湊演算法，計算之後的結果通常會以 40 個十六進位的數字方式呈現。這個演算法的特點之一，就是只要輸入一樣的值，就會有一樣的輸出值，反之，如果是不同的輸入值，就會有不同的輸出值。Git 裡所有物件的「編號」的計算主要都是靠這個演算法產生的。

SHA-1 值會有「重複」的狀況發生嗎？

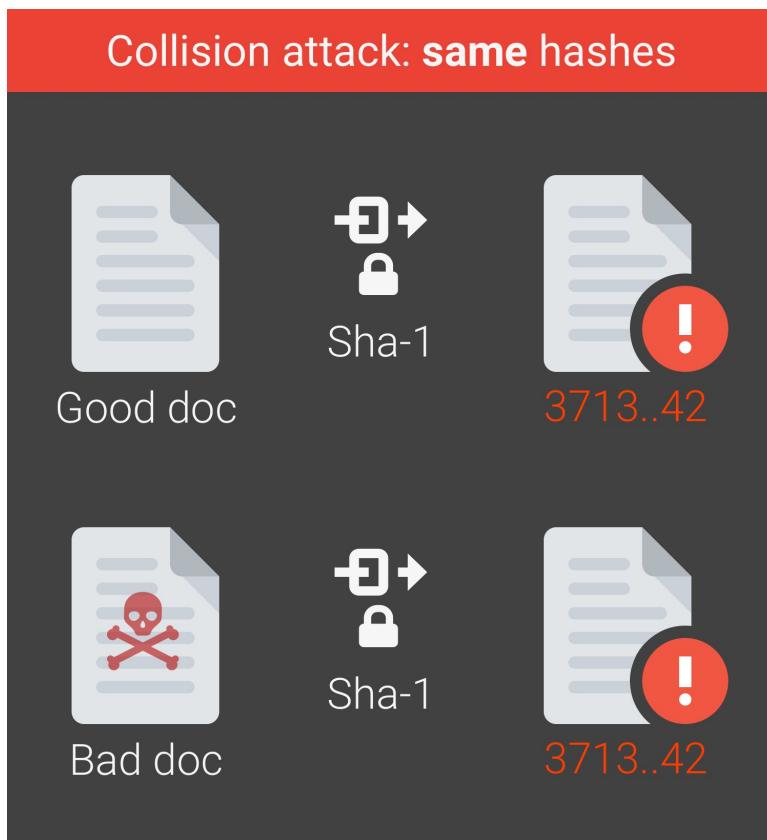
首先，前面提到 SHA-1 演算法的特性之一，就是相同的輸入值就會得到相同的結果，所以當遇到 SHA-1 值「重複」的時候不表示就是「重複」，通常是表示有相同的輸入值而已。

而當輸入兩個不同的值，卻得到相同的結果，也就是說，兩個內容不同的檔案，卻得到一樣的 SHA-1 值，這種情況稱之為碰撞（collision）。

老實說，這不是不可能發生，而是發生的機率大概是連續中 N 次的樂透彩吧。這發生機會有多低？我數學不好沒辦法給一個精準的數字，只能說發生碰撞的機會，就算每秒 Commit 幾十萬上下，在短短有限的人生大概也沒機會遇到碰撞。

等等... 聽說 Google 不是破解 SHA-1 什麼的嗎？

是的，Google 的確在今年（2017 年）年初就公佈了破解 SHA-1 的消息：



圖片來源：<https://shattered.io/> 網站

根據這個網站的說明，可以用二個不同的 PDF 檔（注意，是 PDF 檔而不是文字檔喔）的內容丟給 SHA-1 演算法會得到一樣的結果。

參考網址：<https://shattered.io/>

不過這其實有一點作弊，因為這裡改的是 PDF 檔而不是文字檔，也就是說的確可以硬改出兩個這樣的 PDF 沒錯，但這兩個 PDF 能不能正常被閱讀還不知道，所以如果是要改出兩個內容不同但 SHA-1 值要一樣的文字檔的難度將會更高。而且，這件事如果不是擁有地表最強運算資源的 Google，大概也做不到這件事。

除此之外，對 Git 來說，它也不是單純只用檔案的內容在計算，它還額外加了一些料，讓碰撞機會更低。

計算公式

在 Git 裡，不同種類的物件的 SHA-1 值計算會稍微有些不同，舉例來說，Blob 物件的 SHA-1 計算公式是：

1. "blob" 字樣
2. 一個空白字元
3. 輸入內容的長度

4. Null 結束符號

5. 輸入內容

如果是 Tree 物件，第 1 項則改成 "tree"；如果是 Commit 物件，第 1 項則改成 "commit"，以此類推。

同時，從上列公式可以看得出來，第 1 項到第 5 項都沒有跟時間或亂數有關的內容，只跟要計算的內容有關（不過 Commit 物件以及 Tag 物件除外，因為這兩個物件的「內容」本身就有包括時間）。所以，以 Blob 物件來說，不管是在什麼時間或不同的電腦上，一樣的輸入值，就會有一樣的內容。以下我用一段簡單的 Ruby 程式範例，用來計算 Git 裡 Blob 物件的 SHA-1 值：

```
# 引入 SHA-1 計算函式庫
require "digest/sha1"

# 要計算的內容
content = "Hello, 5xRuby"

# 計算公式
input = "blob #{content.length}\0#{content}"

puts Digest::SHA1hexdigest(input)
# 得到 "4135fc4add3332e25ab3cd5acabe1bd9ea0450fb"
```

如果你用 Git 內建的 `hash-object` 指令幫你算，也可以得到一樣的結果喔：

```
$ printf "Hello, 5xRuby" | git hash-object --stdin
4135fc4add3332e25ab3cd5acabe1bd9ea0450fb
```

【超冷知識】在 .git 目錄裡有什麼東西？Part 1

「想要我的財寶嗎？想要的話可以全部給你，去找吧！我把所有的財寶都放在那裡了。」

《航海王》哥爾羅傑

對 Git 來說，那個 .git 目錄就差不多是這樣的概念，所有的紀錄、所有的備份，都放在那裡了。所以想要好好真正的學好 Git，建議花一些時間來了解這個目錄裡到底藏了什麼東西，會更能理解 Git 運作的原理，操作起來也會更得心應手。

這個章節以及下個章節，將介紹 Git 內部的運作原理，以及在 .git 目錄裡還有哪些奇妙的東西。在開始介紹 .git 目錄裡的東西之前，一定要先知道的就是 Git 裡的四種物件。在 Git 裡，有四個很重要的物件，分別是 Blob 物件、Tree 物件、Commit 物件以及 Tag 物件。

接下來，我將透過實際操作 Git 的指令，慢慢介紹這些物件的關連性。

建立第一個檔案並加入至 Git 控管

首先產生一個 index.html 檔案，內容是 "hello, 5xRuby" 字串：

```
$ echo "hello, 5xRuby" > index.html
```

在前面的章節介紹過，在這個當下，`index.html` 還不算被加到 Git 裡，它只是路過而已，所以目前的狀態是屬於 Untracked，就是還沒有被 Git 追蹤的狀態。接下來，使用這個很常用的指令 `git add`，把它加到 Git 的暫存區，或稱之索引：

```
$ git add index.html
```

這時候的狀態是：

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

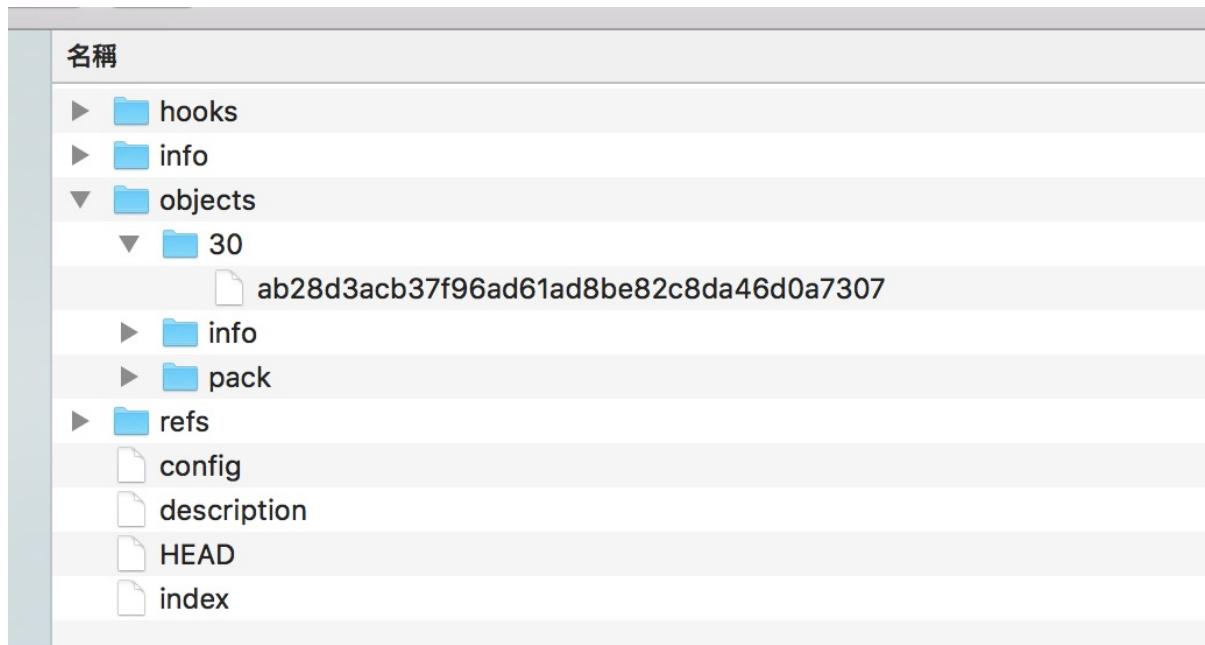
    new file:   index.html
```

`index.html` 已被加入暫存區無誤。當把檔案加入暫存區，Git 便會在 `.git` 目錄裡產生一個 Blob (Binary large object 的縮寫) 物件，並且依照它的「規則」擺放到它的目錄裡。這個 Blob 物件是用來存放 `index.html` 這個檔案的「內容」。注意，我這裡說的是「內容」而不是把整個 `index.html` 搬進 `.git` 目錄喔。

在「[【冷知識】那個長得很像亂碼 SHA-1 是怎麼算出來的？](#)」章節有介紹 Git 是怎麼計算 Blob 物件的 SHA-1 值，或是直接使用 `git hash-object` 指令來算也可以：

```
$ echo "hello, 5xRuby" | git hash-object --stdin  
30ab28d3acb37f96ad61ad8be82c8da46d0a7307
```

經過計算，“hello, 5xRuby”這串字的內容在 Git 得到的 SHA-1 值是 `30ab28d3acb37f96ad61ad8be82c8da46d0a7307`。接著 Git 就會在 `.git/objects` 目錄裡存放檔案。Git 會用這 40 字的 SHA-1 值的前兩個字做為目錄，剩餘的 38 字是檔案名字，像這樣：



那...那個檔案的內容是什麼？因為檔案的內容是已經過壓縮，所以如果用一般的文字編輯器是看不出來內容的，但可使用 `git cat-file` 指令來看看內容：

```
$ git cat-file -t 30ab28d3acb37f96ad61ad8be82c8da46d0a7307  
blob
```

其中那個 `-t` 參數表示要查看這個 SHA-1 值所代表的物件的型態。根據結果，Git 回報這個 SHA-1 值代表的是一種 Blob 物件。如果改使用 `-p` 參數：

```
$ git cat-file -p 30ab28d3acb37f96ad61ad8be82c8da46d0a7307  
hello, 5xRuby
```

則可以看到這個 SHA-1 值指向的那顆物件的內容，它顯示 `hello, 5xRuby`，確定就是我們剛剛的那個 `index.html` 沒錯。到這裡，我們可以得知：

1. 當使用 `git add` 把檔案加入至暫存區，Git 會根據這個物件的「內容」計算出 SHA-1 值。
2. Git 接著會用這個 SHA-1 值的前兩個字當做目錄名稱，後 38 個字當做檔案名稱，把目錄及檔案建立並存放在 `.git/objects` 目錄下。
3. 那個檔案的內容則是 Git 使用壓縮演算法，把原本的「內容」壓縮之後的結果。

規則：

Blob 物件的檔名是由 SHA-1 演算法決定，Blob 的內容則是壓縮演算法決定。

【冷知識】為什麼要拿前兩個字當目錄名稱？

在某些作業系統，一個目錄裡面如果放了非常多的檔案，該目錄的讀取效率會變得非常差，所以 Git 抽出了前兩位數做為目錄名稱，就是避免讓 `.git/objects` 目錄因為檔案過多而降低效能。

建立目錄

接著建立一個名為 `config` 的目錄。

```
$ mkdir config
```

這時候看一下狀態：

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   index.html
```

會發現 Git 完全沒感應該到剛剛建立的那個目錄！

為什麼？想一下剛剛的規則，Git 會對檔案的「內容」使用 SHA-1 演算法計算然後在 `.git/objects` 裡建立對應的目錄及檔案，但因為這只是一個空的 `config` 目錄，它根本就沒有「內容」可以計算，所以 Git 根本連感應都感應不到。請記得，Git 只對「內容」有興趣，這樣一個空的目錄對 Git 來說是無感的，所以它也只是路過，甚至連 Untracked 都算不上。即使手動自己下這個指令：

```
$ git add config
```

也是沒效果的！

重要！

空的目錄是無法被加到 Git 裡。

建立第二個檔案

在上一個步驟，既然 Git 沒辦法處理 `config` 這個空目錄，所以接下來我們在這個目錄裡建立一個檔案：

```
$ touch config/database.yml
```

看一下狀態：

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    config/
```

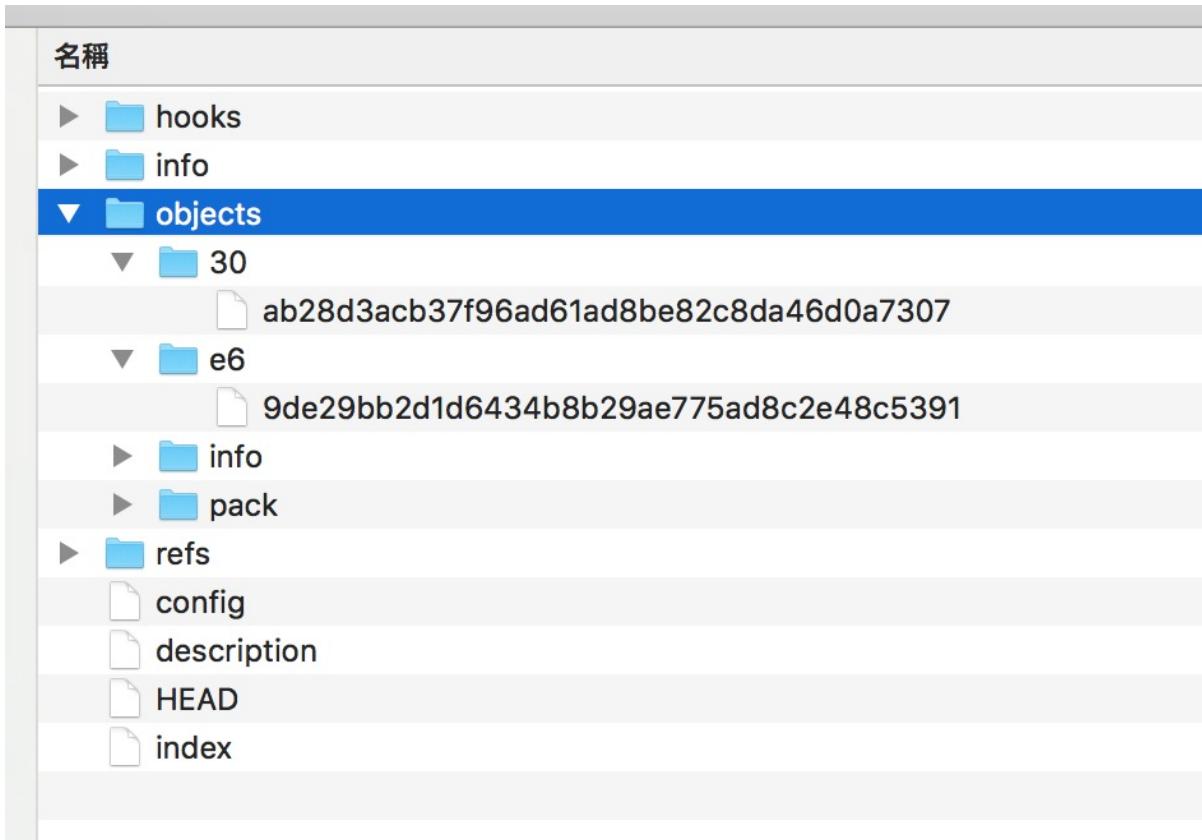
即使只是一個空的檔案，它也是有「內容」的，就是「空的內容」，原本無法被 Git 看見的 `config` 目錄，在裡面放了一個檔案之後就看見它了（有種「母因子而貴」的哀傷感？）。接著把它加到 Git 裡吧：

```
$ git add config/database.yml
```

根據 Git 的 Blob 物件計算方法：

```
$ cat config/database.yml | git hash-object --stdin
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

根據前面的規則，這時候應該會在 `.git/objects` 目錄裡建立一個 `e6` 的目錄，裡面有個 `9de29bb2d1d6434b8b29ae775ad8c2e48c5391` 這個檔案：



果然沒錯！到這裡，我想各位大概對 Git 世界裡的 Blob 物件稍微比較有概念了。

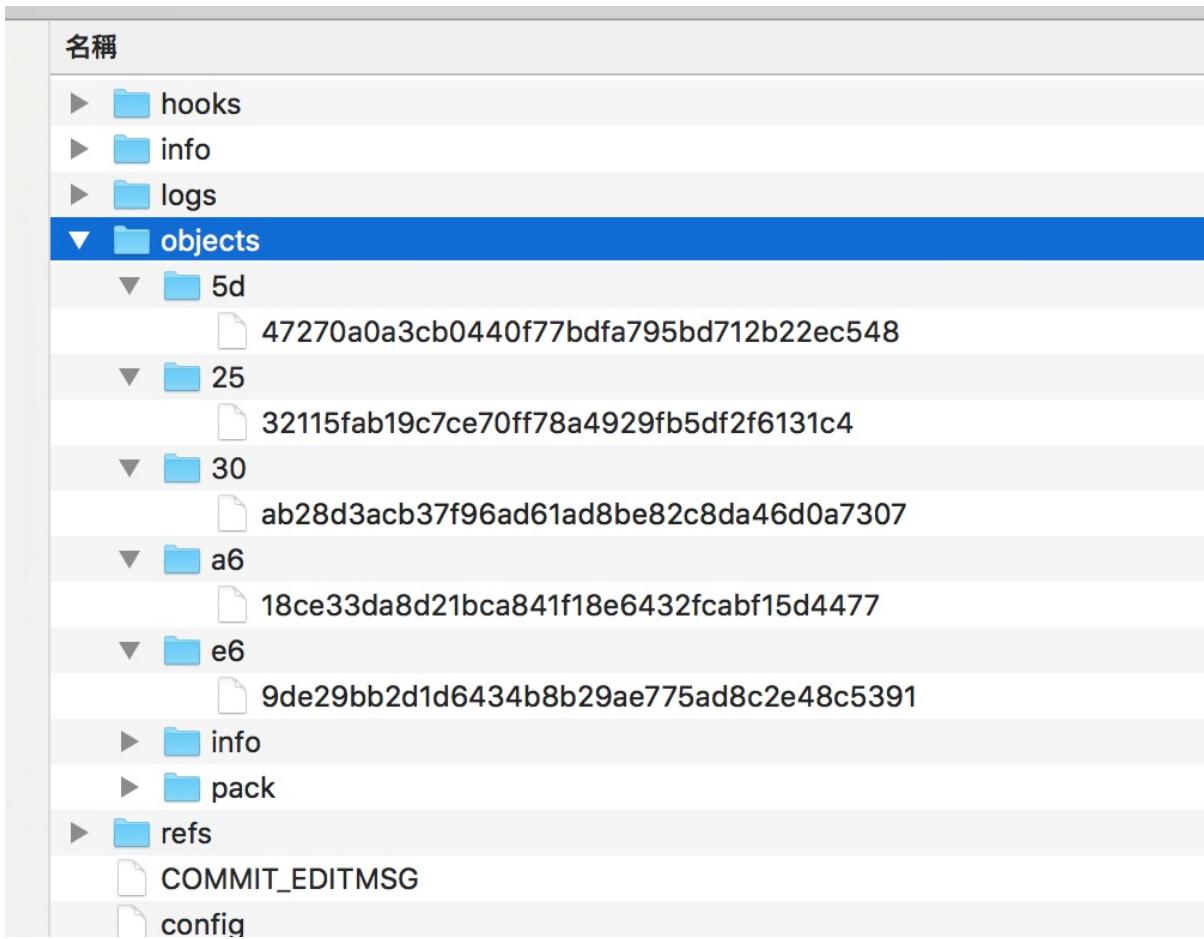
接下來你可能會好奇：「Git 只在意檔案的『內容』，難道目錄完全不重要嗎？檔案叫什麼名字也不重要嗎？」。其實也不是不重要，只是他們不屬於 Blob 物件的範圍，這是待會要介紹的 Tree 物件要處理的。

進行 Commit

既然檔案已加至暫存區，接下來就是要進行 Commit，觀察看看 `.git/objects` 目錄會有什麼變化：

```
$ git commit -m "init commit"
[master (root-commit) 5d47270] init commit
 2 files changed, 1 insertion(+)
 create mode 100644 config/database.yml
 create mode 100644 index.html
```

看看 `.git/objects` 目錄：



除了原本的 `30` 跟 `e6` 目錄之外又長出了好幾個目錄了，我們一個一個來看看這些是什麼東西。從 `25` 目錄裡的 `32115fab19c7ce70ff78a4929fb5df2f6131c4` 開始看：

```
$ git cat-file -t 2532115fab19c7ce70ff78a4929fb5df2f6131c4
tree
```

Git 告訴我們這個是一種 Tree 物件。看看這個物件裡是什麼東西：

```
$ git cat-file -p 2532115fab19c7ce70ff78a4929fb5df2f6131c4
040000 tree a618ce33da8d21bca841f18e6432fcabf15d4477    config
100644 blob 30ab28d3acb37f96ad61ad8be82c8da46d0a7307    index.html
```

嗯...這裡看到一個 Tree 物件以及一個 Blob 物件，而這個 Blob 物件，正是 `index.html`。再往下看看它指的另一個 Tree 物件：

```
$ git cat-file -t a618ce33da8d21bca841f18e6432fcabf15d4477
tree
```

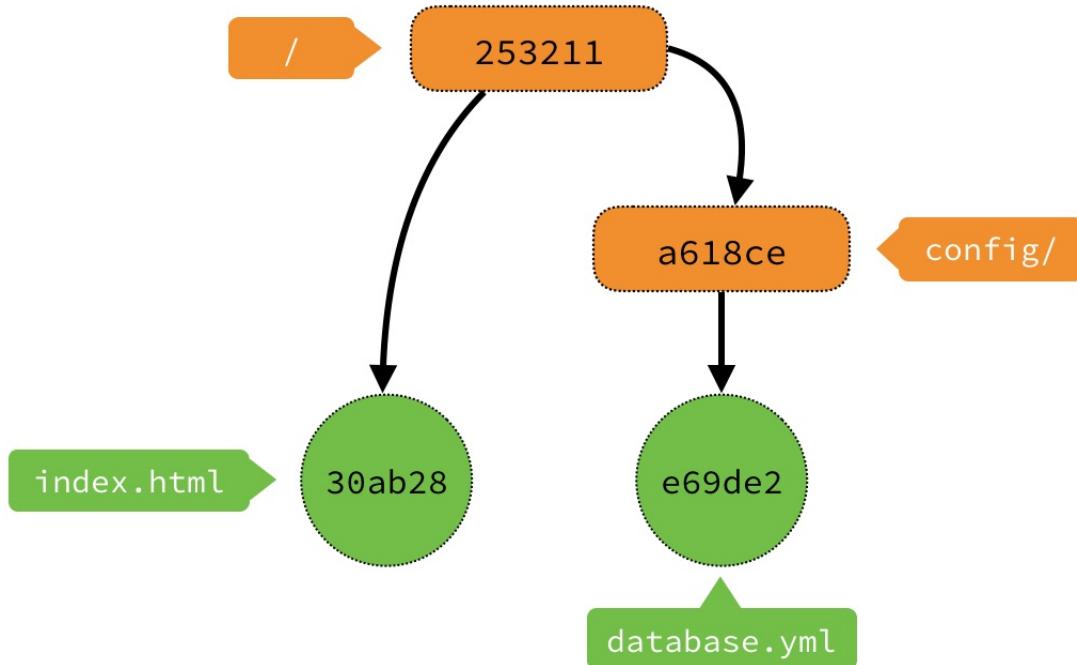
是個 Tree 物件沒錯，內容是：

```
$ git cat-file -p a618ce33da8d21bca841f18e6432fcabf15d4477
```

```
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 database.yml
```

在這個 Tree 物件裡看到一個 Blob 物件，正是剛剛放在 config 目錄裡的 database.yml 。

到這裡，我想大家可能稍微對 Tree 物件有點概念了，讓我用一張圖簡單的畫一下：



說明：

1. 檔案在 Git 裡會以 Blob 物件方式存放，例如在我們這個範例裡的 index.html 跟 database.yml 。
2. 目錄以及檔案的檔名，將會以 Tree 物件的方式存放，例如 253211 代表的是根目錄，而 a618ce 代表的是 config 目錄。
3. Tree 物件的內容會指向某個或某些 Blob 物件，或是其它的 Tree 物件。

你可能會覺得這好像有點像目錄跟子目錄的概念，但其實並不是。這東西有個專有名詞叫做 [Directed Acyclic Graph \(DAG\)](#)，中文翻譯做「有向無環圖」，這些物件們之間只有指來指去的關係，並沒有階層或目錄、子目錄的關係，不管是 Blob 物件或是 Tree 物件，大家都是平行的。上圖會用類似目錄的方式呈現，只是因為這樣比較容易想像、理解罷了，事實上他們並沒有階層關係喔。

介紹完 Tree 物件，剛剛 Commit 完成後多長出來的 3 個物件還有一個沒看：

```
$ git cat-file -t 5d47270a0a3cb0440f77bd795bd712b22ec548
commit
```

這顆就是我們接下來要介紹的 Commit 物件。看看這個 Commit 物件的內容：

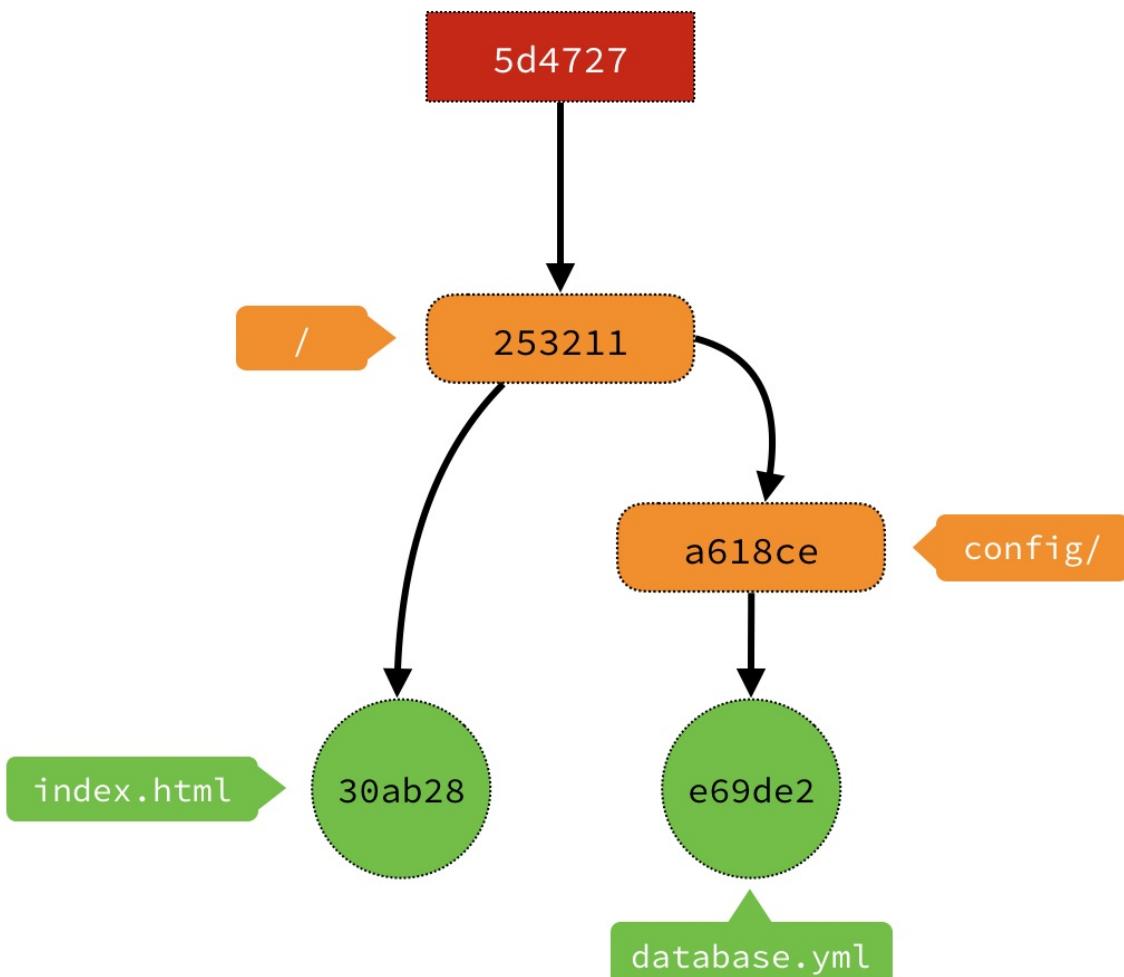
```
$ git cat-file -p 5d47270a0a3cb0440f77bdfa795bd712b22ec548
tree 2532115fab19c7ce70ff78a4929fb5df2f6131c4
author Eddie Kao <eddie@5xruby.tw> 1503442336 +0800
committer Eddie Kao <eddie@5xruby.tw> 1503442336 +0800

init commit
```

這個 Commit 物件包括以下資訊：

1. 某個 Tree 物件。還記得 253211 是誰嗎？如果往回翻就可以發現它就是代表根目錄的那顆 Tree 物件。
2. 本次 Commit 的時間。
3. 作者跟進行這次 Commit 的人。大部份時候會是同一個人，但偶爾也會有「我 Code 寫好了但我沒辦法 Commit 所以請人代發」的情況。
4. 本次 Commit 的訊息。

我把到目前提到的 Blob 物件、Tree 物件以及 Commit 物件的關連圖再重新整理一下：



提醒一下，在這個圖中，在各個物件旁邊寫著目錄或檔名的小提示牌，並不是「這個物件就是這個檔案或目錄」的意思，而是「這個物件的內容是來自這個檔案」的意思，放上提示牌只是為了讓大家更容易理解各物件之間的關係而已。

請參考這個圖，然後記得以下規則：

1. Commit 物件會指向某個 Tree 物件。
2. Tree 物件的內容會指向某個或某些 Blob 物件，或是其它的 Tree 物件。

Git 物件的四大天王已經出現三位了，還少一位，也就是 Tag 物件，不過它現在還沒登場，待稍後再另做介紹。

分支登場

在後面的「[對分支的誤解](#)」章節會有關於分支的介紹，在 Git 裡面的分支其實就跟一張貼紙一樣，它會貼在某個 Commit 上，並且會隨著每次的 Commit 不斷的移動；同時，在「[【冷知識】HEAD 是什麼東西？](#)」章節也有關於 HEAD 的介紹，它是一個會指向某個分支的指標，你可以把 HEAD 看成「目前所在的分支」。綜合以上，這時候應該會有一個分支（就是 Git 預設的 `master` 分支）以及 HEAD 會指向目前唯一的 Commit，就像這樣：



人生繼續往前進

做完了一次 Commit，我們來繼續再來做下一次的 Commit，看看 `.git` 目錄又會有什麼變化吧。這回，我們來編輯一下 `index.html` 這個檔案：

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello, 5xRuby</h1>
  </body>
</html>
  
```

幫它加上了一些 HTML 的內容。這時候的狀態是：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

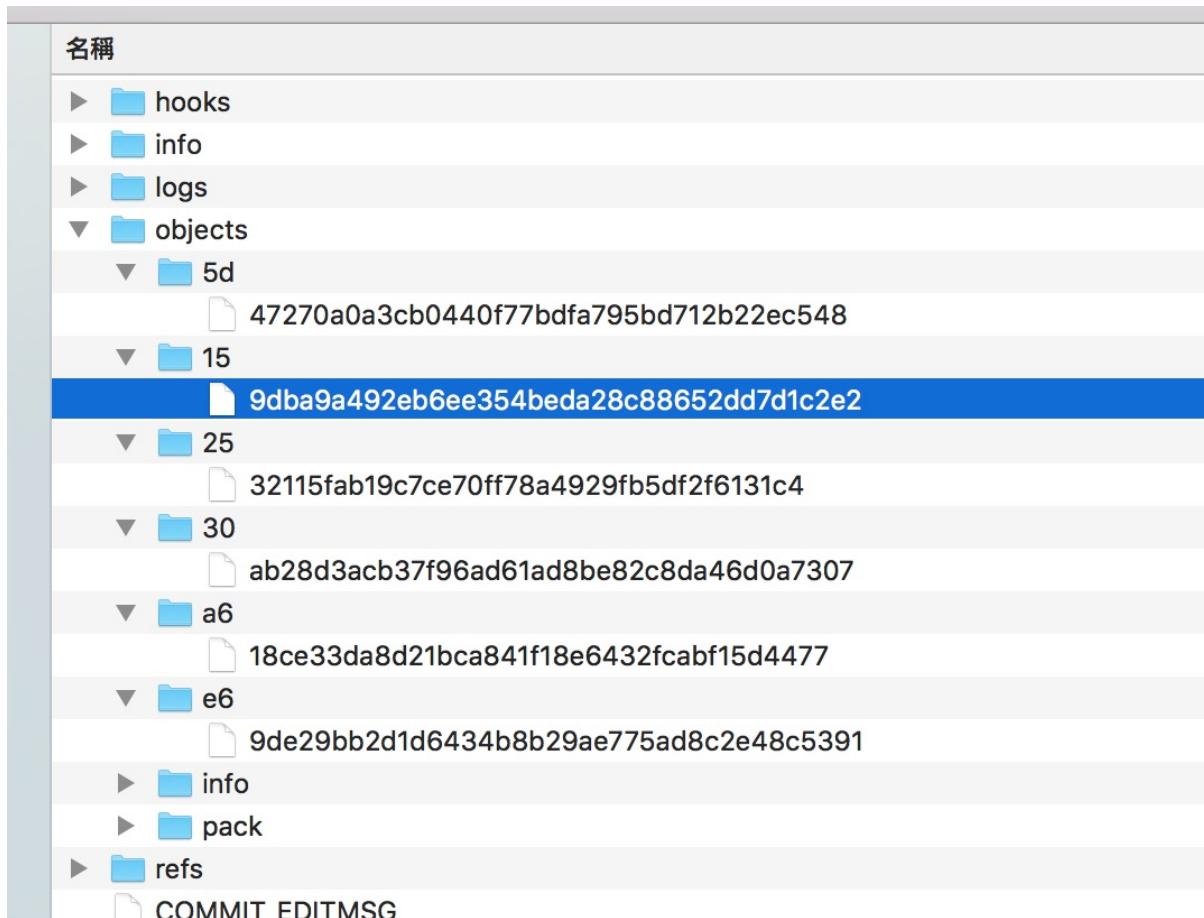
沒有很意外，這個檔案因為修改了，所以它就是處於 `modified` 沒錯。不過這時候還沒有新的物件產生，要把檔案加到暫存區才會長出新的 Blob 物件喔。

```
$ git add index.html
```

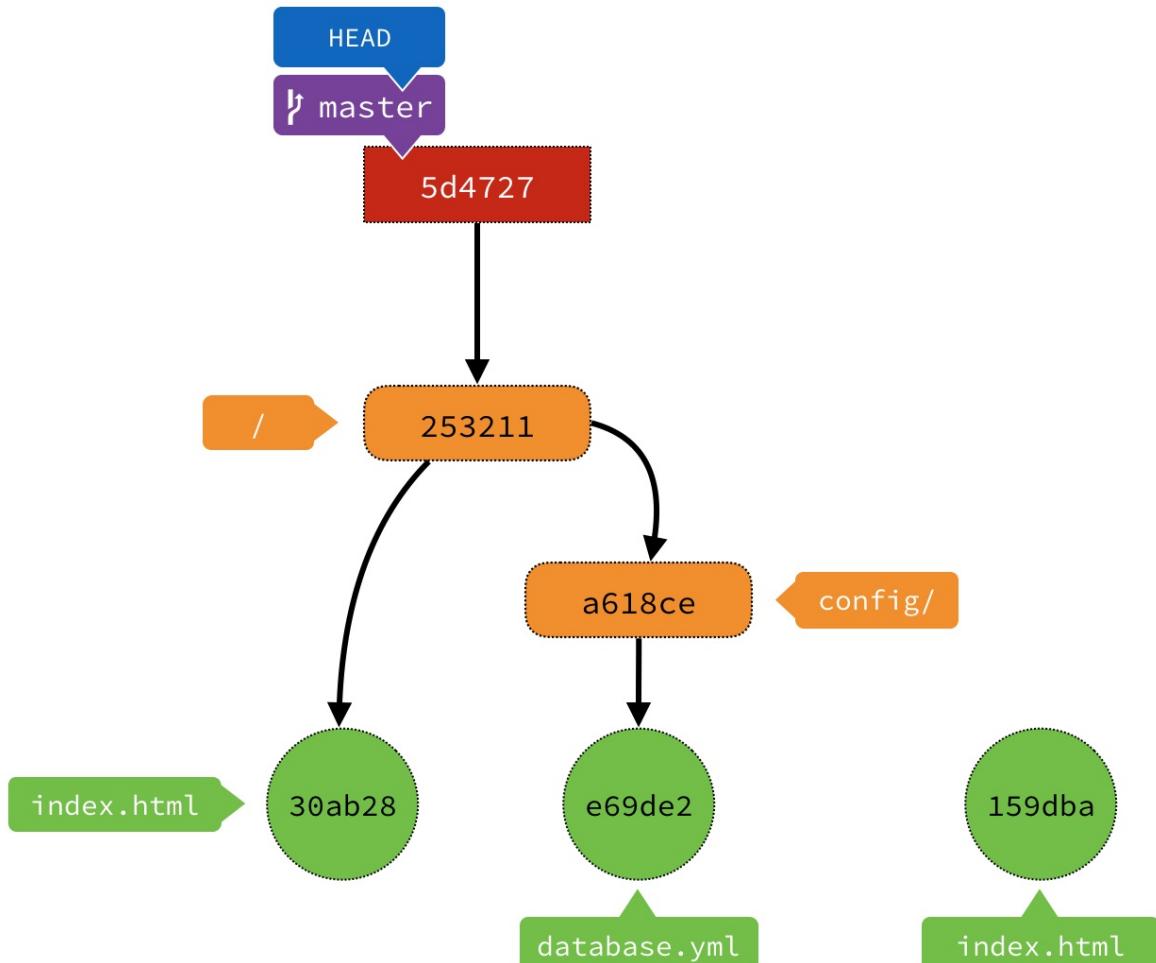
請 `git hash-object` 指令先幫我們算一下：

```
$ cat index.html | git hash-object --stdin
159dba9a492eb6ee354beda28c88652dd7d1c2e2
```

根據前面的規則，這時候 Git 應該會在 `.git/objects` 目錄下會多一個 `15` 目錄，並且在裡面有一個檔案叫做 `9dba9a492eb6ee354beda28c88652dd7d1c2e2`。看一下實際的情況：



是的，就是這樣！所以現在的物件們的樣子大概是這樣：



那個 Blob 物件 `159dba` 目前還沒人理它。不過沒關係，繼續 Commit 下去，看看會發生什麼事。

```
$ git commit -m "update index.html"
[master 4ddb0b5] update index.html
 1 file changed, 10 insertions(+), 1 deletion(-)
```

如果跟 Commit 之前的狀態比對的話，會發現這次的 Commit 多出了兩個目錄，他們的 SHA-1 值分別是 `3a648a` 跟 `4ddb0b`（這裡僅使用六碼短碼表示）。一樣使用 `git cat-file` 來看看內容：

```
$ git cat-file -t 3a648a98d322f82a72bff20ed977539c581a181d
tree

$ git cat-file -p 3a648a98d322f82a72bff20ed977539c581a181d
040000 tree a618ce33da8d21bca841f18e6432fcabf15d4477 config
100644 blob 159dba9a492eb6ee354beda28c88652dd7d1c2e2 index.html
```

嗯...這顆 Tree 物件的內容看起來跟上一個 Commit 裡代表根目錄的那個 Tree 物件有點像，只是內容稍微有點不太一樣。在這個 Tree 物件裡，原本指向 config 目錄的 Tree 物件因為沒有修改，所以它的指向沒有變；反倒是代表 index.html 的那個 Blob 物件因為內容修改，所以指向別顆新的 Blob 物件了。

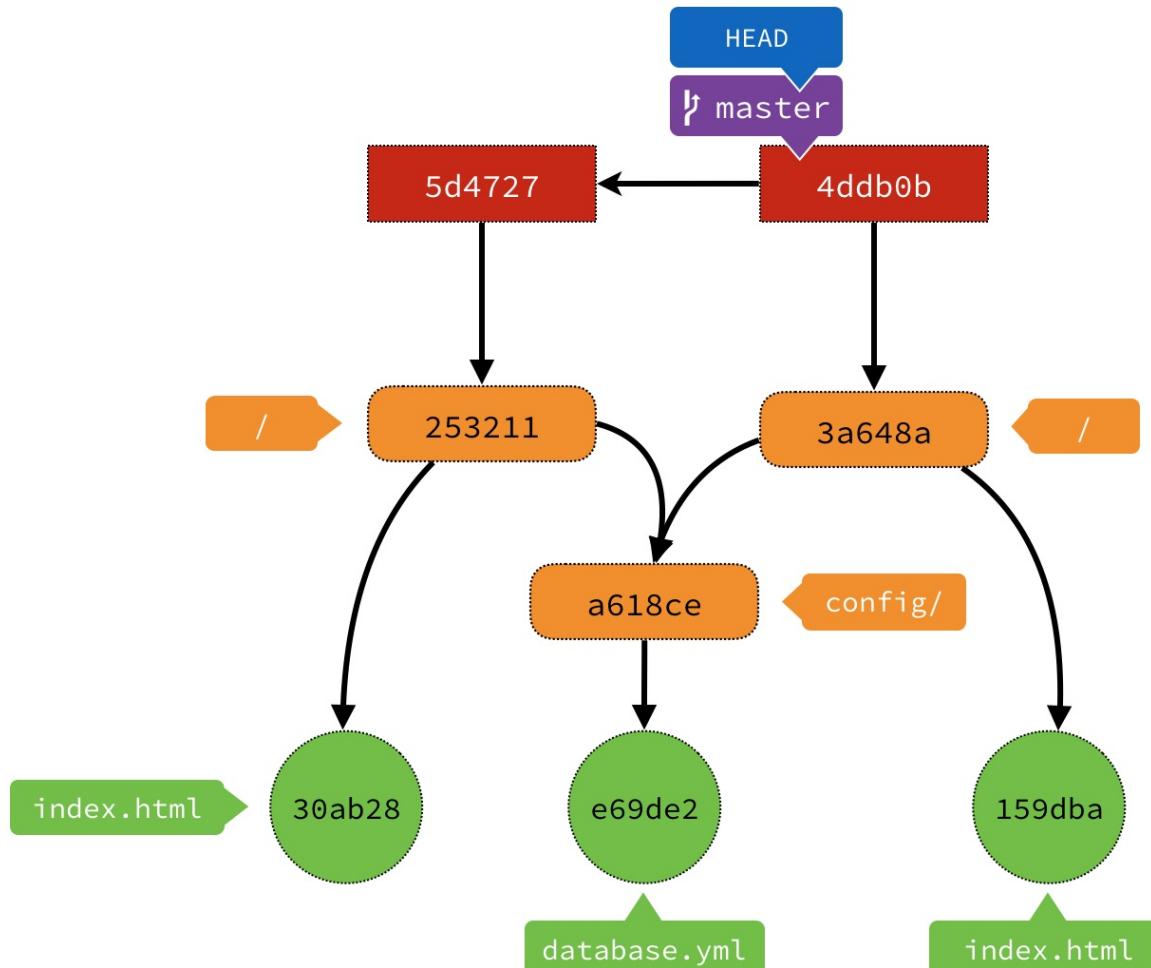
來看另一顆物件 4ddb0b：

```
$ git cat-file -t 4ddb0b5fce084d36248eb1ce14109162617abd51
commit

$ git cat-file -p 4ddb0b5fce084d36248eb1ce14109162617abd51
tree 3a648a98d322f82a72bff20ed977539c581a181d
parent 5d47270a0a3cb0440f77bd795bd712b22ec548
author Eddie Kao <eddie@5xruby.tw> 1503469899 +0800
committer Eddie Kao <eddie@5xruby.tw> 1503469899 +0800

update index.html
```

這個 4ddb0b 是一個 Commit 物件，從上面的訊息來看，也有指向一個 Tree 物件 3a648a，其它資訊都差不多，但這個 Commit 比前一個 Commit 物件多了一個 parent 的資訊，表示指向「上一次」的 Commit。現在目前各個物件的關連圖應該會像這樣：



分支跟 HEAD 也是跟著 Commit 的變化而調整位置。會覺得有點複雜了嗎？來幫剛才的兩條規則再加一條新的規則：

1. Commit 物件會指向某個 Tree 物件。
2. Tree 物件的內容會指向某個或某些 Blob 物件，或是其它的 Tree 物件。
3. 除了第一個 Commit 物件以外，所有 Commit 物件都會指向它的前一次的 Commit 物件。

接下來，再做最後一次的 Commit，為這個「認識 Git 四大天王物件」之旅做結尾。

繼續前進

這回我要來做一個有趣的試驗，我在這個專案的根目錄下面加一個 `README.md`，檔案內容是空的沒關係：

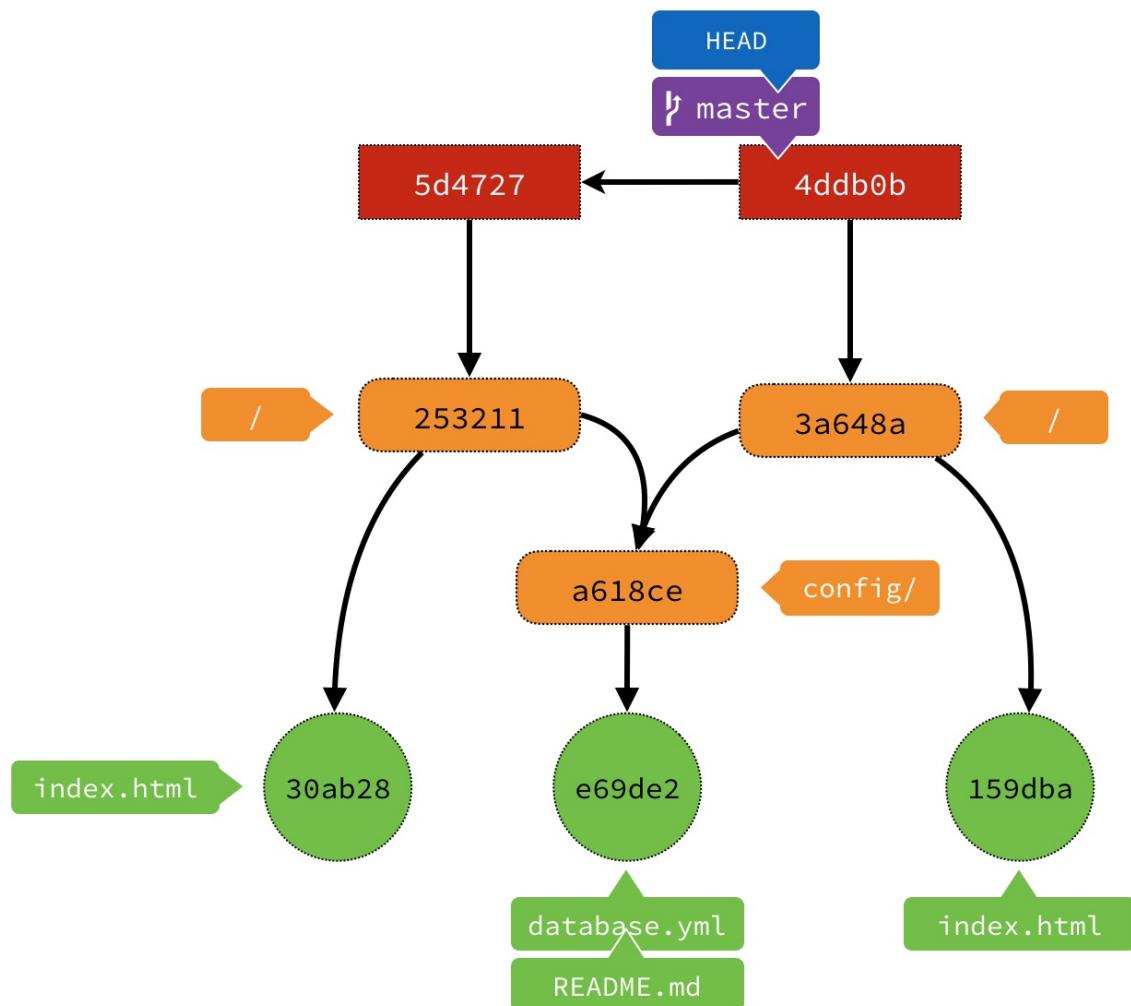
```
$ touch README.md
```

再來把這個檔案加進暫存區：

```
$ git add README.md
```

根據我們之前學到的，執行 `git add` 指令後應該會多一顆 Blob 物件，但如果去 `.git/objects` 目錄下看會發現並沒有變多，原因是因為「在這邊已經有一顆一樣的內容的 Blob 物件了」。

還記得嗎？一開始放在 `config/database.yml` 的這個檔案也是空的喔，在 Git 的世界裡，只要檔案的內容是一樣的，它們在 Git 的世界裡就會是同一顆 Blob 物件。這時候的物件關連圖會是這樣：



其實這跟上一張圖沒太大的差別。接著繼續 Commit 吧：

```
$ git commit -m "add README"
[master 4bf7d4a] add README
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
```

這次的 Commit，多了兩個新物件，分別是 `0fc2cc` 跟 `4bf7d4`。先看第一顆物件 `0fc2cc`：

```
$ git cat-file -t 0fc2cccfb7a20abde62d9cedd577300b8d1de1a7
tree

$ git cat-file -p 0fc2cccfb7a20abde62d9cedd577300b8d1de1a7
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391      README.md
040000 tree a618ce33da8d21bca841f18e6432fcabf15d4477    config
100644 blob 159dba9a492eb6ee354beda28c88652dd7d1c2e2    index.html
```

這是一顆 Tree 物件，拿來跟前一次 Commit 的 Tree 物件相比的話：

```
$ git cat-file -p 3a648a98d322f82a72bff20ed977539c581a181d
040000 tree a618ce33da8d21bca841f18e6432fcabf15d4477    config
100644 blob 159dba9a492eb6ee354beda28c88652dd7d1c2e2    index.html
```

看得出來代表 config 目錄的 Tree 物件以及代表 index.html 內容的 Blob 物件這兩行都沒有變，只是新增了一行代表 README.md 的 Blob 物件。雖然此次的 Commit 並沒有產生新的 Blob 物件，但 Tree 物件還是需要紀錄有 README.md 這個檔案的存在，所以 Tree 物件的「內容」改變了，因此 Git 會做出一顆新的 Tree 物件。

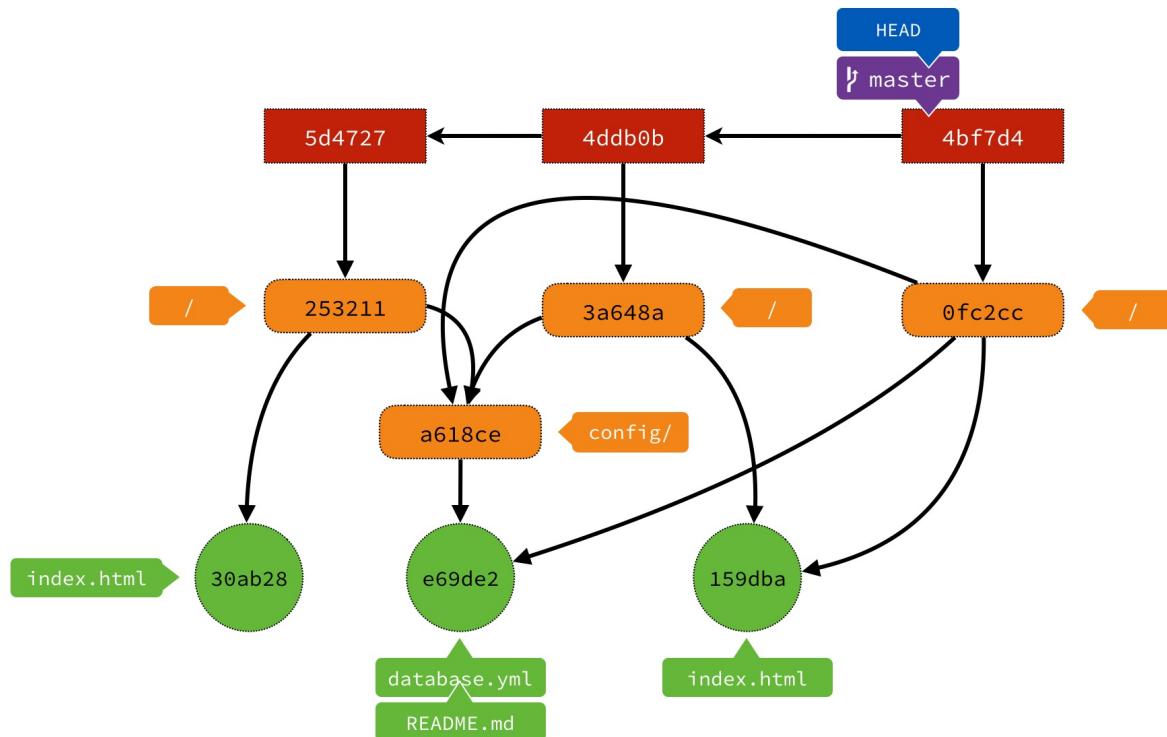
再來看看第二顆物件 4bf7d4：

```
$ git cat-file -t 4bf7d4adf6e56964ae3c0625bbc54275a02672d7
commit

$ git cat-file -p 4bf7d4adf6e56964ae3c0625bbc54275a02672d7
tree 0fc2cccfb7a20abde62d9cedd577300b8d1de1a7
parent 4ddb0b5fce084d36248eb1ce14109162617abd51
author Eddie Kao <eddie@5xruby.tw> 1503471804 +0800
committer Eddie Kao <eddie@5xruby.tw> 1503471804 +0800

add README
```

這個 4bf7d4 是一個 Commit 物件，不過內容就沒什麼意外了，這個 Commit 物件指向剛剛那顆 Tree 物件 0fc2cc，並且也指向前一次的 Commit 物件。到這裡，我們可以把這三次 Commit 裡所有物件的關連圖畫出來了：



這三次的 Commit，共總產生了 10 顆物件（分支不算物件），除了用肉眼算之外，也可以請 Git 幫你算：

```
$ git count-objects
10 objects, 40 kilobytes
```

總共 10 顆沒錯喔 :)

經過這三次的 Commit，我想各位對於 Blob、Tree 以及 Commit 物件應該有更清楚的認識了。這樣的土砲流程也許會有點悶，但透過這個過程，如果你能理解以及預測下一次的 Commit 大概會長出什麼物件，應該就會更清楚 Git 內部的運作原理。

等等.. 四大天王的最後一位還沒登場啊

最後一位，Tag 物件，它不會在 Commit 的過程中長出來，必須要手動的把 Tag 貼在某個 Commit 上，而且還不是一般的輕量 Tag，而是有附註的 Tag (Annotated Tag)，關於 Tag 的介紹請參閱「[使用標籤](#)」章節。

讓我們在目前這個 Commit 打上一個 Tag 吧：

```
$ git tag -a big_treasure -m "媽，我在這"
```

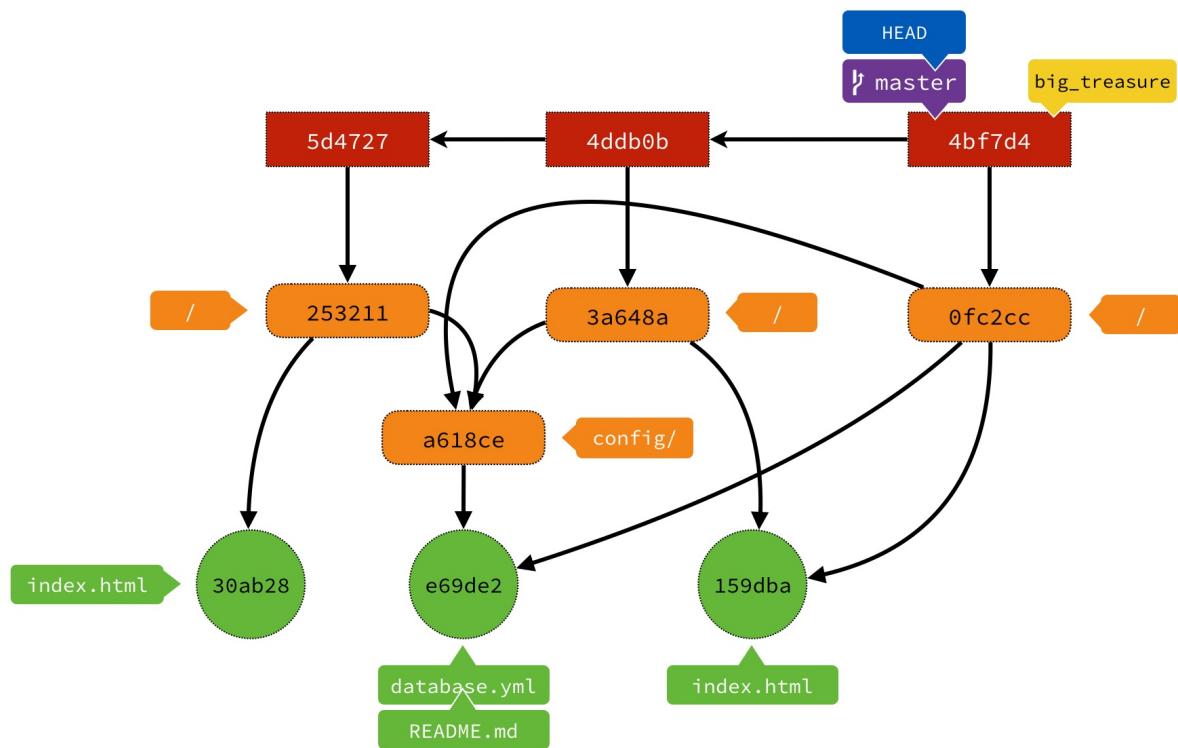
這時候，`.git/objects` 目錄下多長了一顆 `3b9eab` 物件，用一樣的方式來看看這傢伙：

```
$ git cat-file -t 3b9eab6d623496a272f61076a765301bb7af4367
tag

$ git cat-file -p 3b9eab6d623496a272f61076a765301bb7af4367
object 4bf7d4adf6e56964ae3c0625bbc54275a02672d7
type commit
tag big_treasure
tagger Eddie Kao <eddie@5xruby.tw> 1503508397 +0800

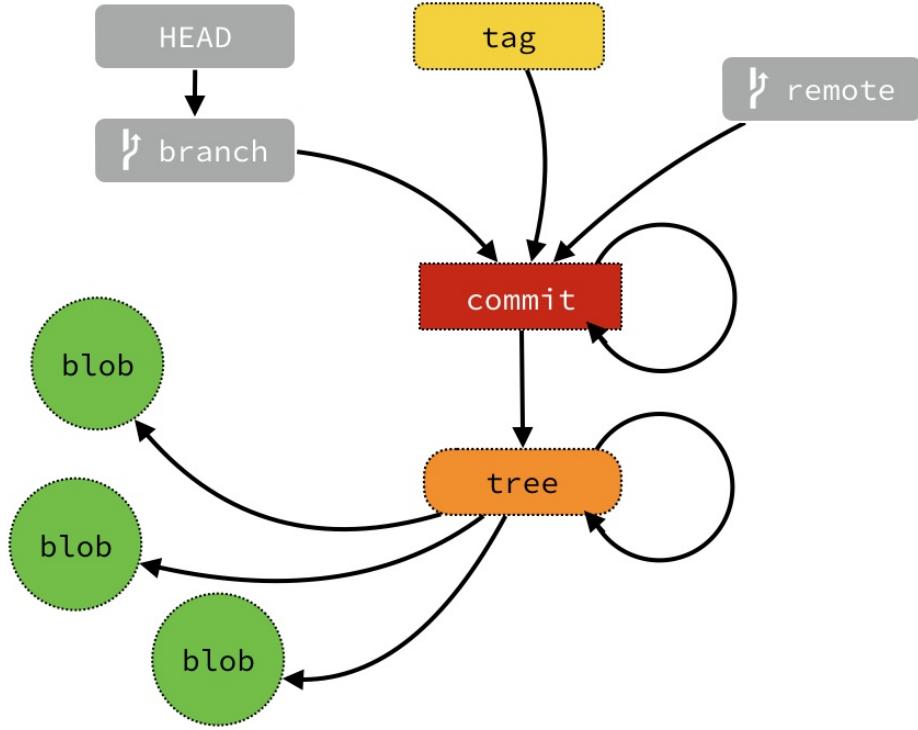
媽，我在這
```

嗯，這是一顆 Tag 物件，這顆 Tag 物件同樣也會標記是誰在什麼時候做了這顆 Tag，並且標在 4bf7d4 這個 Commit 物件上。現在完整的關連圖變這樣：



小結

Git 裡的四種物件的關係，可以用下面這張圖做個簡單的總結：



1. 把檔案加入 Git 之後，檔案的內容會被轉成 Blob 物件儲存。
2. 目錄以及檔名會存放在 Tree 物件內，Tree 物件會指向 Blob 物件，或是其它的 Tree 物件。
3. Commit 物件會指向某個 Tree 物件。除了第一個 Commit 之外，其它的 Commit 都會指向前一次的 Commit 物件。
4. Tag 物件（Annotated Tag）會指向某個 Commit 物件。
5. 分支雖然不屬於這四個物件之一，但它會指向某個 Commit 物件。
6. 當開始往 Git Server 上推送之後，在 `.git/refs` 底下就會多出一個 `remote` 的目錄，裡面放的是遠端的分支，基本上跟本地的分支是差不多的概念，同樣也會指向某個 Commit 物件。
7. HEAD 也不屬於這四個物件之一，它會指向某個分支。

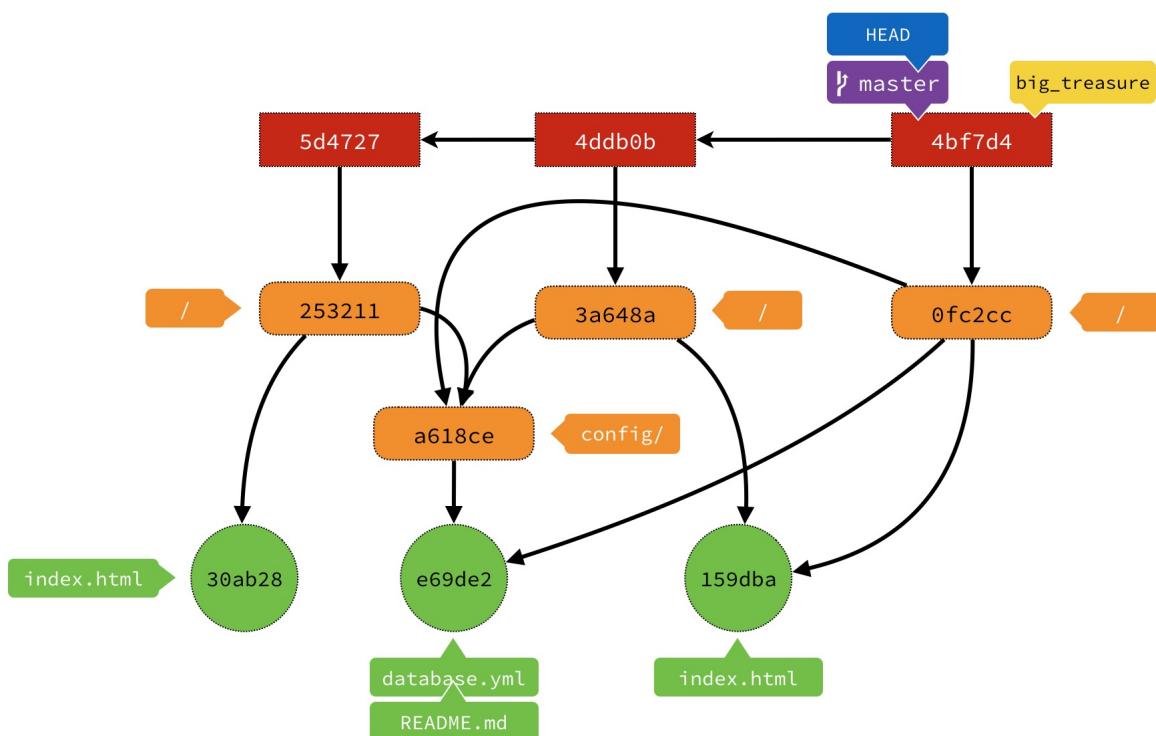
下個章節，我們將介紹當 Git 在進行 Checkout 的時候會發生的事。

【超冷知識】在 .git 目錄裡有什麼東西？Part 2

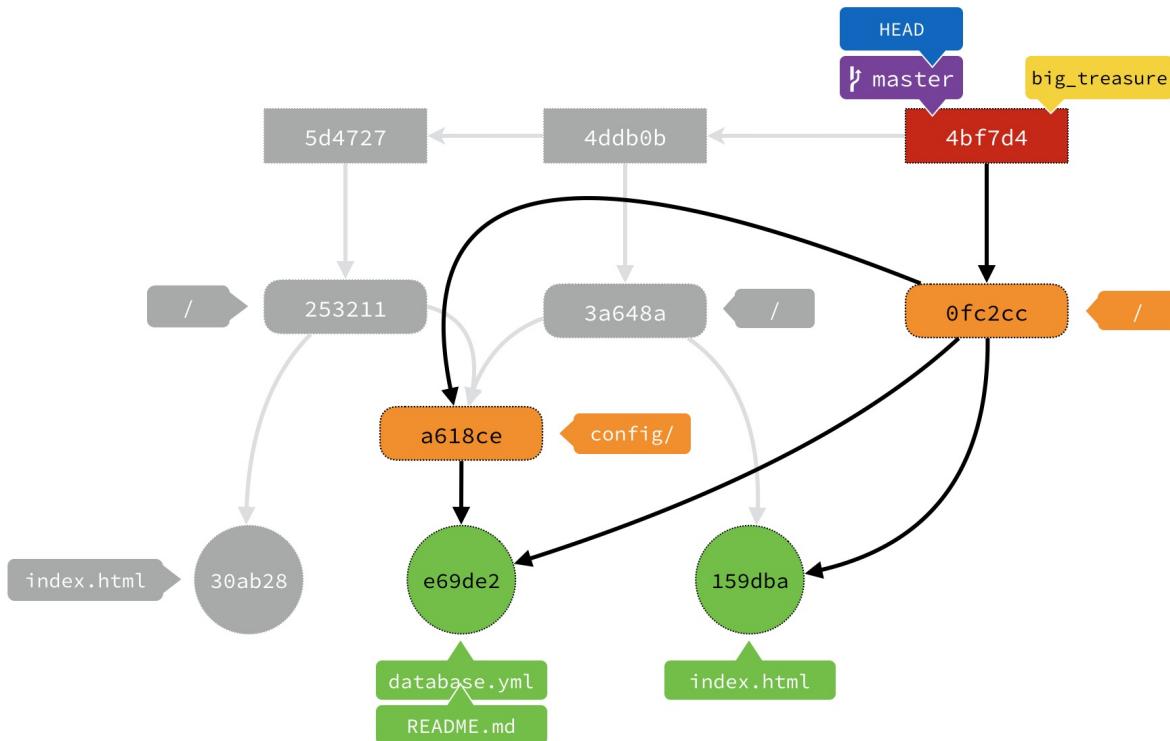
上個章節我們介紹了在 `.git/objects` 這個目錄裡的內容，以及 Git 的一些運作原理，接下來再看看當 Git 在 Checkout 的時候發生什麼事情，看看為什麼 Git 可以這麼快的就切換狀態。

Git 在 Checkout 時候的變化

在上個章節，做了三次的 Commit 之後，最後各個物件的關連圖是這個樣子：



雖然 Git 保留了完整的內容在 `.git/objects` 目錄裡，但在工作目錄的內容，會根據當下的這個 Commit，一個一個的把所有物件抽出來，就跟葡萄一樣，從頭的地方拎起來，整串就抽出來了，所以目前工作目錄的內容應該只有這些：



一樣以我們這個例子來說，當我試著 Checkout 到另一個 Commit 的時候：

```
$ git checkout 4dbb0b5
Note: checking out '4dbb0b5'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

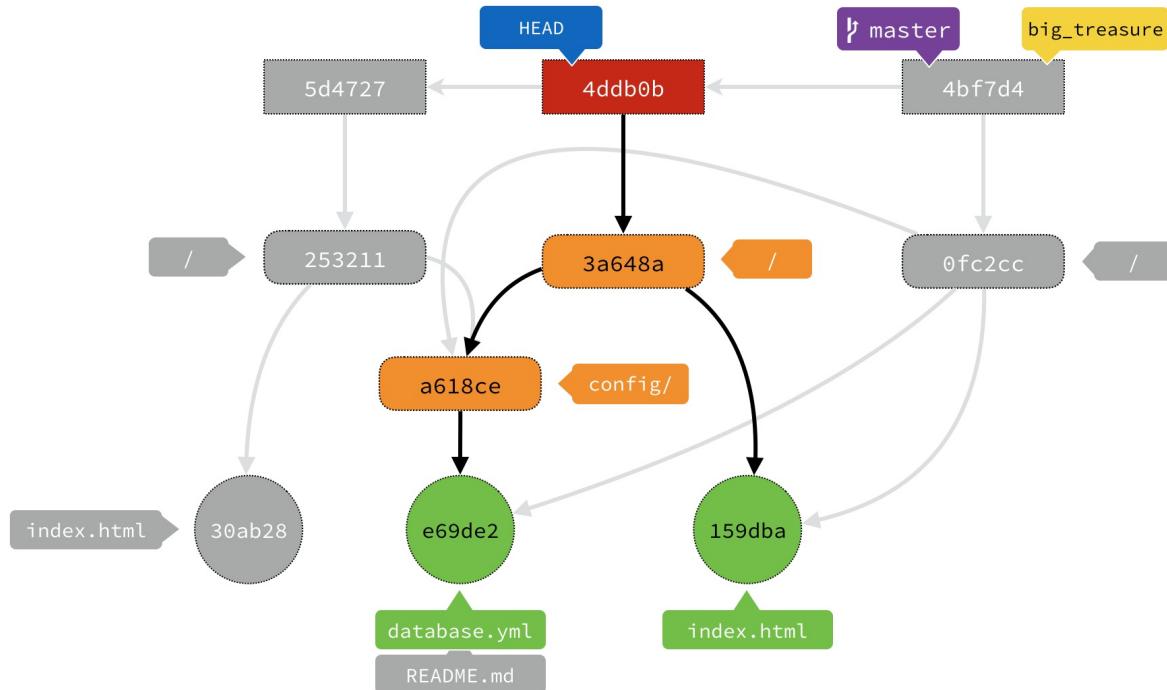
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 4dbb0b5... update index.html
```

因為 4dbb0b5 這個 Commit 沒有分支指著，所以會發生 detached HEAD 的狀況，關於 detached HEAD，可參閱「[【冷知識】斷頭（detached HEAD）是怎麼一回事？](#)」章節說明。

當 Git 切換到這個節點的時候，這時候這串「葡萄」的樣子會像這樣：



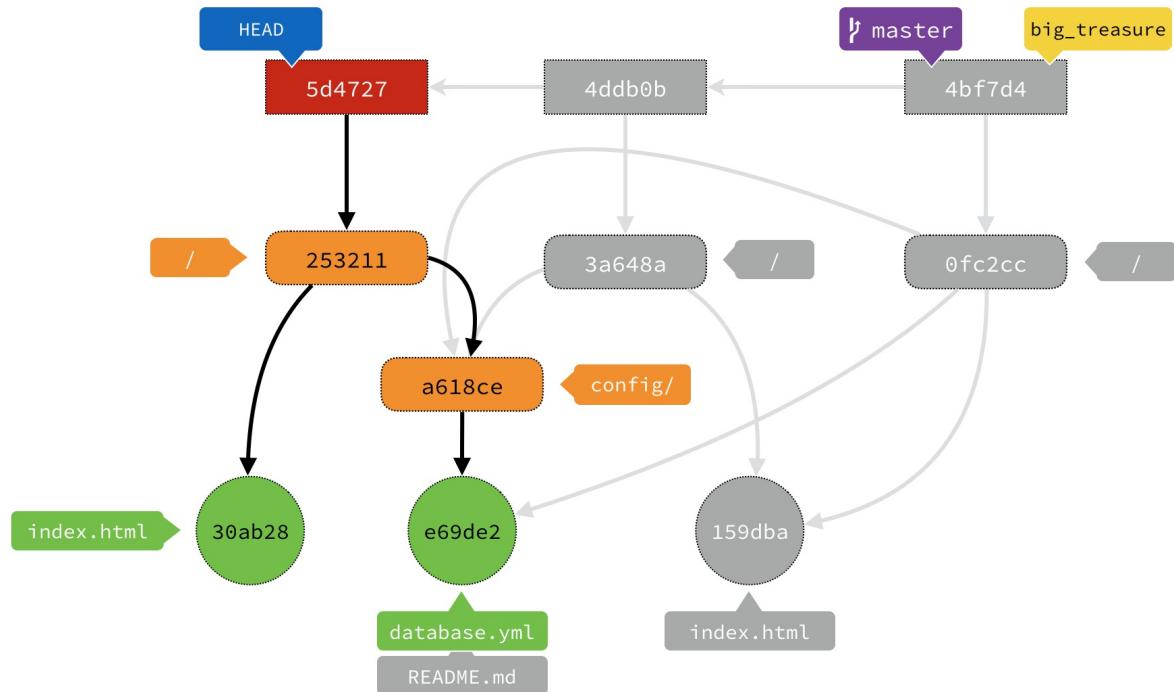
Git 根據這串資訊，把原本躺在 `.git/objects` 裡以 SHA-1 計算命名的目錄以及檔案，一個一個把檔案及目錄復原成原來的樣子，所以現在的目錄內容會像這樣：

```
$ ls -al
total 8
drwxr-xr-x  5 eddie  staff  170 Aug 24 12:39 .
drwxr-xr-x  8 eddie  staff  272 Aug 22 16:23 ..
drwxr-xr-x 13 eddie  staff  442 Aug 24 12:39 .git
drwxr-xr-x  3 eddie  staff  102 Aug 23 06:22 config
-rw-r--r--  1 eddie  staff  143 Aug 24 12:39 index.html
```

在第 3 次 Commit 才加進來的 `README.md`，因為這串葡萄裡並沒有提到它，所以它自然就不會被拎出來。同理，當再繼續往前切換的時候：

```
$ git checkout 5d47270
Previous HEAD position was 4ddb0b5... update index.html
HEAD is now at 5d47270... init commit
```

現在的狀況就變這樣了：



檢視目錄內容：

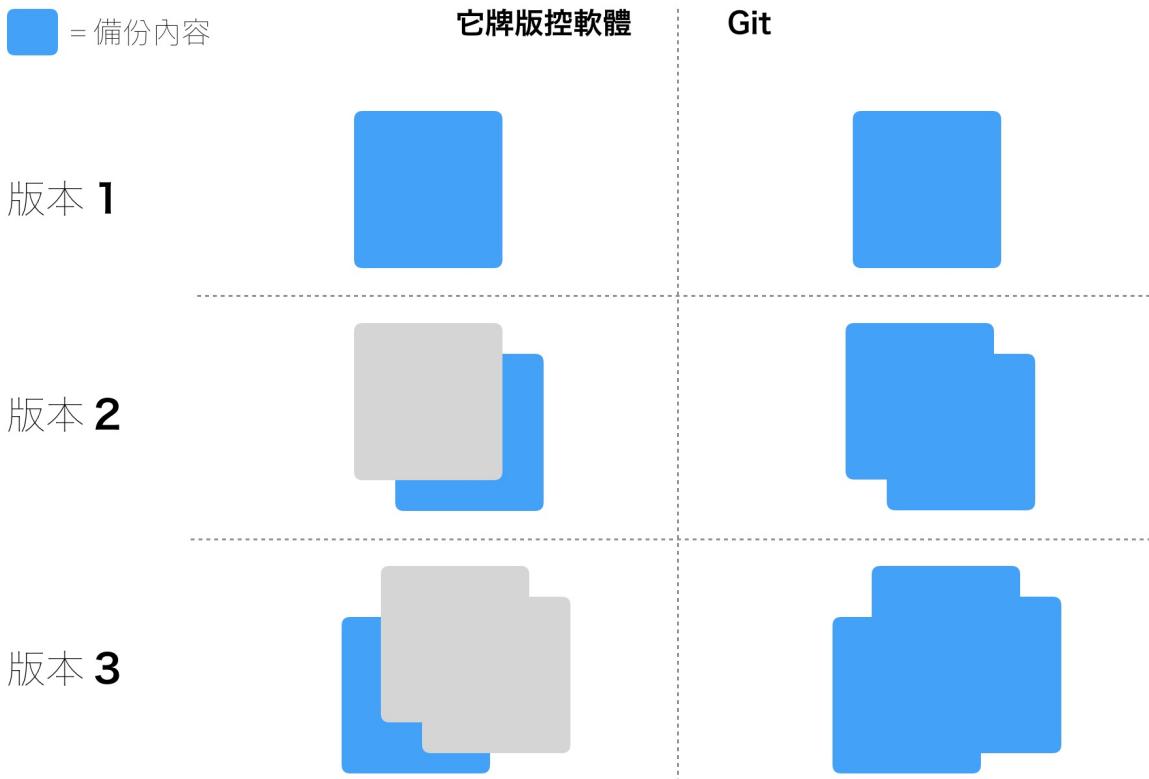
```
$ ls -al
total 8
drwxr-xr-x  5 eddie  staff  170 Aug 24 12:39 .
drwxr-xr-x  8 eddie  staff  272 Aug 22 16:23 ..
drwxr-xr-x 13 eddie  staff  442 Aug 24 12:39 .git
drwxr-xr-x  3 eddie  staff  102 Aug 23 06:22 config
-rw-r--r--  1 eddie  staff   14 Aug 24 12:35 index.html
```

如果能理解這個「拎葡萄」的概念，應該就能比較理解為什麼檔案跟目錄會有這樣的變化了。

更多關於分支以及 Checkout 時候發生的事情，例如 `.git/HEAD` 的變化、工作目錄、暫存區的變化，可參閱「[對分支的誤解](#)」章節的補充。

Git 不是做差異備份

有些版控系統，會備份每次 Commit 之間的「差異」，例如這次加了兩行、下次刪了五行之類的歷史紀錄，然後透過這些資訊，像拼圖一樣一個一個的把檔案還原成它該有的樣子。



不過 Git 並不是這樣設計的。從前面的流程看下來，當每次使用 `git add` 指令把檔案加到暫存區的時候，即使檔案的內容只改了一個字，因為算出來的 SHA-1 值不同，所以 Git 也會為它做出一顆全新的 Blob 物件，而不是只記錄「差異」。也因為 Git 在切換 Commit 的時候會像「拎葡萄」一樣整串抽出來，不需要一個一個去拼湊歷史紀錄，所以在做 Checkout 的時候效能相對的比較好。有些人也會用快照（Snapshot）來形容這個「拎葡萄」的概念。

但這樣感覺很浪費空間？

如果 Git 是用這種方式在存放檔案，就算只改一個字也會做出一顆新的物件，那你也可能會認為「這樣不是很浪費空間嗎？」。想想看，如果有一個 100KB 的檔案，因為改了一行程式，就必須再做出一顆也差不多是 100KB 的檔案出來，雖然 Git 在製作 Blob 物件的時候已先進行壓縮，但對於這樣「只差一點點就要備份整個檔案」的作法，看起來的確是有點浪費沒錯。

不過其實 Git 有所謂的「資源回收機制」，當這個機制發動的時候，Git 會用非常有效率的方式壓縮物件以及製作索引。我另外開一個全新的專案來舉例，這裡我先使用 `git ls-files -s` 指令來查詢目前這些檔案在 Git 裡的樣子：

```
$ git ls-files -s
100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0 config/database.yml
100644 159dba9a492eb6ee354beda28c88652dd7d1c2e2 0 index.html
```

目前只有 2 個檔案，分別是 `index.html` 以及在 `config` 目錄裡的 `database.yml`，而 `index.html` 的內容如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Hello, 5xRuby</h1>
  </body>
</html>
```

只是一個簡單的 HTML 頁面，這個檔案目前的大小是 143 bytes。接著對這個檔案做一些修改，像這樣：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>5xRuby</title>
  </head>
  <body>
    <h1>Hello, 5xRuby</h1>
  </body>
</html>
```

我只在 `<title></title>` 加上了 `5xRuby` 字樣。然後進行 Commit：

```
$ git add index.html

$ git commit -m "update index"
[master 23ee05d] update index
 1 file changed, 1 insertion(+), 1 deletion(-)
```

完成 Commit，檢視一下目前的狀態：

```
$ git ls-files -s
100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0      config/database.yml
100644 6303bc8384a837e7368692b03cb36cdf16d7f660 0      index.html
```

Git 的確為了這幾個字元的修改做了一顆新的 Blob 物件 `6303bc` 沒錯。

```
$ git cat-file -s 6303bc8384a837e7368692b03cb36cdf16d7f660
149
```

這顆新的 Blob 物件的大小是 149 bytes。只增加了 `5xRuby` 這六個字，Git 就為了它做了一顆新的 Blob 物件，這就是前面提到的浪費。就光這點來看的確是，但當 Git 發動「資源回收機制」，Git 會把這些檔案以非常有效率的方式打包並做好索引。Git 的資源回收機制通常會在它覺得物件太多的時候自動觸發，但也可直接執行 `git gc` 指令來手動觸發它：

```
$ git gc
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (11/11), done.
Total 11 (delta 1), reused 0 (delta 0)
```

這個指令，會把原本放在 `.git/objects` 目錄下的那些物件全部打包到 `.git/objects/pack` 目錄下，變成這個樣子：

```
$ find .git/objects -type f
.git/objects/info/packs
.git/objects/pack/pack-ea00f1558d67a7df25bf9744f3d83a17a7a2bf43.idx
.git/objects/pack/pack-ea00f1558d67a7df25bf9744f3d83a17a7a2bf43.pack
```

Git 有另一個比較底層的指令 `git verify-pack`，可以用來看看這個打包的狀況：

```
$ git verify-pack -v .git/objects/pack/pack-ea00f1558d67a7df25bf9744f3d83a17a7a2bf43.idx
23ee05d5652c770990e4c65d5a0b8ec34ba4f64f commit 215 151 12
44785d18bb2804a9455d7a2e0d9e9f60df482af7 commit 220 156 163
cd05448adf3ee9b11b3cf846a387d1af754b6191 commit 166 120 319
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 blob 0 9 439
6303bc8384a837e7368692b03cb36cdf16d7f660 blob 149 110 448
140454ce2b3d2d493653d01fb8d41ab6bcc22f87 tree 71 81 558
a618ce33da8d21bca841f18e6432fcabf15d4477 tree 40 51 639
3a648a98d322f82a72bff20ed977539c581a181d tree 71 82 690
159dba9a492eb6ee354beda28c88652dd7d1c2e2 blob 9 20 772 1 6303bc8384a837e7368692b03cb36cd
f16d7f660
2532115fab19c7ce70ff78a4929fb5df2f6131c4 tree 71 81 792
30ab28d3acb37f96ad61ad8be82c8da46d0a7307 blob 14 23 873
non delta: 10 objects
chain length = 1: 1 object
.git/objects/pack/pack-ea00f1558d67a7df25bf9744f3d83a17a7a2bf43.pack: ok
```

上面這些資訊的欄位，第一個欄位是物件的 SHA-1 值，第二個欄位是物件的型態，第三個則是檔案大小。先看一下這行：

```
6303bc8384a837e7368692b03cb36cdf16d7f660 blob 149 110 448
```

這個檔案是剛剛修改過的 `index.html`，檔案大小是 149，再看另一行：

```
159dba9a492eb6ee354beda28c88652dd7d1c2e2 blob 9 20 772 1 6303bc8384a837e7368692b03cb36cd
```

f16d7f660

這個檔案是修改前的 `index.html`，但它的大小竟然只有 9？原因是因為這個檔案參照了後面的那個檔案 `6303bc`，所以這個檔案才會這麼小。也就是說，雖然在物件狀態的時候是完整的檔案（打包前），在 Git 做資源回收打包的時候則是使用了類似差異備份的方式，有效的降低了這些物件的體積。

那 Git 什麼時候會自動觸發資源回收機制？

1. 當在 `.git/objects` 目錄的物件或是打包過的 `packfile` 數量過多的時候，Git 會自動觸發資源回收指令。
2. 當執行 `git push` 指令把內容推至遠端伺服器時（如果你有仔細觀察過 Push 指令的訊息的話...）。

話說回來，其實 Git 也不是真的很在意浪費空間這回事，反正現在硬碟這麼便宜，能夠快速、有效率的操作才是 Git 關切的重點。

為什麼要使用分支？

大家不知道有沒看過《火影忍者》漫畫？漫畫的主角之一 - 漩渦鳴人，他著名的忍術是「影分身術」。分支的概念就有點像「影分身術」，當你做出一隻新的分身（分支），這個分身會去執行任務或是打倒敵人，如果執行失敗了，最多就是那個分身消失，就再做一隻新的分身就行了，本體不會因此受到影響。

在開發的過程中，一路往前 Commit 也沒什麼問題，但當開始越來越多同伴一起在同一個專案工作的時候，可能就不能這麼隨興的想 Commit 就 Commit，這時候分支就很好用。例如想要增加新功能，或是修正 Bug，或是想實驗看看某些新的做法，都可以另外做一個分支來進行，待做完確認沒問題之後再合併回來，不會影響正在運行的產品線。

在多人團隊共同開發的時候，甚至也可引入像 Git Flow / GitHub Flow / GitLab Flow 之類的開發流程，讓同一個團隊的人都可以用相同的方式進行開發，減少不必要的溝通成本。

在 Git 使用分支非常方便，成本也很低（原因在「[【冷知識】為什麼大家都說在 Git 開分支「很便宜」？](#)」章節會再說明），所以即使是只有自己一個人進行開發，也很推薦使用分支喔。

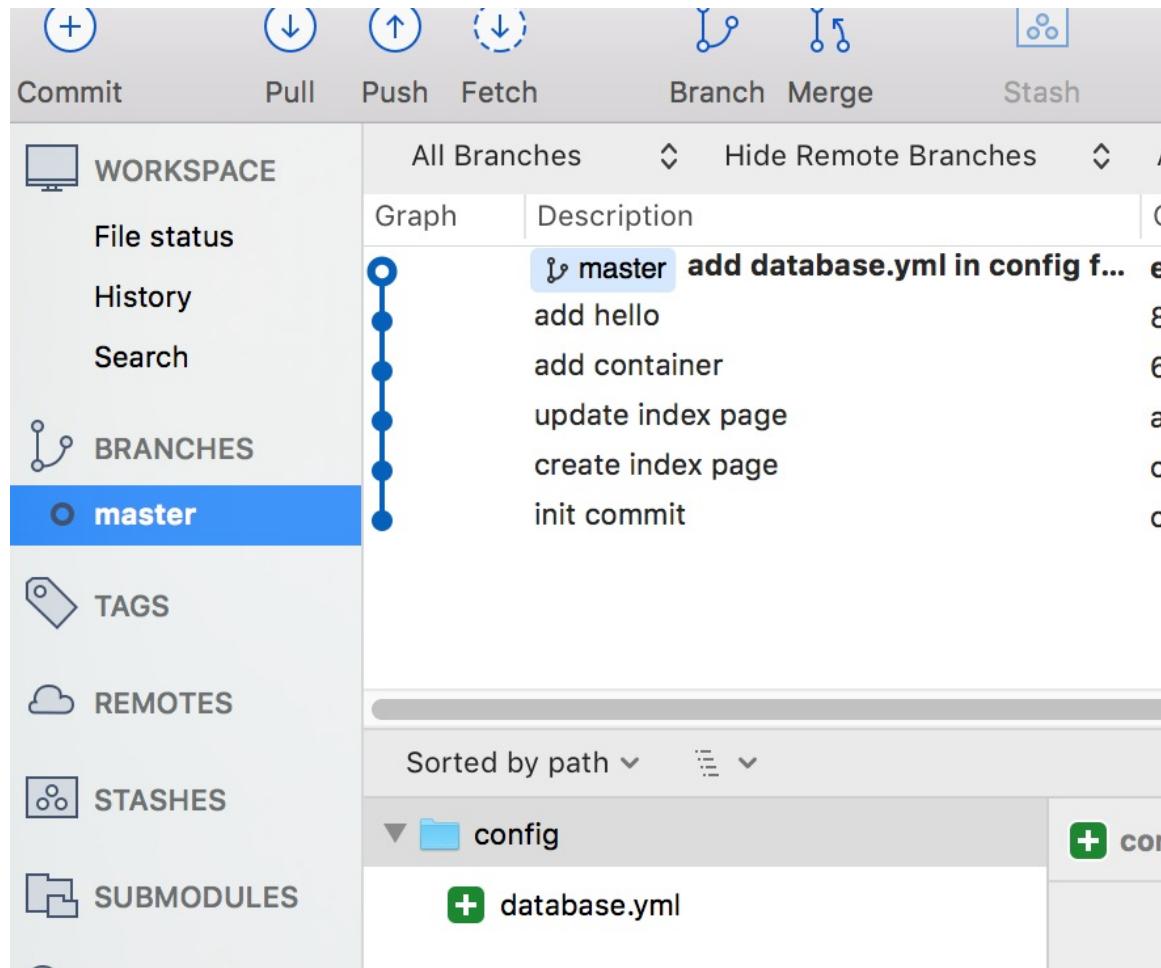
開始使用分支

在 Git 使用分支很簡單，只要使用 `git branch` 指令就行了：

```
$ git branch  
* master
```

如果 `git branch` 後面沒接任何參數，它僅會印出目前在這個專案有哪些分支。Git 預設會幫你設定一個名為 `master` 的分支，前面的星號 `*` 表示現在正在這個分支上。

在 SourceTree 看的話，會在左邊的選單上看到「BRANCHES」：



目前看到也只有 `master` 一條分支，在 `master` 字樣前面的空心小圓圈表示目前正在這個分支上（也就是 HEAD）。

新增分支

要增加一個分支，就是在執行 `git branch` 指令的時候，在後面加上想要的分支的名字：

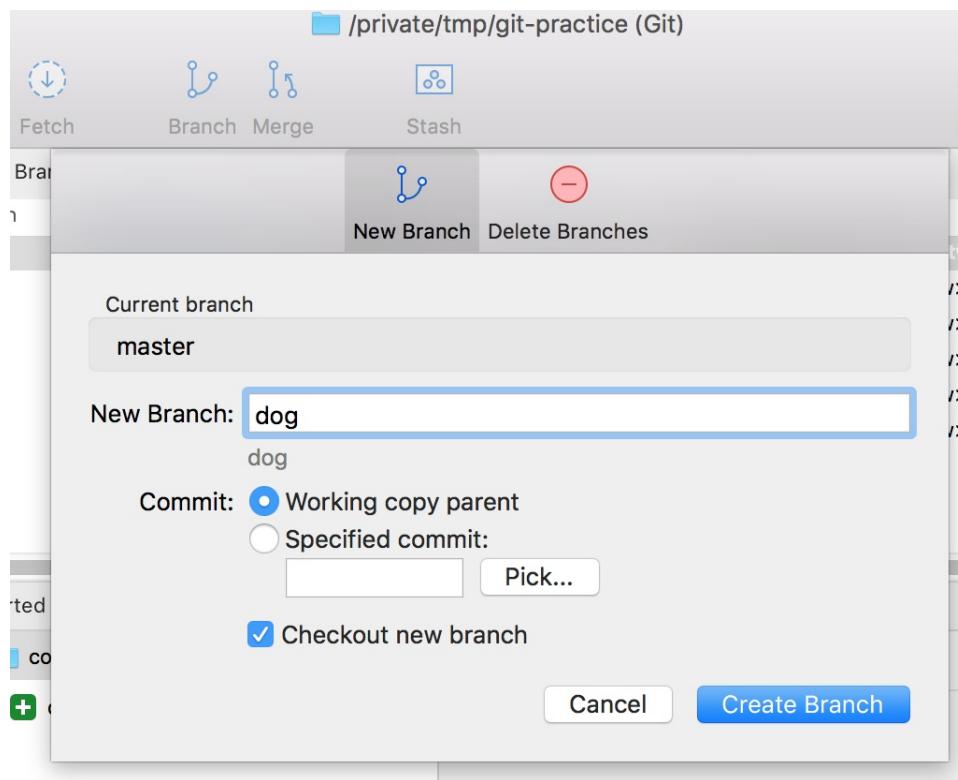
```
$ git branch cat
```

這樣就新增了一個 `cat` 分支，再檢視一下：

```
$ git branch
  cat
* master
```

的確是有多一個分支，但現分支還是在 `master` 上。

使用 SourceTree 開分支，只要在上面主選單按下「Branch」按鈕便會跳出對話框：



在這裡填寫想要開的分支的名字，最下方的「Checkout new branch」選項如果打勾的話，就會在建立分支完成之後直接切換到該分支。

分支改名字

如果覺得分支名字取的不夠響亮，想換隨時可以換，而且完全不會影響檔案或目錄。假設現在的分支有這三個：

```
$ git branch
  cat
```

```
dog
* master
```

然後我想把 `cat` 分支改成 `tiger` 分支，使用的是 `-m` 參數：

```
$ git branch -m cat tiger
```

看一下目前的分支：

```
$ git branch
tiger
dog
* master
```

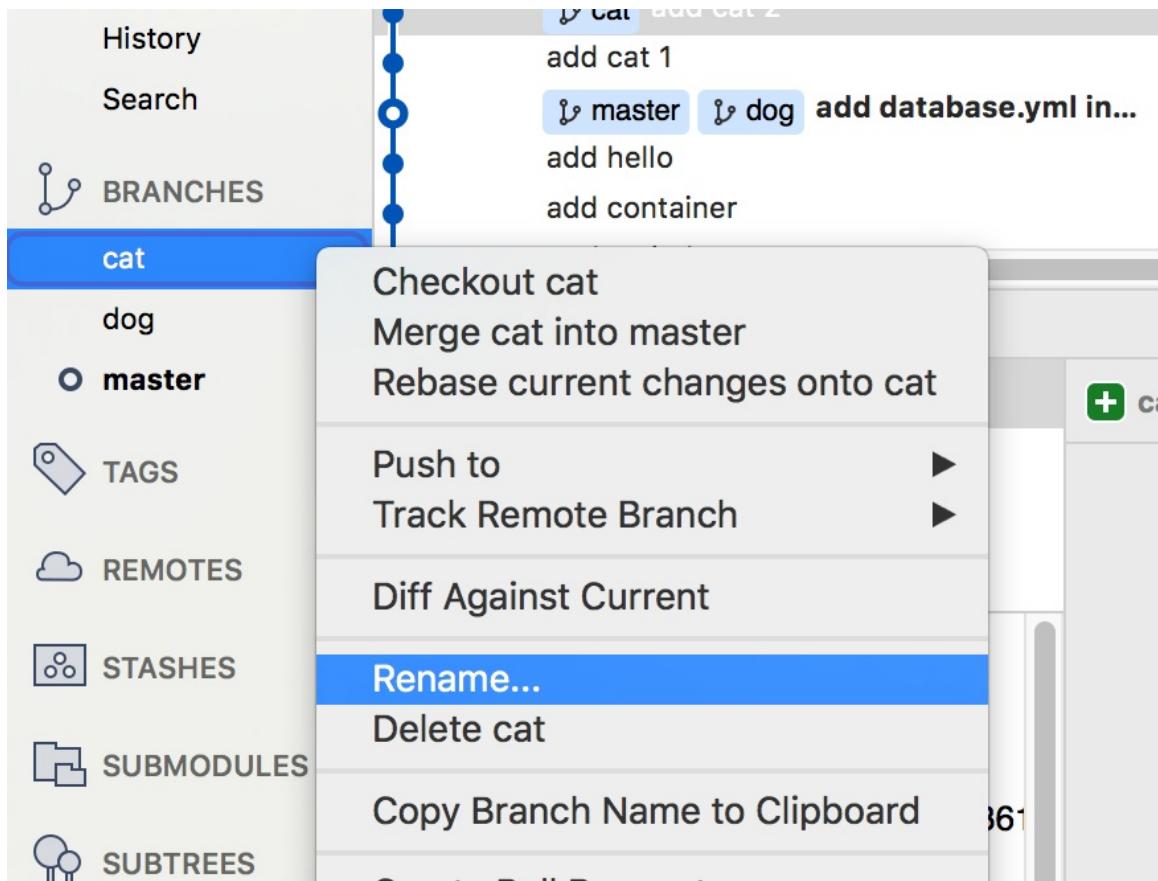
這樣就改掉了。而且即使是 `master` 分支想改也可以改，例如想把 `master` 改成 `slave`：

```
$ git branch -m master slave
```

看一下目前的分支：

```
$ git branch
tiger
dog
* slave
```

如果使用 SourceTree 要改分支名字也很簡單，只要在左邊選單的分支上按滑鼠右鍵，選擇「Rename」：



接著輸入想要修改的分支名稱即可。

刪除分支

檢視一下目前的分支：

```
$ git branch
  cat
  dog
* master
```

目前共有三個分支，其中如果 `dog` 分支不想要了，可以使用 `-d` 參數來刪除它：

```
$ git branch -d dog
Deleted branch dog (was e12d8ef).

$ git branch
  cat
* master
```

`dog` 分支就不見了。但如果要刪的分支還沒被完全合併，Git 會有貼心小提示：

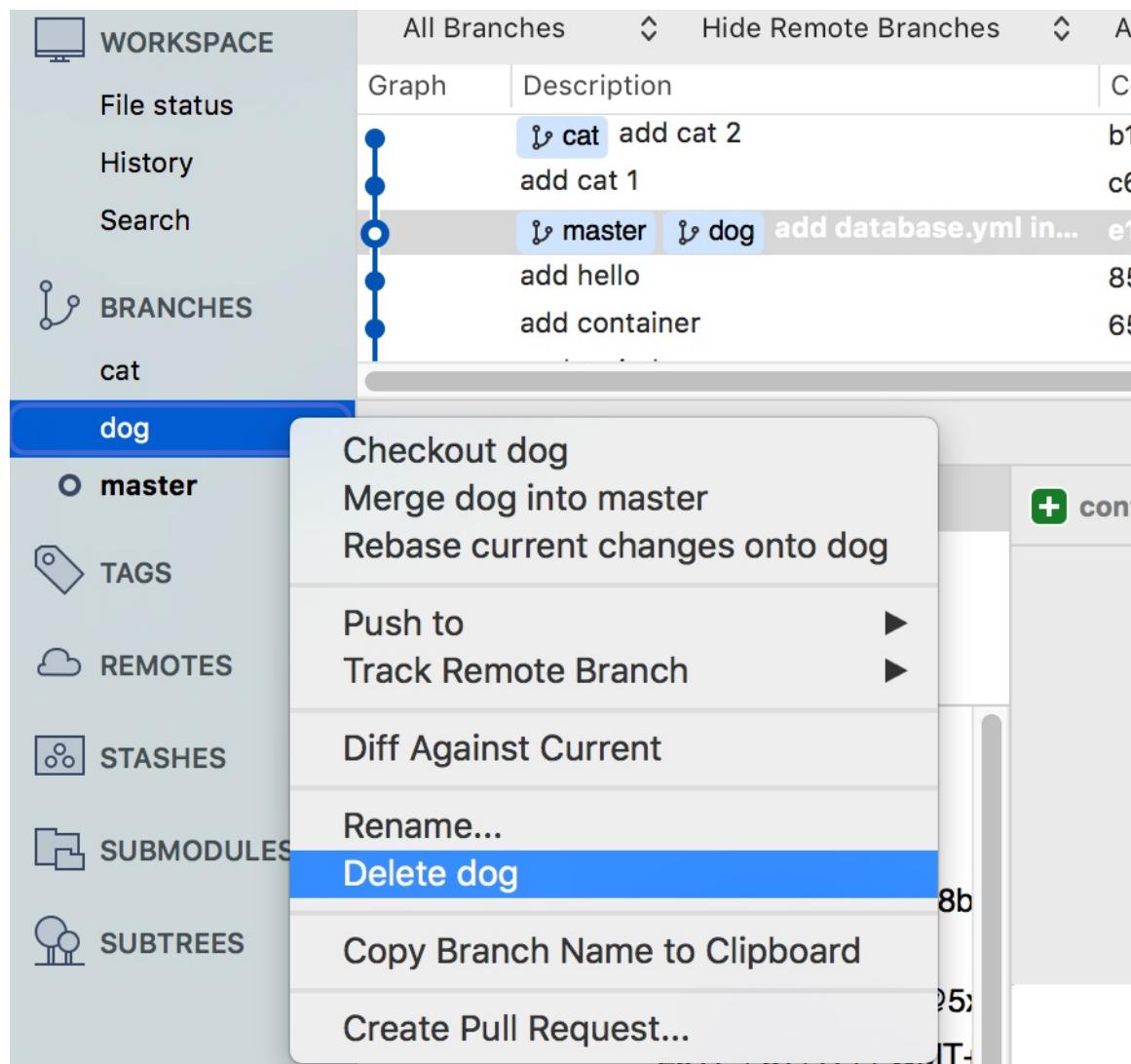
```
$ git branch -d cat
error: The branch 'cat' is not fully merged.
If you are sure you want to delete it, run 'git branch -D cat'.
```

的確，因為 `cat` 的內容還沒被合併，所以使用 `-d` 參數不給刪。這時只要改用 `-D` 參數就可以強制刪除：

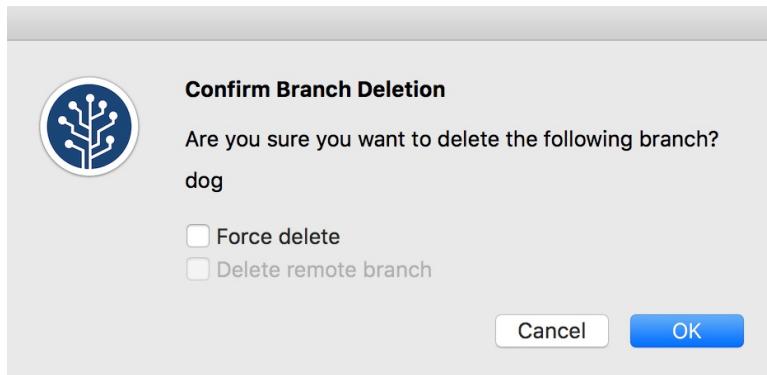
```
$ git branch -D cat
Deleted branch cat (was b174a5a).
```

使用 `-D` 參數可以強制把還沒合併的分支砍掉，但如果後悔想救回來，請見「[【狀況題】不小心把還沒合併的分支砍掉了，救得回來嗎？](#)」章節的說明。

使用 SourceTree 的話，則是在左邊分支的選單上按滑鼠右鍵，選擇「Delete...」功能：



它會出現一個對話框，按下 OK 按鈕就可以把指定的分支刪除：



同樣的，如果該分支還沒被完全合併但仍想強制刪除，則勾選「Force delete」就可順利刪除。

沒有什麼分支是不能刪的！

在 Git 裡什麼分支都可以刪，包括預設的 `master` 也可以，`master` 分支只是預設的分支，它並沒有比較特別。真的硬要說哪個分支不能刪的，只有「現在目前所在的分支」不能刪而已（因為刪了的話要去哪裡？），不過只要先切到別的分支就可以刪掉它了。

切換分支

要切換分支，使用的指令在前面「[【狀況題】啊！不小心把檔案或目錄刪掉了…](#)」章節也曾經出現過，就是 `git checkout`：

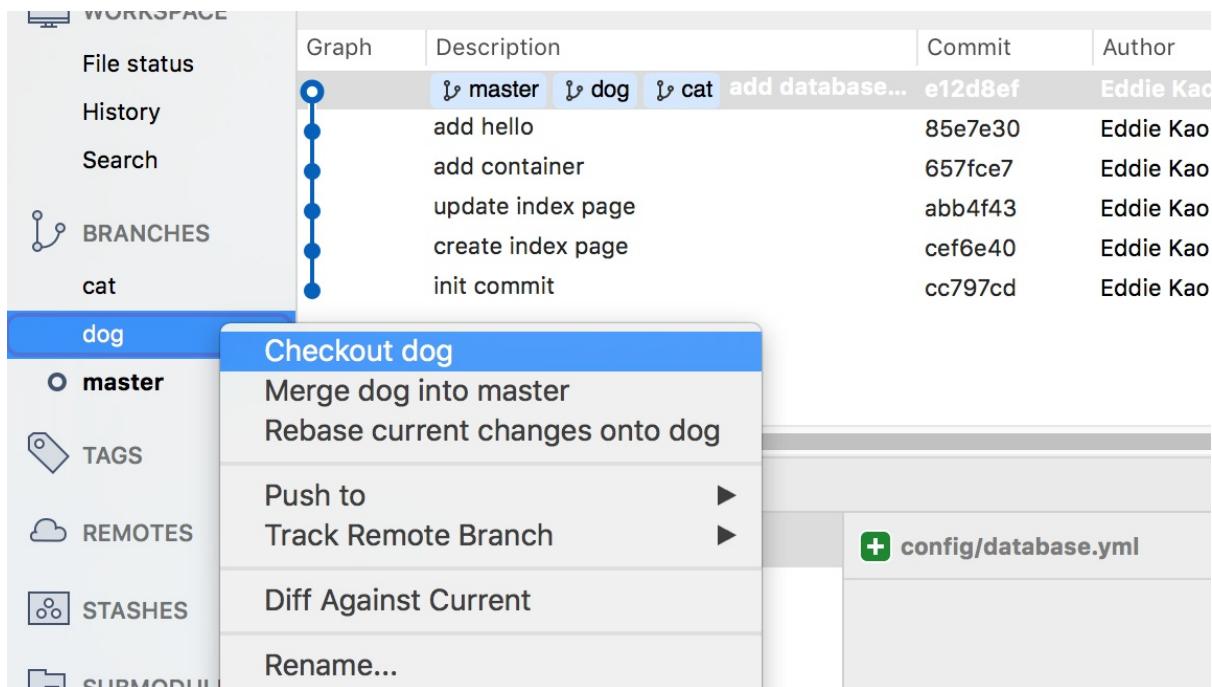
```
$ git checkout cat  
Switched to branch 'cat'
```

看一下目前的分支狀態：

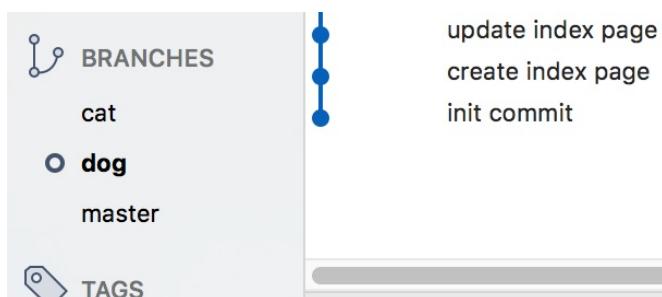
```
$ git branch  
* cat  
  dog  
  master
```

前面的那個星號已經移到 `cat` 分支上了。

如果是使用 SourceTree，則是在左邊選單的分支名字上按滑鼠右鍵，選擇「Checkout ...」



或是直接在分支名字上點兩下也可以切換過去，切換成功的話前面的空心小圈圈（也就是那個 HEAD 啦）就會移到切換的分支上了。



接下來操作就跟一般的差不多，一樣都是先 `add` 再 `commit`，但不一樣的是，當你在 `Commit` 的時候就只有那個分支會前進喔。

切換分支的時候...

舉個例子來說，我切換到 `cat` 線，然後在這邊加了兩次 Commit，分別新增一個 `cat1.html` 跟 `cat2.html`：

```
$ git checkout cat
Switched to branch 'cat'

$ touch cat1.html

$ git add cat1.html

$ git commit -m "add cat 1"
[cat c68537b] add cat 1
 1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 cat1.html

$ touch cat2.html

$ git add cat2.html

$ git commit -m "add cat 2"
[cat b174a5a] add cat 2
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 cat2.html
```

看一下 Git 紀錄：

```
$ git log --oneline
b174a5a (HEAD -> cat) add cat 2
c68537b add cat 1
e12d8ef (master, dog) add database.yml in config folder
85e7e30 add hello
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

的確比 `master` 以及 `dog` 分支多前進了兩次的 Commit。再看一下檔案列表：

```
$ ls -al
total 16
drwxr-xr-x  9 eddie  wheel   306 Aug 17 18:38 .
drwxrwxrwt 72 root   wheel 2448 Aug 17 18:18 ..
drwxr-xr-x 16 eddie  wheel   544 Aug 17 18:38 .git
-rw-r--r--  1 eddie  wheel     0 Aug 17 18:38 cat1.html
-rw-r--r--  1 eddie  wheel     0 Aug 17 18:38 cat2.html
drwxr-xr-x  3 eddie  wheel   102 Aug 17 15:06 config
-rw-r--r--  1 eddie  wheel     0 Aug 17 15:06 hello.html
-rw-r--r--  1 eddie  wheel   161 Aug 17 18:00 index.html
-rw-r--r--  1 eddie  wheel    11 Aug 17 14:56 welcome.html
```

在這兩次的 Commit 中，共新增了 `cat1.html` 及 `cat2.html` 這兩個檔案。這時候，如果切換回原本的 `master` 分支：

```
$ git checkout master
Switched to branch 'master'
```

再看一下檔案列表：

```
$ ls -al
total 16
drwxr-xr-x  7 eddie  wheel   238 Aug 17 19:10 .
drwxrwxrwt 72 root   wheel 2448 Aug 17 18:18 ..
drwxr-xr-x 16 eddie  wheel   544 Aug 17 19:10 .git
```

```
drwxr-xr-x  3 eddie  wheel  102 Aug 17 15:06 config
-rw-r--r--  1 eddie  wheel     0 Aug 17 15:06 hello.html
-rw-r--r--  1 eddie  wheel   161 Aug 17 18:00 index.html
-rw-r--r--  1 eddie  wheel    11 Aug 17 14:56 welcome.html
```

咦？剛剛那兩個檔案不見了！別擔心，其實他們都還在的，只是在不同的分支而已，只要切回 `cat` 分支，檔案就會出現了。

要切換到那個分支，首先你要有那個分支...

如果要切換到某個分支，那個分支必要先存在，不然會發生錯誤：

```
$ git checkout sister
error: pathspec 'sister' did not match any file(s) known to git.
```

醒醒吧，你沒有這個分支...但沒關係，在 `git checkout 分支名稱` 的時候加上 `-b` 參數就沒問題了。如果這個分支本來就存在，那就會直接切換過去；如果不存在，Git 就會幫你建一個，然後再切換過去：

```
$ git checkout -b sister
Switched to a new branch 'sister'
```

這樣就搞定了！

對分支的誤解

在上個章節介紹了如何使用分支，但我相信應該有些人（即使已經使用 Git 好一陣子）對分支是有些誤解的，本章將介紹關於分支的一些觀念。

你認為的分支長什麼樣子？

首先，如果你曾經使用過 Git 的分支，你想像中的分支會是什麼樣子？是這樣？



photo by [Mark Fischer](#)

還是這樣？



photo by Caroline

分支是什麼？

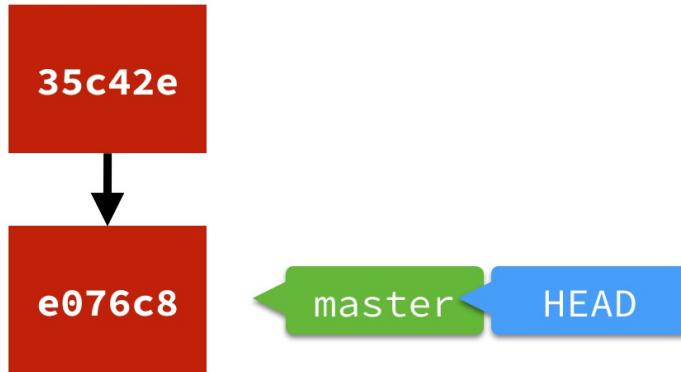
所以分支是什麼？有人可能認為，所謂的「開分支」是把檔案先複製到另外的目錄，然後進行修改之後再合併，把檔案跟原本的檔案比對之後放回原本的目錄... 其實 Git 不是這樣的。

分支像貼紙一樣...

你可以把分支想像成一張貼紙，它是貼在某一個 Commit 上面，像這樣：



當做了一次新的 Commit 之後，這個新的 Commit 會指向它的前一個 Commit：



而接下來「目前的分支」，也就是 HEAD 所指的這個分支，會跟著貼到剛剛做的那個 Commit 上，同時 HEAD 也跟著前進：

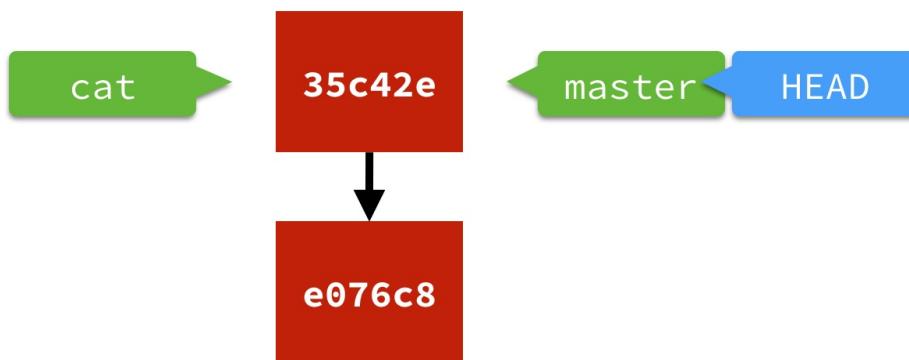


至於什麼是 HEAD，請參閱「[【冷知識】HEAD 是什麼東西？](#)」章節說明。

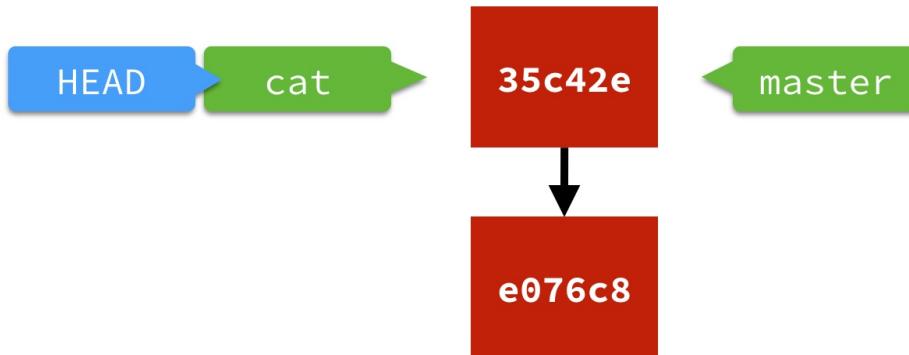
不知道這樣你對分支是不是比較有概念了？在 Git 的分支並不是複製什麼目錄或檔案來進行修改，分支就只是一個指標、一張貼紙，貼在某個 Commit 上面而已。

分支一個不夠，就來兩個吧

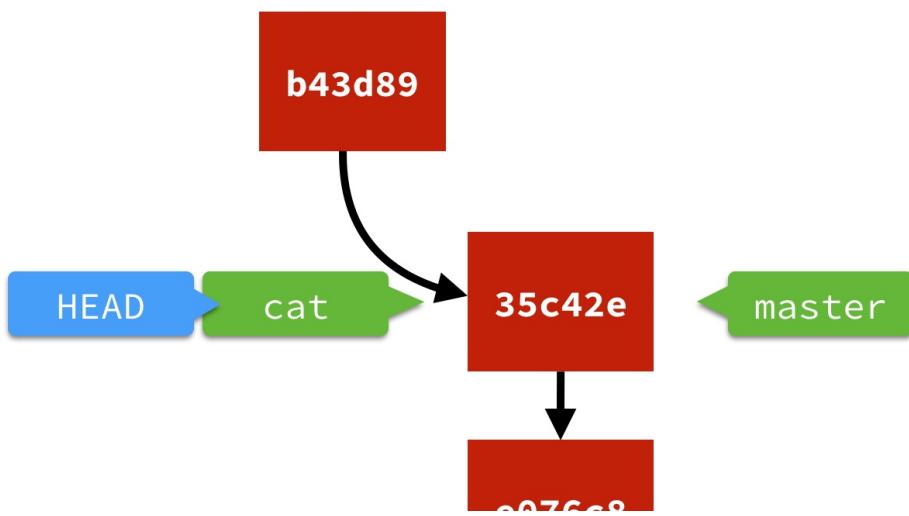
如果一個分支不夠說明，那就來兩個。我透過指令 `git branch cat` 開了一個新的分支，別忘了，分支就像是一張貼紙一樣，它現在跟 `master` 貼在同一個地方：



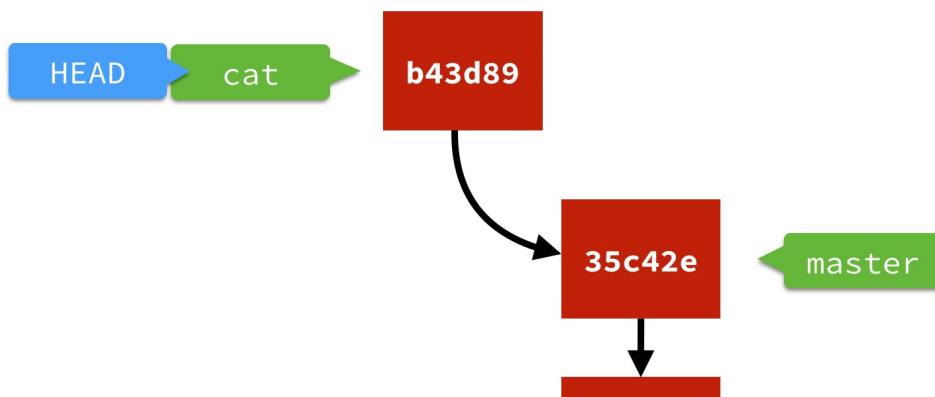
接下來執行 `git checkout cat` 指令切換到 `cat` 分支，此時，HEAD 便會換成指向 `cat` 分支，表示它是「目前的分支」：



接著，我做了一次新的 Commit，這個新的 Commit 會指向前一次 Commit：



然後，`cat` 分支這張貼紙就會被撕下來，換貼到最新的那個 Commit 上了，當然 HEAD 也是跟著去：



在 Git 的分支大概就是這樣一回事。

【冷知識】在切換分支的時候發生了什麼事？

在「[開始使用分支](#)」章節有提到一段，原本在 `cat` 分支新增的 `cat1.html` 跟 `cat2.md` 檔案，當切換回 `master` 分支之後就不見了，但切回 `cat` 分支就又出現了...Git 在切換分支的時候到底做了什麼事情？明明前面提到分支不是複製目錄或檔案出來修改再放回去，到底是什麼黑魔法，可以切換分支就有這種神奇的效果？

Git 在切換分支的時候主要做了兩件事：

1. 更新暫存區以及工作目錄

Git 在切換分支的時候，會用該分支指向的那個 Commit 的內容來「更新」暫存區（Staging Area）以及工作目錄（Working Directory）。但在切換分支之前所做的修改則還是會留在工作目錄，不受影響。

這句話看起來有點難理解，先來看看前半段的意思。假設我原本正處於 `cat` 分支，當執行下面這個指令，就會由 `cat` 分支切換到 `master` 分支：

```
$ git checkout master
```

接下來，Git 會用 `master` 這個分支指向的那個 Commit 的內容，拿來更新暫存區以及工作目錄。因為這 `master` 這時候指的這個 Commit 並沒有 `cat1.html` 跟 `cat2.html` 這兩個檔案，所以「更新」之後，不管在暫存區或是工作目錄都不會有這兩個檔案。同理，當我再次切回 `cat` 分支的時候：

```
$ git checkout cat
```

Git 會拿 `cat` 分支指向的那個 Commit 的內容來「更新」暫存區以及工作目錄的檔案，所以這兩個檔案就又長出來了。

等等，什麼叫做「那個 Commit 的內容」？

在 Git 的世界裡，每一次的 Commit 都是一個物件，它會指向某一個 Tree 物件（目錄），而這些 Tree 物件會指向其它的 Tree 物件（子目錄）或是 Blob 物件（檔案）。

這結構有點像葡萄一樣，只要伸手從源頭的 Commit 物件拎起來，整串內容都可以被拿出來。關於這些 Git 的物件，可參閱「[【超冷知識】在 .git 目錄裡有什麼東西？Part 1](#)」章節，會有更詳細的說明。



photo by [Erika Rathje](#)

2. 變更 HEAD 的位置

除了更新暫存區以及工作目錄的內容外，同時 HEAD 也會指向剛剛切換過去的那個分支，也就是說 `.git/HEAD` 這個檔案會跟著被修改。關於 HEAD 的介紹可參閱「[【冷知識】HEAD 是什麼東西？](#)」章節。

那如果改一半切換分支會發生什麼事？

假設我現在還在 `cat` 分支，在切換到 `master` 分支之前，新增了一個 `cat3.html` 檔案，同時也修改了 `index.html` 檔案的內容，這時候狀態是：

```
$ git status
On branch cat
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    cat3.html

no changes added to commit (use "git add" and/or "git commit -a")
```

目前 index.html 是 modified 狀態，而 cat3.html 是 Untracked 狀態。然後如果在這時候就直接切換到 master 分支，也就是還沒收功就罵髒話...不是，是還沒 Commit 就切換分支，會發生什麼事？

如果各位有注意到上一段提到的內容：

Git 在切換分支的時候，會用該分支指向的 Commit 的內容來「更新」暫存區（Staging Area）以及工作目錄（Working Directory）。但在切換分支之前做的修改則還是會留在工作目錄，不受影響。

後半段說的「在切換分支之前做的修改則還是會留在工作目錄，不受影響」就是這件事。直接來操作看看：

```
$ git checkout master
M     index.html
Switched to branch 'master'
```

看一下檔案列表：

```
$ ls -al
total 16
drwxr-xr-x  8 eddie  wheel   272 Aug 18 03:50 .
drwxrwxrwt 84 root   wheel  2856 Aug 18 03:03 ..
drwxr-xr-x 16 eddie  wheel   544 Aug 18 03:50 .git
-rw-r--r--  1 eddie  wheel    0 Aug 18 03:48 cat3.html
drwxr-xr-x  3 eddie  wheel   102 Aug 17 15:06 config
-rw-r--r--  1 eddie  wheel    0 Aug 18 03:27 hello.html
-rw-r--r--  1 eddie  wheel   174 Aug 18 03:27 index.html
-rw-r--r--  1 eddie  wheel   11 Aug 17 14:56 welcome.html
```

那個剛剛新增的 cat3.html 還活著。再看一下 Git 的狀態：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    cat3.html

no changes added to commit (use "git add" and/or "git commit -a")
```

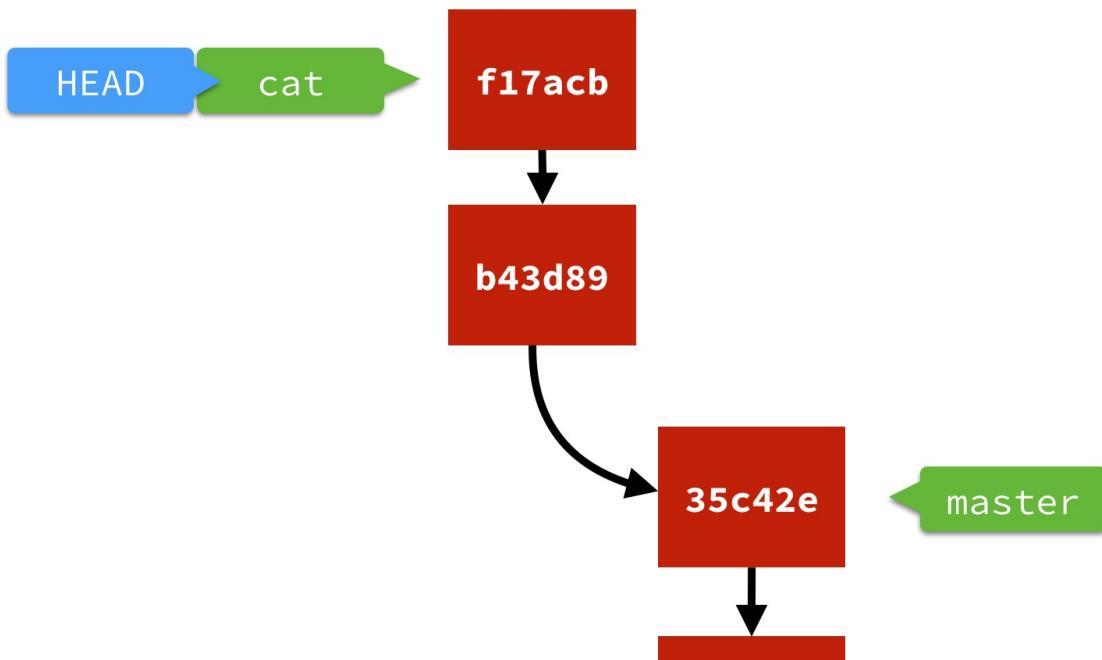
跟剛剛在 `cat` 分支的時候的狀態是一樣的，也就是說換切分支並不會影響已經在工作目錄的那些修改喔。

小結

不知道上面這些內容跟你原本想像或認知的分支有沒有一樣呢？分支在 Git 裡的使用頻率非常高，甚至可以說是大家使用 Git 的主要原因之一，所以大家一定要建立分支的正確觀念，這樣一來不管是在終端機畫面使用指令或是用圖形介面工具，在操作分支的時候就更不容易出問題囉。

合併分支

在前面章節的例子中，我從 `master` 分支開了一個 `cat` 分支，並且做了兩次 Commit，現在看起來的樣子大概像這樣：



任務執行的差不多了，就要準備合併回來了。如果我想要 `master` 分支來合併 `cat` 分支的話，我會先切回 `master` 分支：

```
$ git checkout master
Switched to branch 'master'
```

接下來，要合併分支是使用 `git merge` 指令：

```
$ git merge cat
Updating 35c42e..f17acb
Fast-forward
 cat1.html | 0
 cat2.html | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat1.html
 create mode 100644 cat2.html
```

看一下檔案列表：

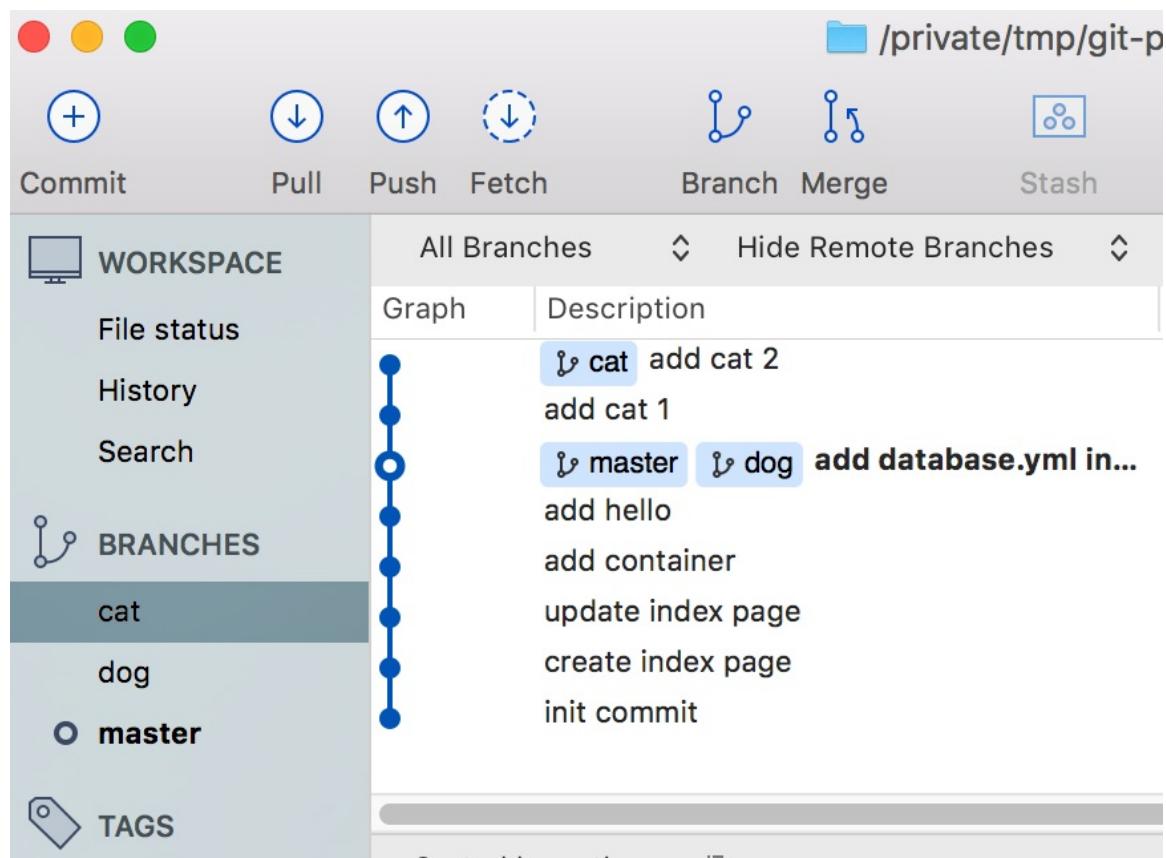
```
$ ls -al
total 16
```

```
drwxr-xr-x  9 eddie  wheel  306 Aug 18 05:14 .
drwxrwxrwt 84 root   wheel  2856 Aug 18 04:29 ..
drwxr-xr-x 16 eddie  wheel  544 Aug 18 05:15 .git
-rw-r--r--  1 eddie  wheel    0 Aug 18 05:14 cat1.html
-rw-r--r--  1 eddie  wheel    0 Aug 18 05:14 cat2.html
drwxr-xr-x  3 eddie  wheel  102 Aug 17 15:06 config
-rw-r--r--  1 eddie  wheel    0 Aug 18 03:27 hello.html
-rw-r--r--  1 eddie  wheel  161 Aug 18 04:24 index.html
-rw-r--r--  1 eddie  wheel   11 Aug 17 14:56 welcome.html
```

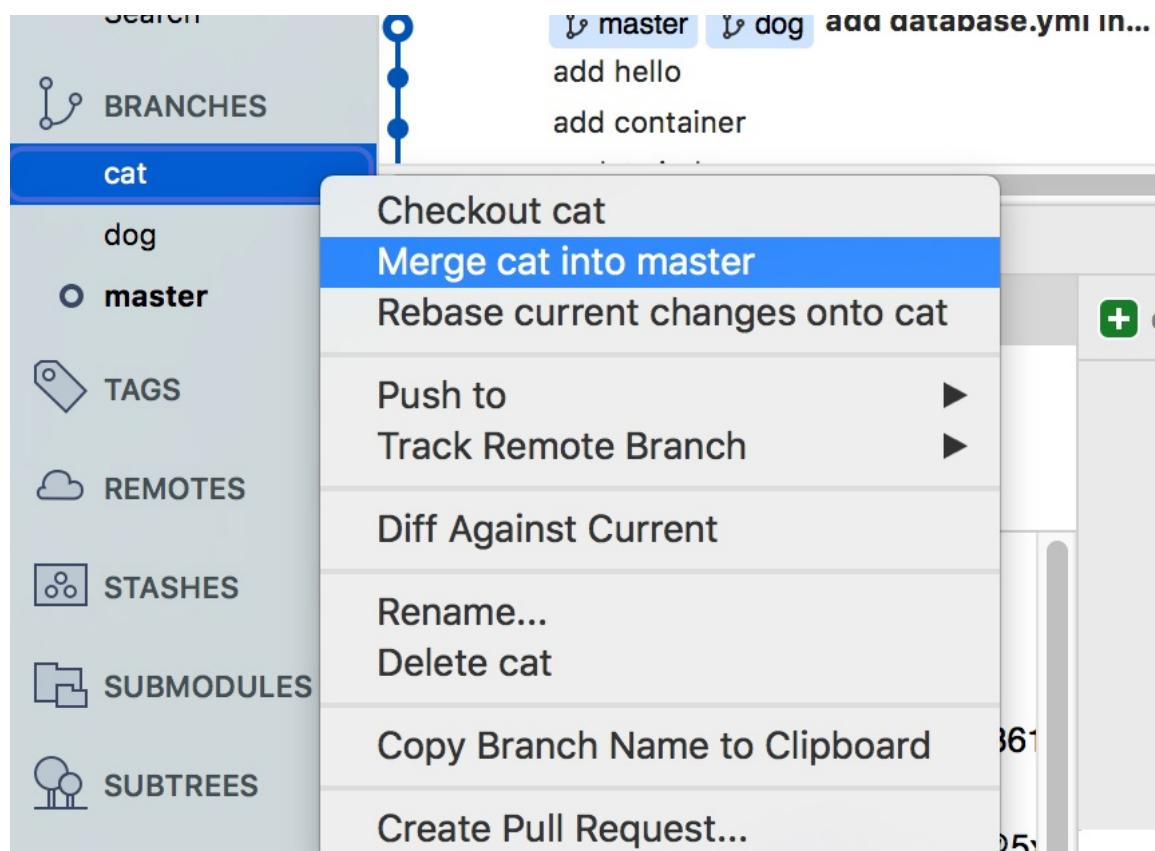
在 `cat` 分支新增的 `cat1.html` 跟 `cat2.html`，因為 `master` 現在已經合併 `cat` 分支，所以現在在 `master` 分支也有一份了。

回到 SourceTree 看一下目前的狀況：

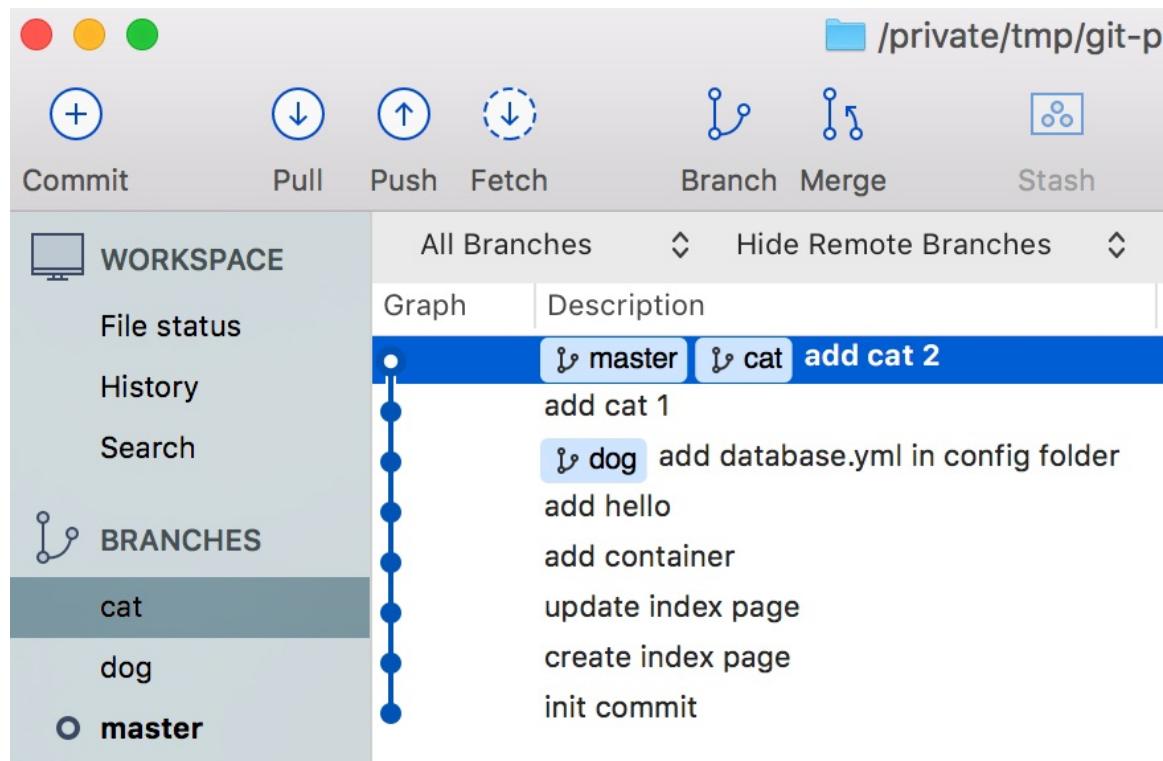
從左邊的「BRANCHES」選單看得出來現在正處於 `master` 分支，同時從右方的 Commit 紀錄也看得出來 `cat` 分支現在領先 `master` 分支 2 個 Commit：



如果要合併 `cat` 分支，在分支上按滑鼠右鍵，選擇「Merge cat into master」：



它會跳出一個對話框，點擊 OK 按鈕便可完成合併。這時候再看一下右邊的 Commit 紀錄：

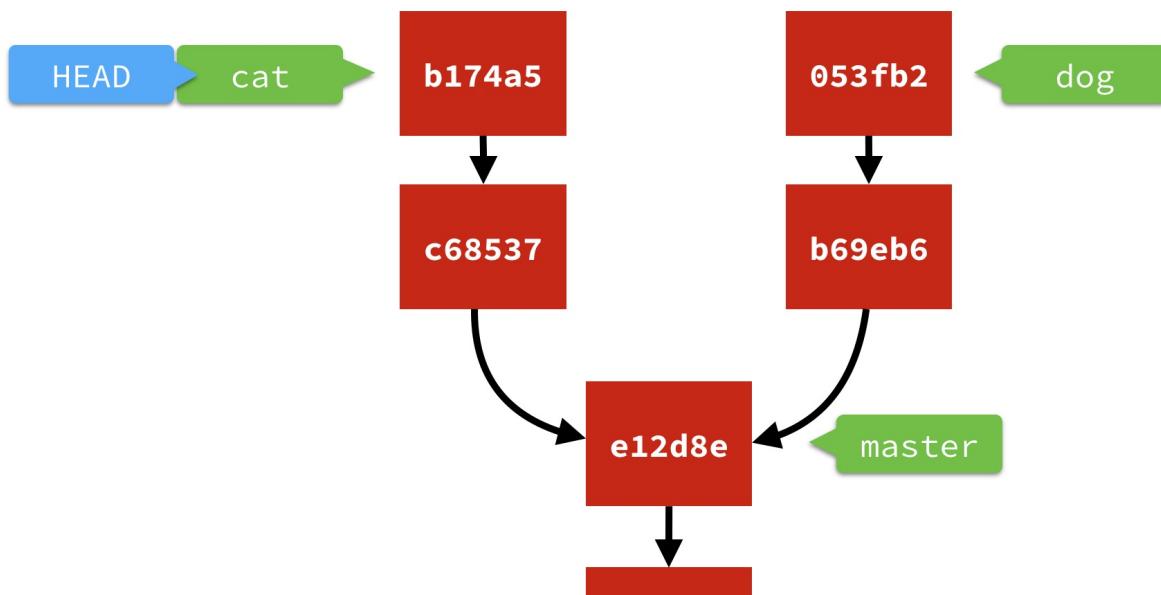


本來落後 2 個 Commit 的 `master` 分支，在進行合併之後，進度也已經跟上 `cat` 分支，跟它在同一個 Commit 上。

至於已經合併的分支要不要留下來？請見「[【常見問題】合併過的分支要留著嗎？](#)」章節說明。

A 合併 B，跟 B 合併 A 有什麼不同？

這個問題在我一開始學 Git 的時候也曾經困擾過我好一陣子，到底誰合併誰有那麼重要嗎？這就要看你著眼的重點是什麼了。如果以最終結果來看是一樣的，但過程可能會有些差別。我先各別從 `master` 分支做出了 `cat` 跟 `dog` 這兩個分支，並且現在正在 `cat` 分支：



`cat` 跟 `dog` 這兩個分支都是來自 `master` 分支，可以想像成是「`cat` 分支跟 `dog` 分支啊，你們身上都流著我 `master` 的血...」的意思，所以如果是 `master` 不管是要合併 `cat` 或是 `dog` 分支，Git 會直接使用快轉模式（Fast Forward）進行合併，說得白一點就是 `master` 直接「收割」`cat` 或 `dog` 的成果了。

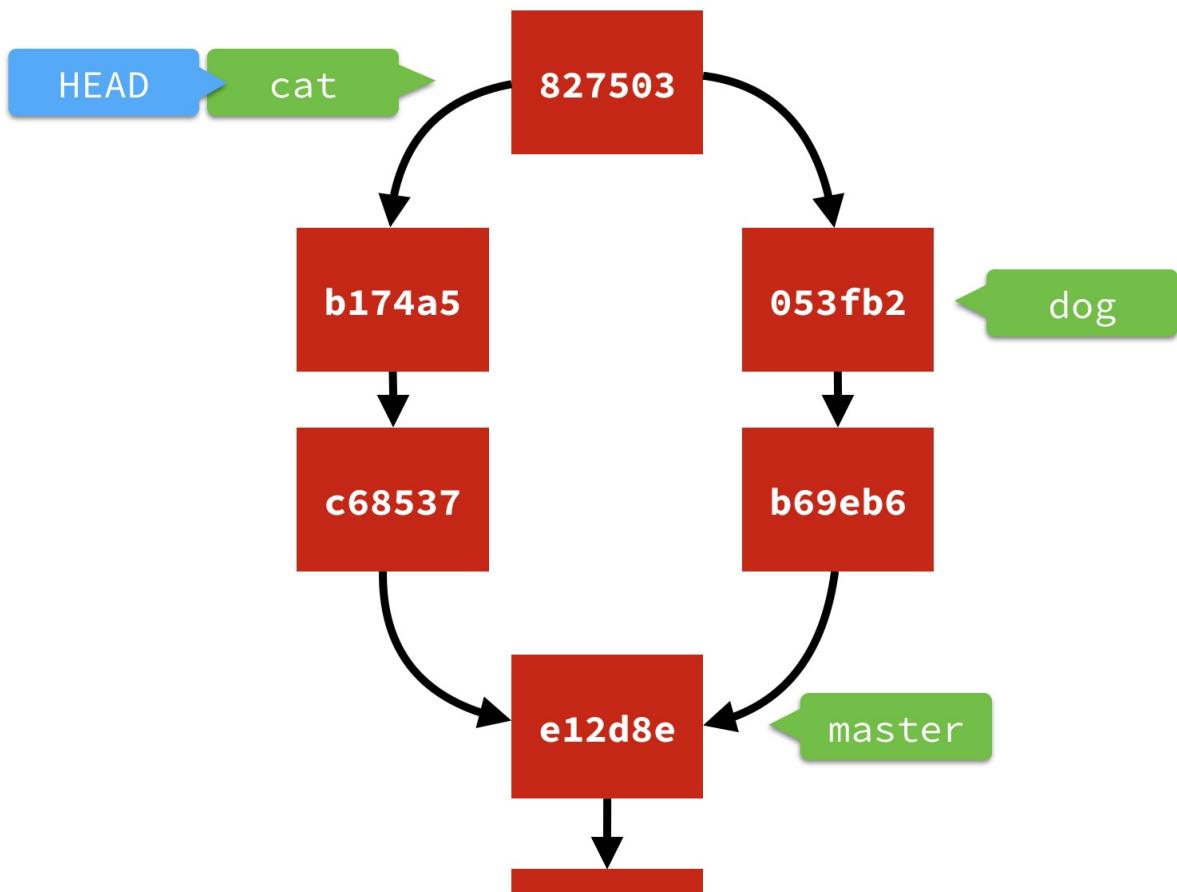
但如果是 `cat` 跟 `dog` 這兩個分支要互相合併就不一樣了，雖然它們有同樣的來源，但各自長大之後要合併就不會這麼順利了（想想看要你把你的家產跟你哥哥或姐姐的家產合併在一起...）。在這個情況下，Git 會產生一個額外的 Commit 來處理這件事。一般的 Commit 只會指向某一個 Commit，但這個 Commit 會指向二個 Commit，明確的標記是來自哪兩個分支，親兄弟也是要明算帳啊！

來看看會怎麼演變。假設我想用 `cat` 分支來合併 `dog` 分支：

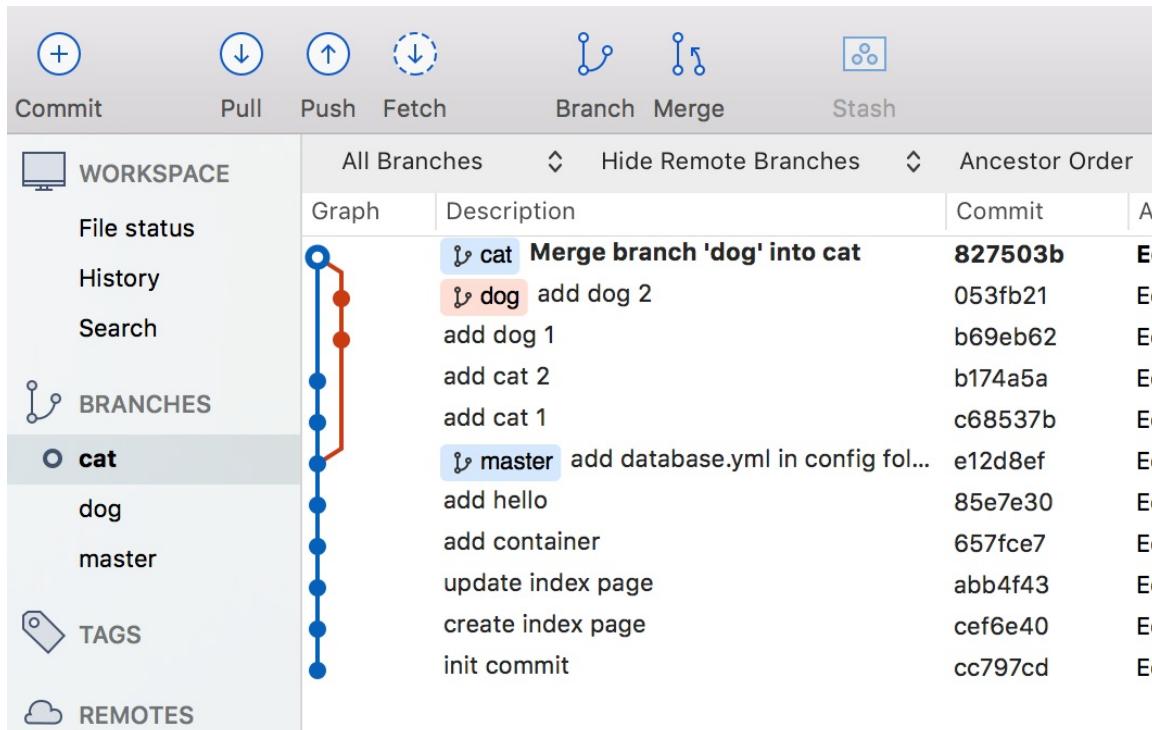
```

$ git merge dog
Merge made by the 'recursive' strategy.
  dog1.html | 0
  dog2.html | 0
  2 files changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 dog1.html
  create mode 100644 dog2.html
  
```

執行這個指令的時候會跳出一個 Vim 編輯器視窗，如果忘記 Vim 怎麼操作，請再回顧一下「[超精簡 Vim 操作介紹](#)」章節。為了要進行這次的合併，Git 做出了這個額外的 Commit 物件，這個 Commit 會分別指向 `cat` 跟 `dog` 這兩個分支，HEAD 隨著 `cat` 分支往前，而 `dog` 分支停留在原地：



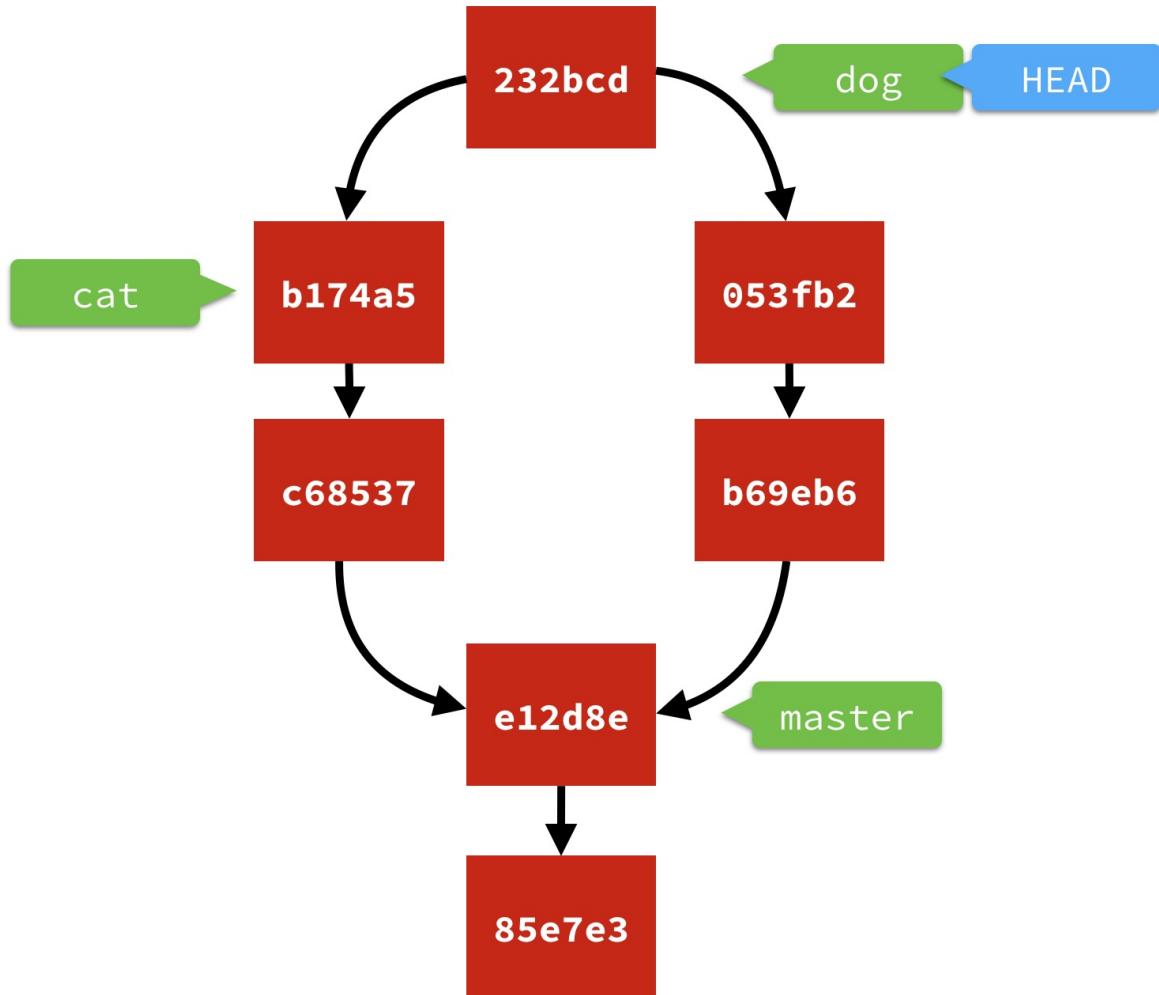
如果用 SourceTree 來看會像這樣：



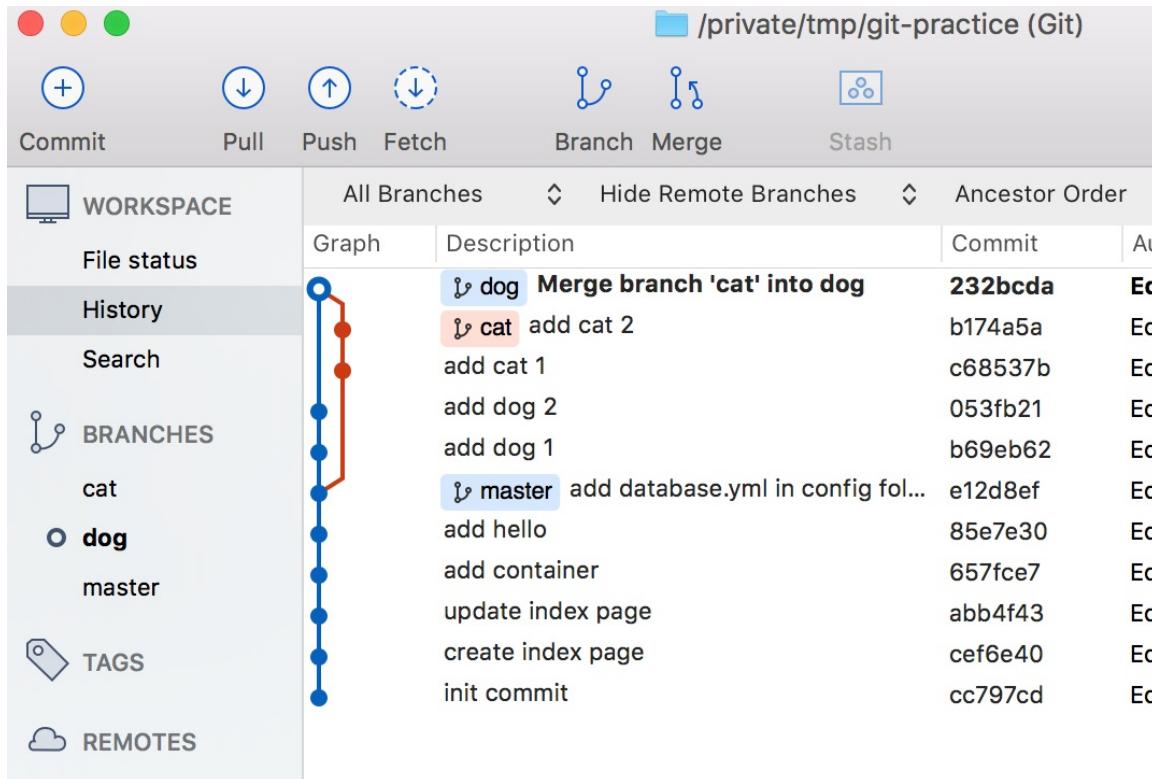
如果改由 `dog` 分支來合併 `cat` 分支：

```
$ git merge cat
Merge made by the 'recursive' strategy.
 cat1.html | 0
 cat2.html | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat1.html
 create mode 100644 cat2.html
```

流程上跟剛才幾乎是一樣的。這時候的狀態會變成這樣：



如果用 SourceTree 來看會像這樣：



哪裡不一樣？

你有看出哪裡不一樣嗎？其實就以結果來看，不管是誰合併誰，這兩個分支的檔案最後都拿到了。你可能看 SourceTree 的畫面，會認為誰合併誰會有誰在前面、誰在後面的差別，但事實上並不是這樣，那只是因為軟體沒辦法畫出來「平行」的效果而已。

事實上不管誰合併誰，這兩個分支上的 Commit 都是對等的。硬是要說哪裡不一樣，就是 `cat` 分支合併 `dog` 分支的時候，`cat` 分支會往前移動，反之亦然。不過前面曾經提到分支就像貼紙一樣，隨時要刪掉或改名都不會影響現在已經存在的 Commit。

有啦，真的不一樣的還有一點，就是這個為了合併而產生的這個額外的 Commit 物件，裡面會記錄兩個老爸是誰，誰合併誰就會有「誰放前面」的差別，不過這可能就有點太過細節了。

這是 `cat` 分支合併 `dog` 分支，所以 `cat` 分支 `b174a5a95a` 放前面：



Merge branch 'dog' into cat

```
Commit: f74353935bb31b984d27b966065215343c3f44dd [f743539]
Parents: b174a5a95a, 053fb212bb
Author: Eddie Kao <eddie@5xruby.tw>
Date: 2017年8月18日 GMT+8 14:28:59
Labels: HEAD -> cat
```

這是 `dog` 分支合併 `cat` 分支，所以 `dog` 分支 `053fb212bb` 放前面：



Merge branch 'cat' into dog

Commit: 232bcda133ccb037f9a48c17200d287e2826a2b5 [232bcda]

Parents: [053fb212bb](#), [b174a5a95a](#)

Author: Eddie Kao <eddie@5xruby.tw>

Date: 2017年8月18日 GMT+8 14:02:35

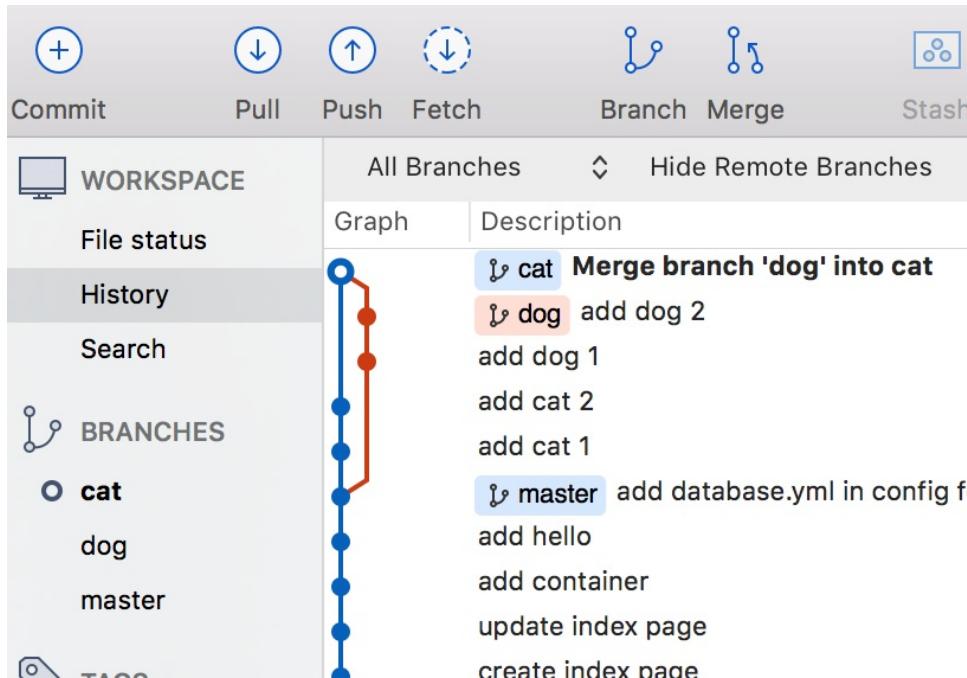
Labels: HEAD -> dog

這很重要嗎？國外曾經有一個 Ruby 跟 Python 一起合辦的研討會名字叫做 [RuPy](#)（後來已改名成 PolyConf），那為什麼 Ruby 要放前面？Python 的人可能會想為什麼不叫 PyRu？這大概就跟 A 合併 B 跟 A 合併 B 的差別差不多吧。

【狀況題】為什麼我的分支都沒有「小耳朵」？

我看別人的合併都會有小耳朵耶，為什麼我的都沒有？

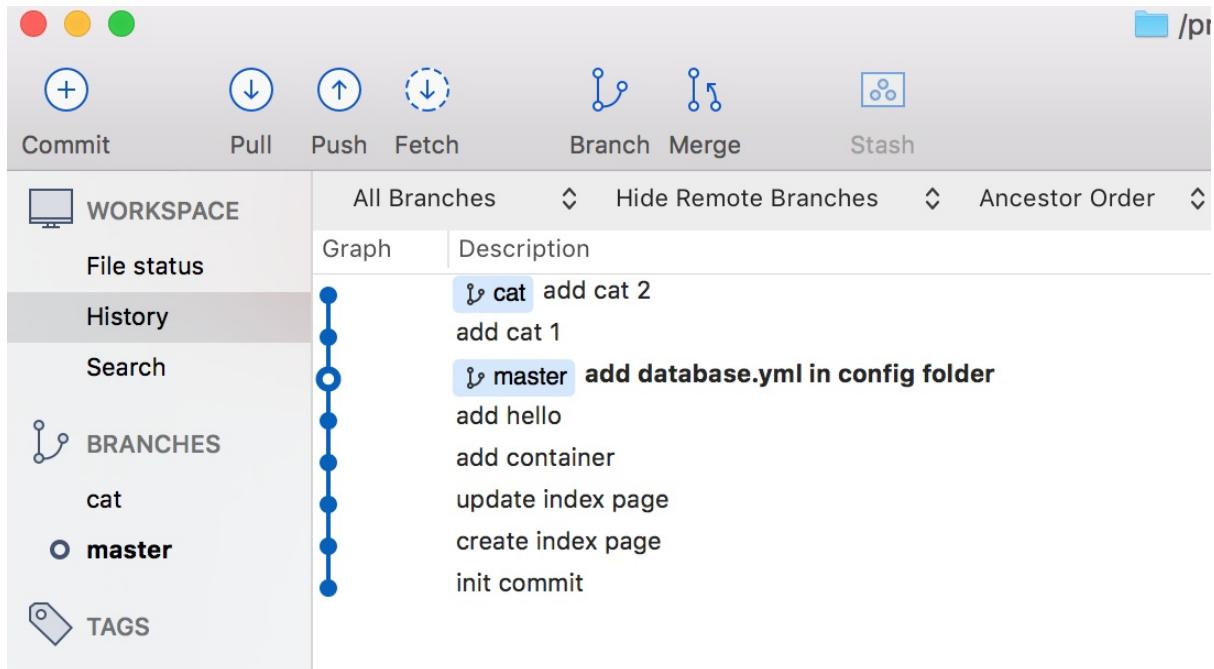
這裡指的「小耳朵」是指在合併的時候產生的線圖：



不是每次的合併都要這麼複雜的...

以上圖這個例子來說，其實會有這個線圖也是不得已的，`cat` 分支跟 `dog` 分支雖是本是同根生，但後來已各自長大、分家了，所以最後要合併回來的時候，Git 就會做出一個額外的 Commit 記錄來自是來自哪兩個 Commit。

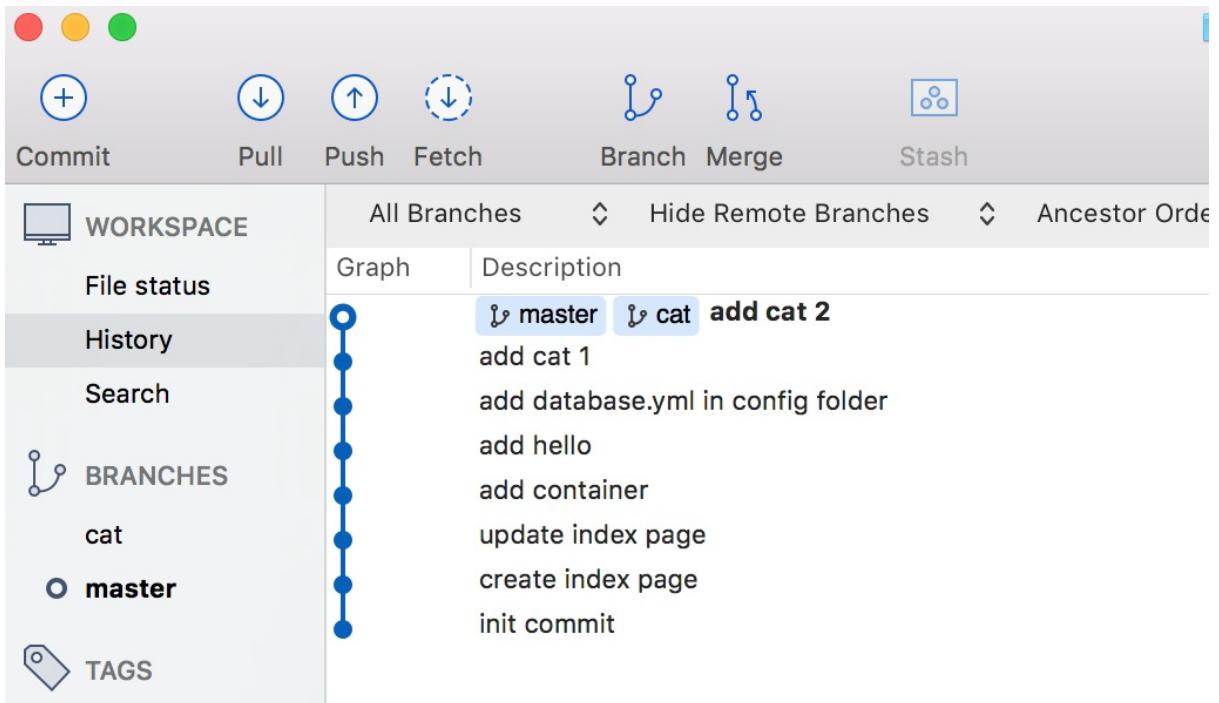
再看看這個例子：



`cat` 分支是從 `master` 分支切出去的，目前領先 `master` 兩次 Commit。如果這時候回到 `master` 並且合併 `cat` 分支，Git 會發現「其實你是從我這邊出去的啊，除了這兩個新的 Commit 之外的東西我都有了」，所以 Git 就會自動選用「快轉模式（Fast Forward）」來進行合併：

```
$ git merge cat
Updating e12d8ef..b174a5a
Fast-forward
 cat1.html | 0
 cat2.html | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat1.html
 create mode 100644 cat2.html
```

你有注意到上面這段訊息裡也有提到「Fast-forward」字樣嗎？這個所謂的「快轉」模式合併，其實就是把 `master` 這張貼紙撕起來，然後往前貼到 `cat` 分支所指的那個 Commit 而已：



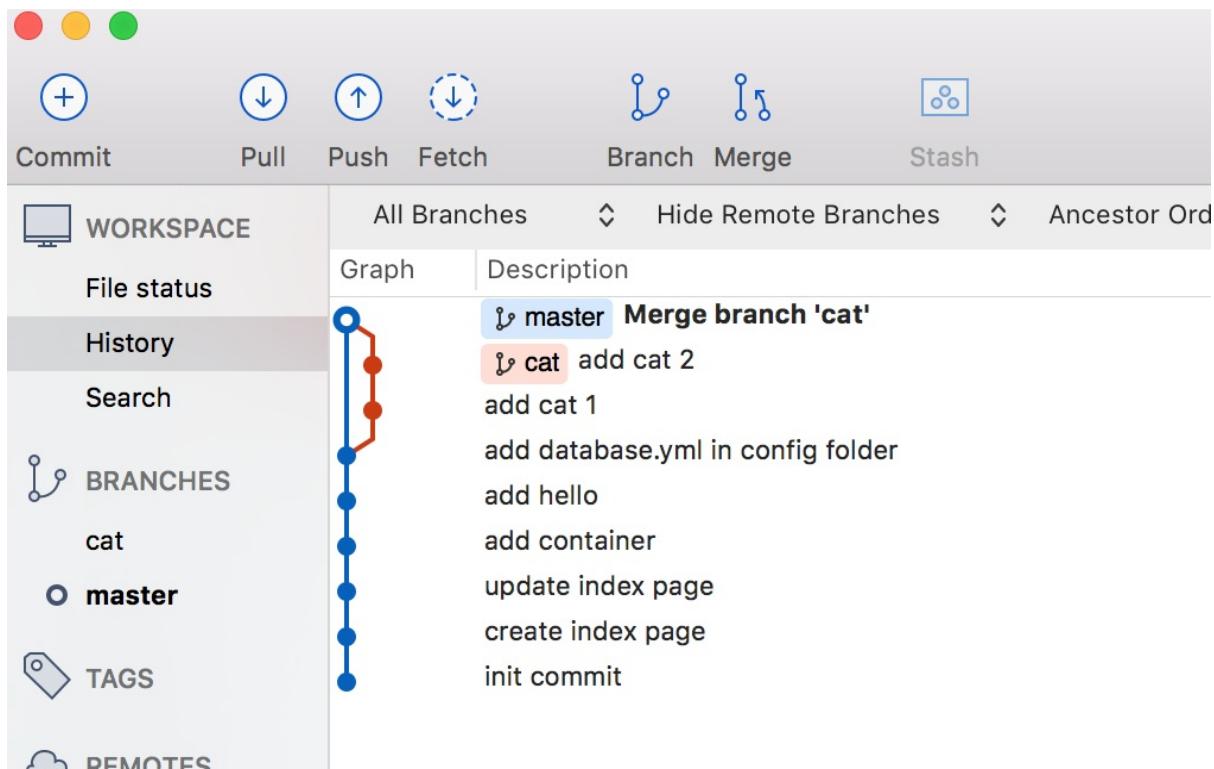
這種快轉模式的合併不會有額外的線圖出現（也就是小耳朵）。

不管，就是要！

硬是要有線圖也是可以的，只要在合併的時候，加上 `--no-ff` 參數：

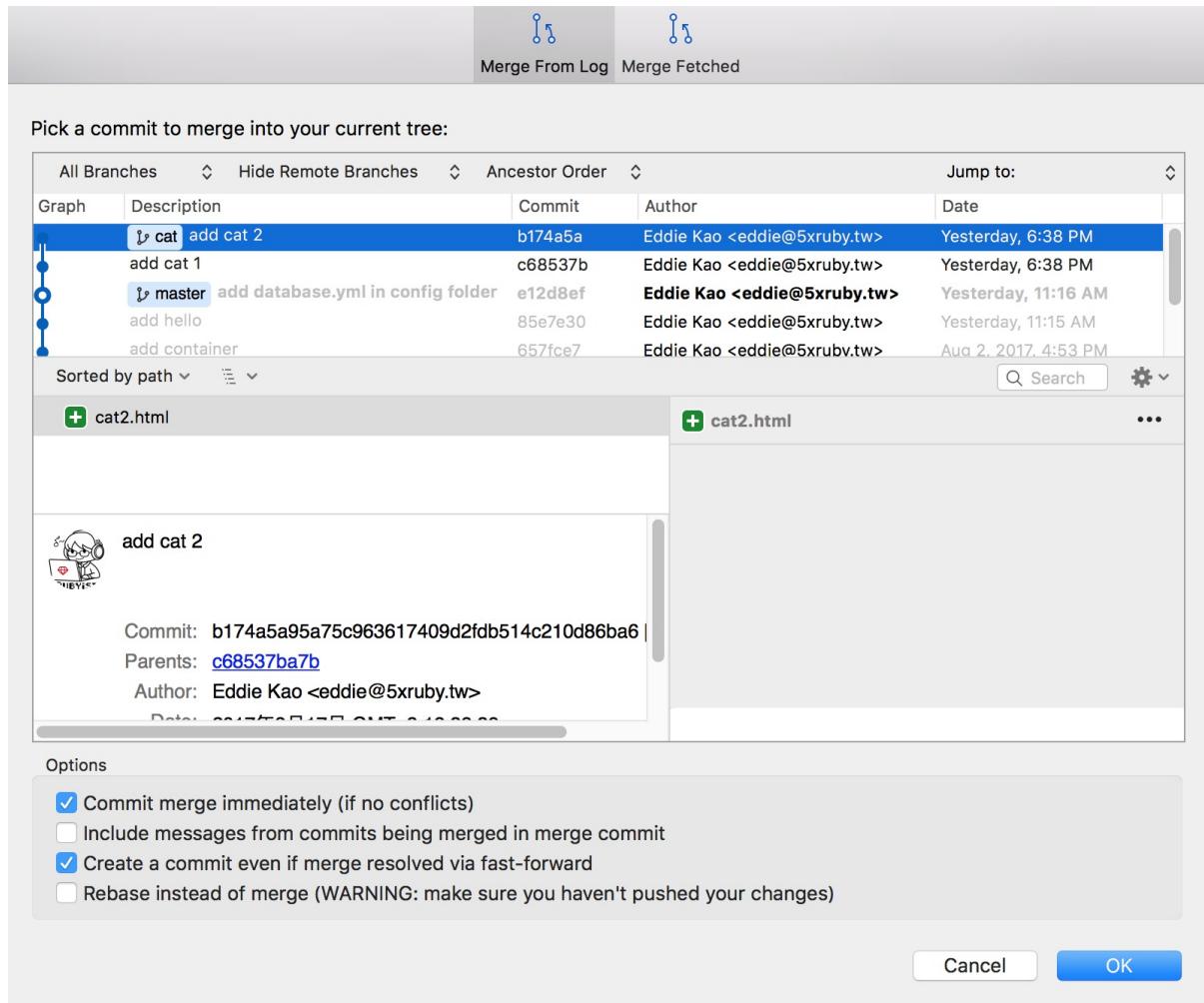
```
$ git merge cat --no-ff
Merge made by the 'recursive' strategy.
 cat1.html | 0
 cat2.html | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat1.html
 create mode 100644 cat2.html
```

`--no-ff` 參數是「不要使用快轉模式合併」的意思，這樣一來就會額外做出一個 Commit 物件：



這個做出來的 Commit 物件一樣會指向前面的兩個 Commit。

如果使用 SourceTree 要做這件事，請選擊上面的「Merge」按鈕，在跳出來的這個對話框中，勾選「Create a commit even if merge resolved via fast-forward」選項，就可以做出跟 `--no-ff` 的效果了



有必要嗎？

那到底這個東西到底需要嗎？硬是要做出這個小耳朵有意義嗎？非快轉模式合併的好處是可以完整保留分支的樣子，但如果你不是很介意這件事的話其實也就沒必要硬要做出小耳朵囉。

【常見問題】合併過的分支要留著嗎？

先說答案：「都可以，看你心情」

首先，你需要先知道什麼是一個「分支」，可以參閱「[【冷知識】為什麼大家都說在 Git 開分支「很便宜」？](#)」章節說明。

基本上，所謂的分支就只是一個有 40 個字元的檔案而已，而這個分支，也就是這 40 個字元，會標記出它目前指向哪一個 Commit。而該分支只要經過合併，不管是上個章節提到的快轉模式（Fast Forward）還是非快轉模式，合併過就代表「這些內容本來只有你有，現在我也有了」。

既然合併過之後，原本沒有的內容我都有了，分支本身又像一張貼紙一樣沒有舉足輕重的地位，老實說它已經沒有利用價值了，可能只剩下挑燈夜戰、加班趕工的時候所留下的一點革命情感吧。所以，合併過的分支想刪就刪吧，刪除分支這個動作就只是把一張貼紙撕起來而已，原來被這張貼紙貼著的東西並不會因此而不見。

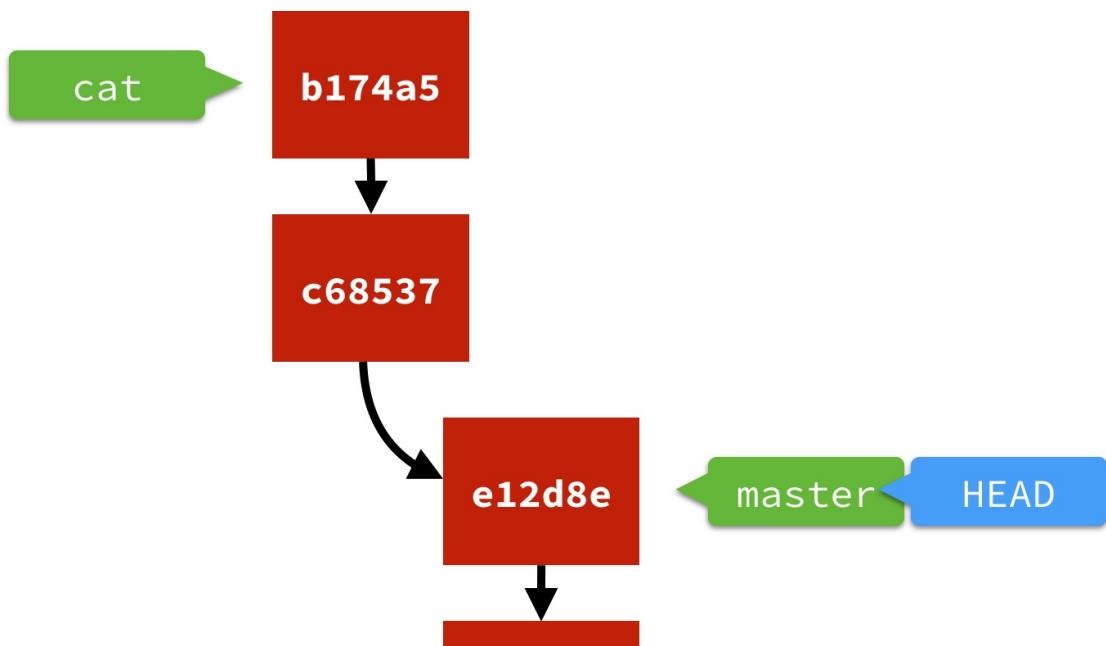
當然，如果還沒合併過的分支就是另一回事了。

回到原本的問題「合併過的分支要留著嗎？」都可以，你想要刪掉，或是已經跟這個分支建立感情了，想留著做紀念也可，都好。

【狀況題】不小心把還沒合併的分支砍掉了，救得回來嗎？

上個章節「【常見問題】合併過的分支要留著嗎？」提到合併過的分支如果想留就留、想刪就刪，Git 的分支並不是複製檔案到某個目錄，所以不會因為刪掉分支檔案就不見。

但如果刪的是還沒合併的分支就不一樣了，先想像一下這個畫面：



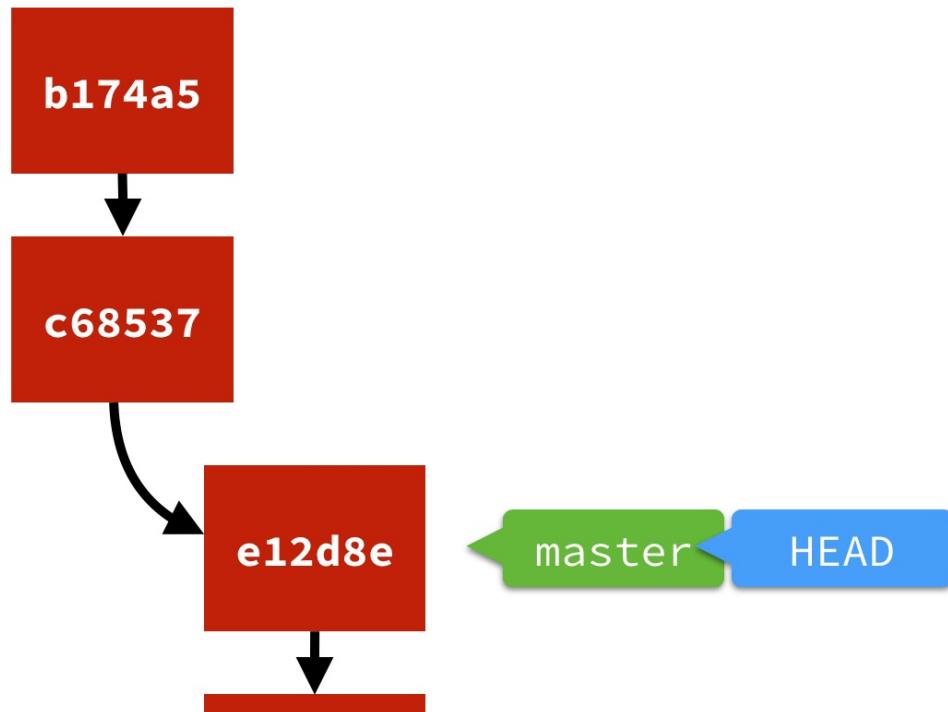
`cat` 分支是從 `master` 分支出去的，目前領先 `master` 分支兩次 Commit，而且也還沒有合併。這時候如果試著刪掉 `cat` 分支，它會提醒你：

```
$ git branch -d cat
error: The branch 'cat' is not fully merged.
If you are sure you want to delete it, run 'git branch -D cat'.
```

「嘿！這個分支還沒全部合併喔」，雖然 Git 這麼貼心的提醒你，你還是依舊把它砍了：

```
$ git branch -D cat
Deleted branch cat (was b174a5a).
```

請先記一下這個訊息「Deleted branch cat (was b174a5a).」，待會可能會用到它。這時候的樣子會像這樣：



等等，我不是砍掉 `cat` 分支了嗎？怎麼東西還在？再次跟大家說明一下分支的觀念：

分支只是一個指向某個 Commit 的指標，刪除這個指標並不會造成那些 Commit 消失。

所以，刪掉分支，那些 Commit 還是在，只是因為你可能不知道或沒記下那些 Commit 的 SHA-1 值，所以不容易再拿來利用。現在原本領先 `master` 分支的那兩個 Commit 就跟空氣一樣，你看不到空氣，但空氣是存在的。既然它還存在，那就把它「接回來」吧：

```
$ git branch new_cat b174a5a
```

這個指令的意思是「請幫我建立一個叫做 `new_cat` 的分支，讓它指向 `b174a5a` 這個 Commit」，就是再去拿一張新的貼紙貼回去的意思啦。看一下現在的分支：

```
$ git branch
* master
  new_cat
```

切換過去試試看：

```
$ git checkout new_cat
Switched to branch 'new_cat'
```

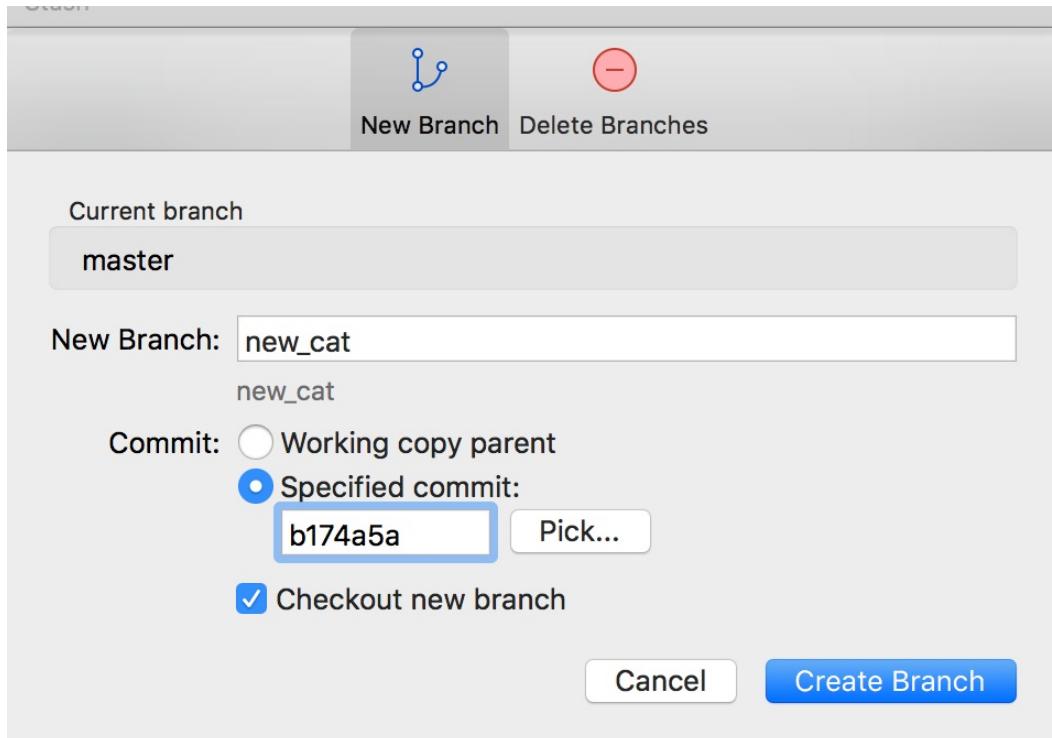
確認一下檔案列表：

```
$ ls -al
total 16
```

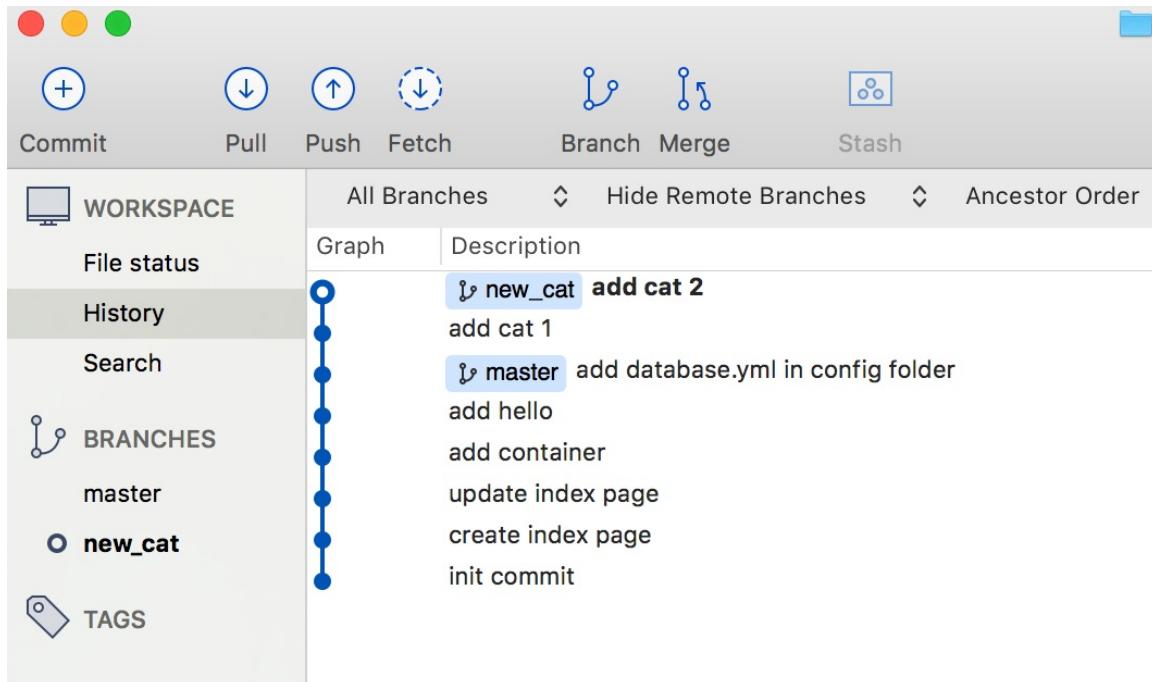
```
drwxr-xr-x  9 eddie  wheel  306 Aug 19 04:14 .
drwxrwxrwt 95 root   wheel  3230 Aug 19 04:06 ..
drwxr-xr-x 16 eddie  wheel  544 Aug 19 04:14 .git
-rw-r--r--  1 eddie  wheel     0 Aug 19 04:14 cat1.html
-rw-r--r--  1 eddie  wheel     0 Aug 19 04:14 cat2.html
drwxr-xr-x  3 eddie  wheel  102 Aug 17 15:06 config
-rw-r--r--  1 eddie  wheel     0 Aug 18 03:27 hello.html
-rw-r--r--  1 eddie  wheel  161 Aug 18 04:24 index.html
-rw-r--r--  1 eddie  wheel    11 Aug 17 14:56 welcome.html
```

cat1.html 跟 cat2.html 都回來了！

使用 SourceTree 也可以做這件事，點擊上面選單的「Branch」按鈕後會看到這個對話框：



在「New Branch」欄位填上「new_cat」或是任何你喜歡的名字，但 Commit 選擇「Specified commit」，並把原本舊的 cat 分支指向的 Commit 的 SHA-1 值 b174a5a 填上去，按下「Create Branch」後：



原本被砍掉的 `cat` 分支，就以 `new_cat` 的姿態轉生復活了！

我沒把剛剛刪掉的那個 `cat` 分支的 SHA-1 記下來怎麼辦？查得到嗎？

還是可以的，你可以從 `git reflog` 指令去翻翻看，Reflog 預設會保留 30 天，所以 30 天內應該都還找得到。Reflog 的使用方式，可參閱「[【狀況題】不小心使用 hard 模式 Reset 了某個 Commit，救得回來嗎？](#)」章節介紹。

【觀念】其實所謂的「合併分支」，合併的並不是「分支」…

如果你對 Git 的分支的觀念是正確的，你應該可以猜到知道下面這個指令在做什麼：

```
$ git merge b174a5a
Updating e12d8ef..b174a5a
Fast-forward
 cat1.html | 0
 cat2.html | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat1.html
 create mode 100644 cat2.html
```

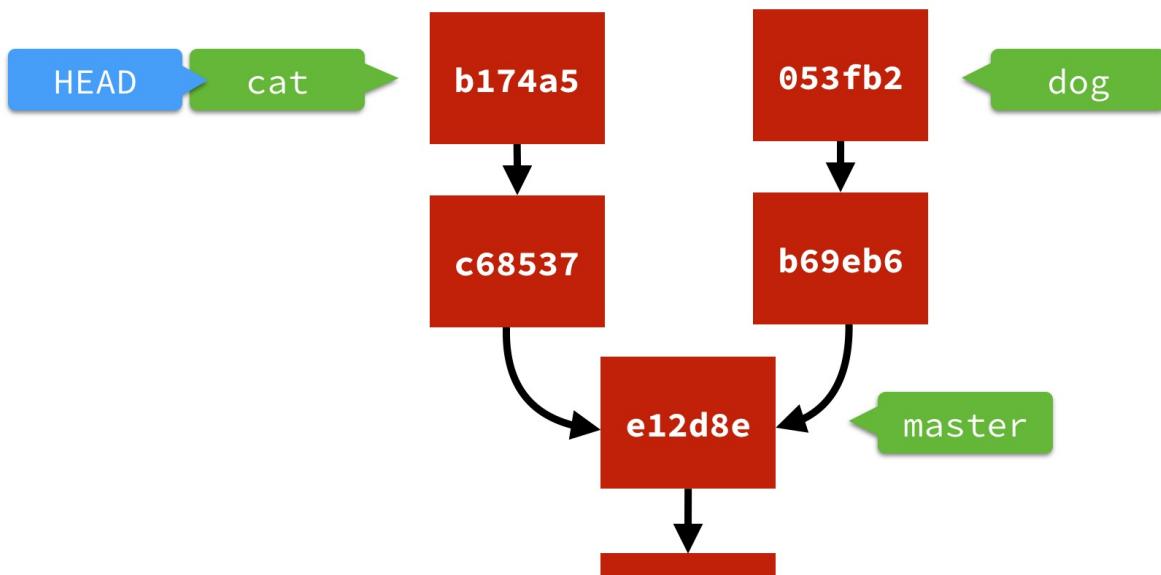
嘍？這訊息不是在合併分支的時候跳出來的訊息嗎？是的，所謂的「合併分支」，其實是合併「分支指向的那個 Commit」。分支只是一張貼紙，它是沒辦法被合併的，只是我們會用「合併分支」這個說法，畢竟它比較「合併 Commit」來得容易想像。

重要觀念！

分支只是一個指向某個 Commit 的指標。

另一種合併方式（使用 rebase）

前面介紹了使用 `git merge` 指令來合併分支，接下來介紹另一種合併分支的方式。假設我們現在的狀態是這樣：



有 `cat` 、`dog` 以及 `master` 這三個分支，並且切換至 `cat` 分支上。這時候如果下這個指令：

```
$ git merge dog
```

則會產生一個額外的 Commit 來接合兩邊分支，這就是我們在「合併分支」章節曾經介紹過的方式。

Git 有另一個指令叫做 `git rebase`，也可以用來做跟 `git merge` 類似的事情。

從字面上來看，「rebase」是「re」加上「base」，翻成中文大概是「重新定義分支的參考基準」的意思。

所謂「base」就是指「你這分支是從哪裡生出來的」，以上面這個例子來說，`cat` 跟 `dog` 這兩個分支的 base 都是 `master`。接著我們試著使用 `git rebase` 指令來「組合」`cat` 跟 `dog` 這兩個分支：

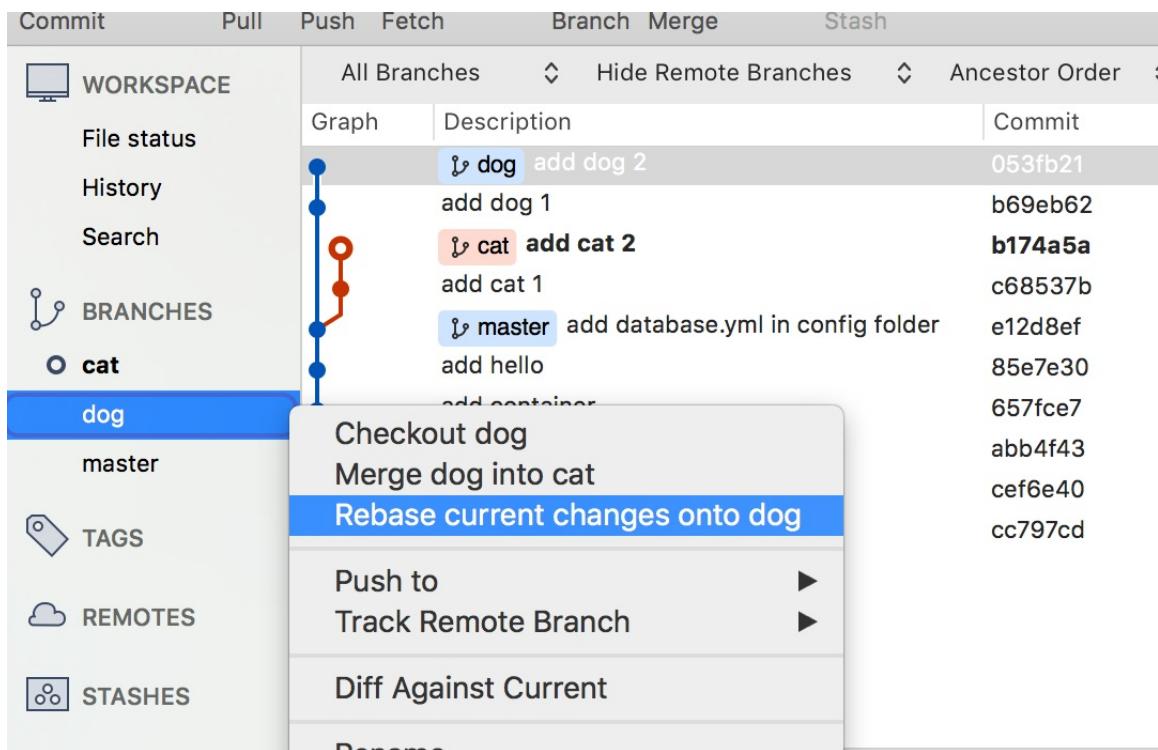
```
$ git rebase dog
```

這個指令翻成白話文，大概就是「我，就是 `cat` 分支，我現在要重新定義我的參考基準，並且將使用 `dog` 分支當做我新的參考基準」的意思。這個指令執行的訊息如下：

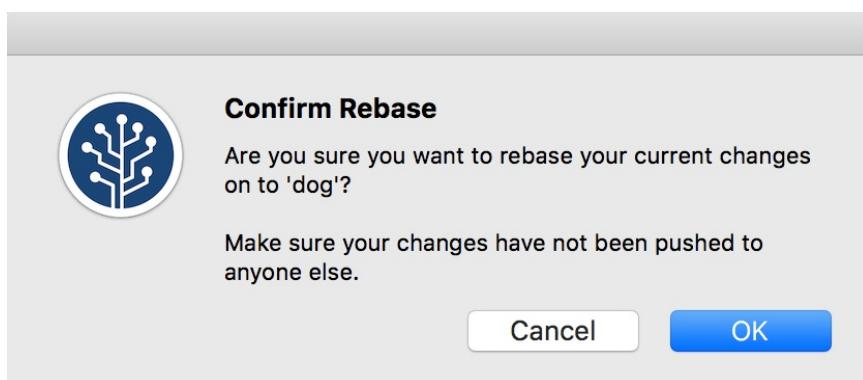
```
$ git rebase dog
```

```
First, rewinding head to replay your work on top of it...
Applying: add cat 1
Applying: add cat 2
```

如果要使用 SourceTree 來進行 Rebase，可在左邊選單找到想要 Rebase 的對象，按滑鼠右鍵並選擇「Rebase current branch onto dog」：

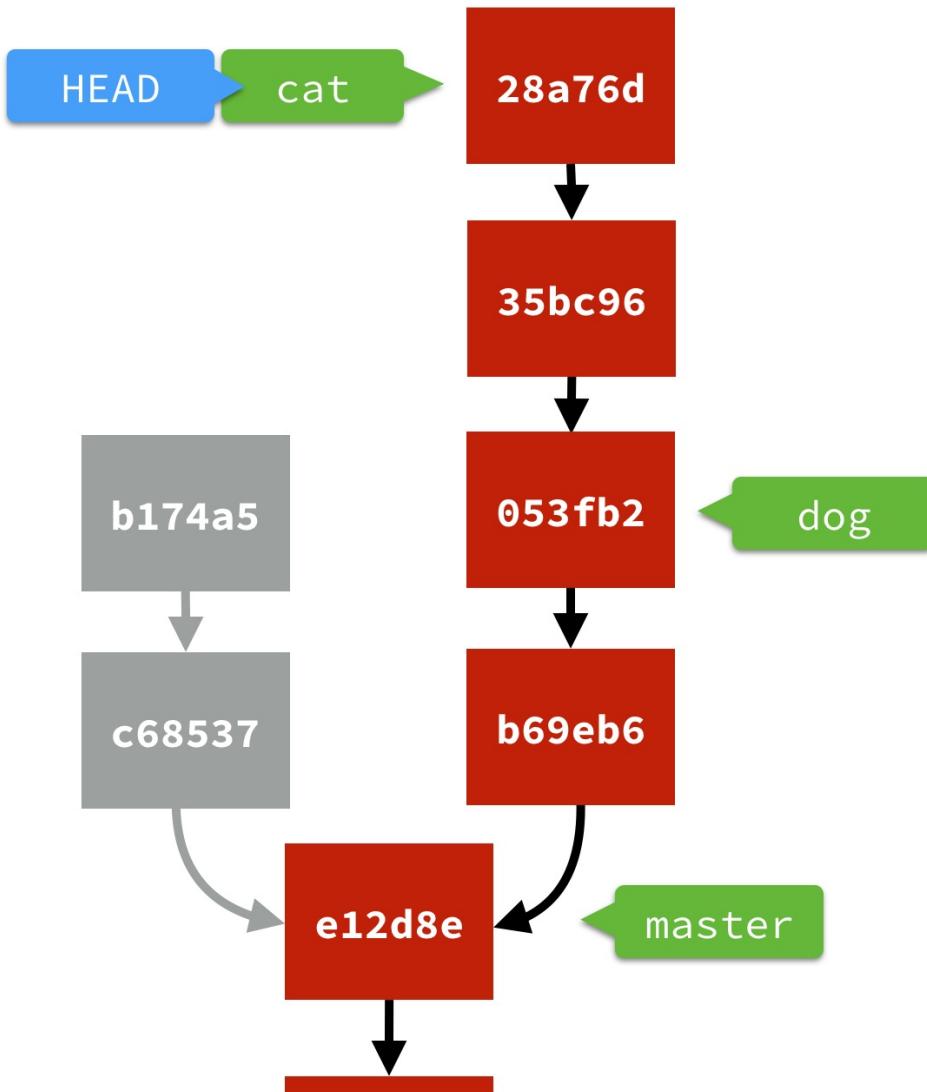


它會跳出一個對話框：

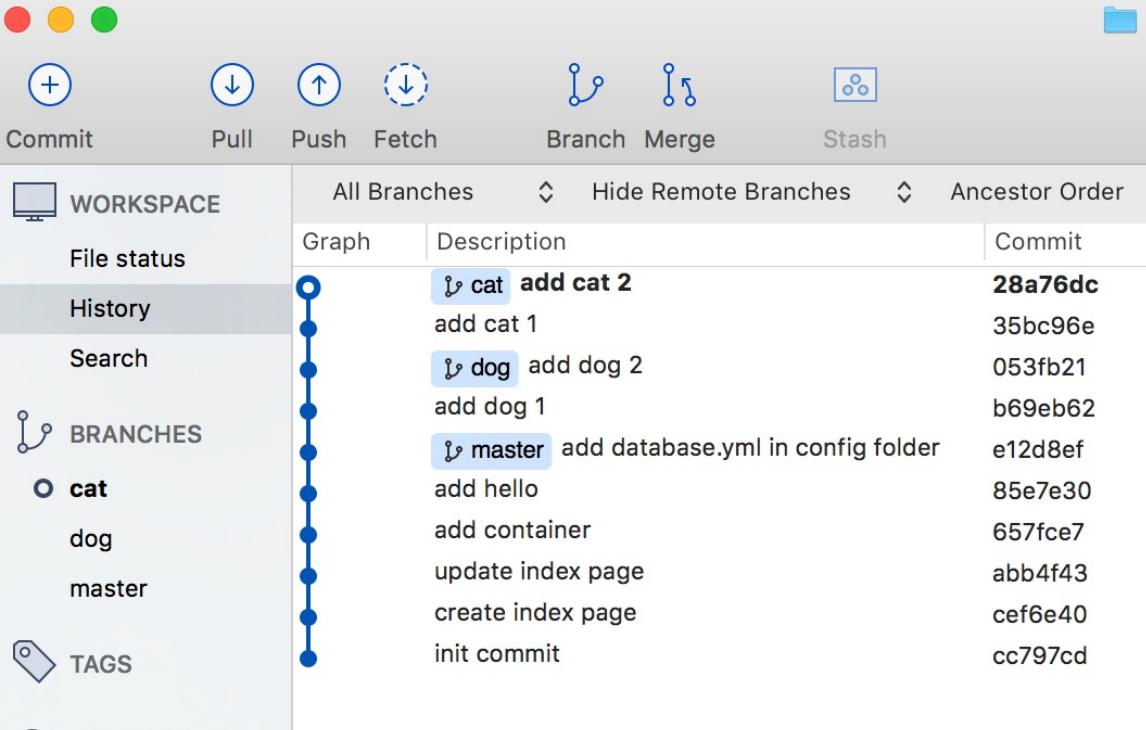


上面還寫著「Make sure your changes have not been pushed to anyone else」的貼心小提示，因為 Rebase 指令等於是修改歷史，不應該隨便對已經推出去給別人內容進行 rebase，因為這很容易造成其它人的困擾。這方面的「困擾」，可參閱「修改歷史紀錄」相關章節的介紹。

不管如何，按下 OK 鈕便會完成。完成之後，`cat` 分支將會接到 `dog` 分支上，像這樣：



如果使用 SourceTree 觀看歷史紀錄：



The screenshot shows a Git interface with a 'History' tab selected. On the left, there's a sidebar with 'WORKSPACE', 'BRANCHES' (containing 'cat', 'dog', 'master'), and 'TAGS'. The main area displays a timeline of commits. A vertical blue line indicates the rebase operation, where the 'cat' branch has been moved onto the 'dog' branch. The commits are listed as follows:

Commit	Description	Commit
28a76dc	↳ cat add cat 2	
35bc96e	add cat 1	
053fb21	↳ dog add dog 2	
b69eb62	add dog 1	
e12d8ef	↳ master add database.yml in config folder	
85e7e30	add hello	
657fce7	add container	
abb4f43	update index page	
cef6e40	create index page	
cc797cd	init commit	

Rebase 合併分支跟一般的合併分支，第一個很明顯的差別，就是使用 Rebase 方式合併分支的話，Git 不會特別做出一個專門用來合併的 Commit。

是剪下、貼上嗎？

就以結果來看，感覺像是「把 `cat` 分支剪下來，然後貼在 `dog` 分支上面」，有點像插花時候「嫁接」的概念：

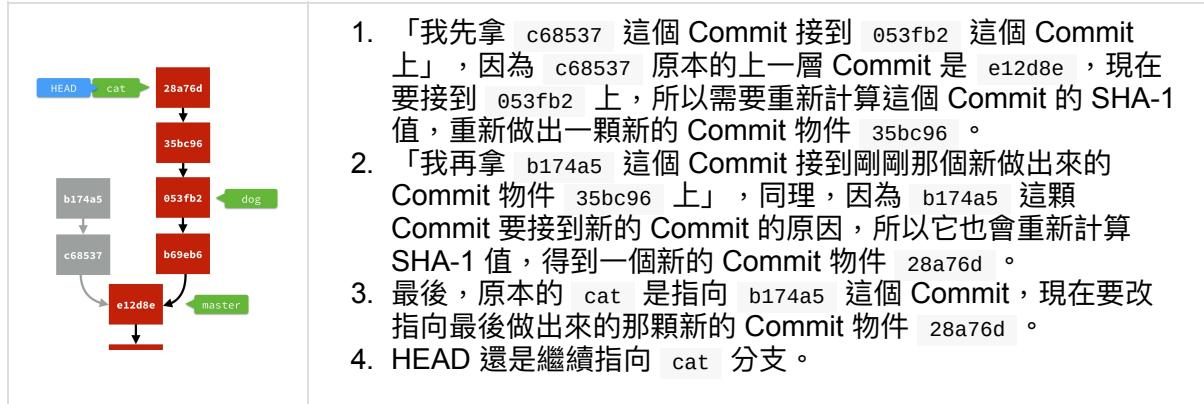


photo by [UGA College of Ag & Environmental Sciences](#)

但其實不太一樣，Rebase 不是「剪下、貼上」這麼單純。有注意到剛剛在 rebase 的時候的那段訊息嗎？

```
$ git rebase dog
First, rewinding head to replay your work on top of it...
Applying: add cat 1
Applying: add cat 2
```

以我們這個例子來說，Rebase 的過程大概是這樣（請搭配下圖服用）：



所以，在 Rebase 的過程中你會看到 2 次的 "Applying" 的字樣，就是在做「重新計算」的事情。

那原本舊的那些...？

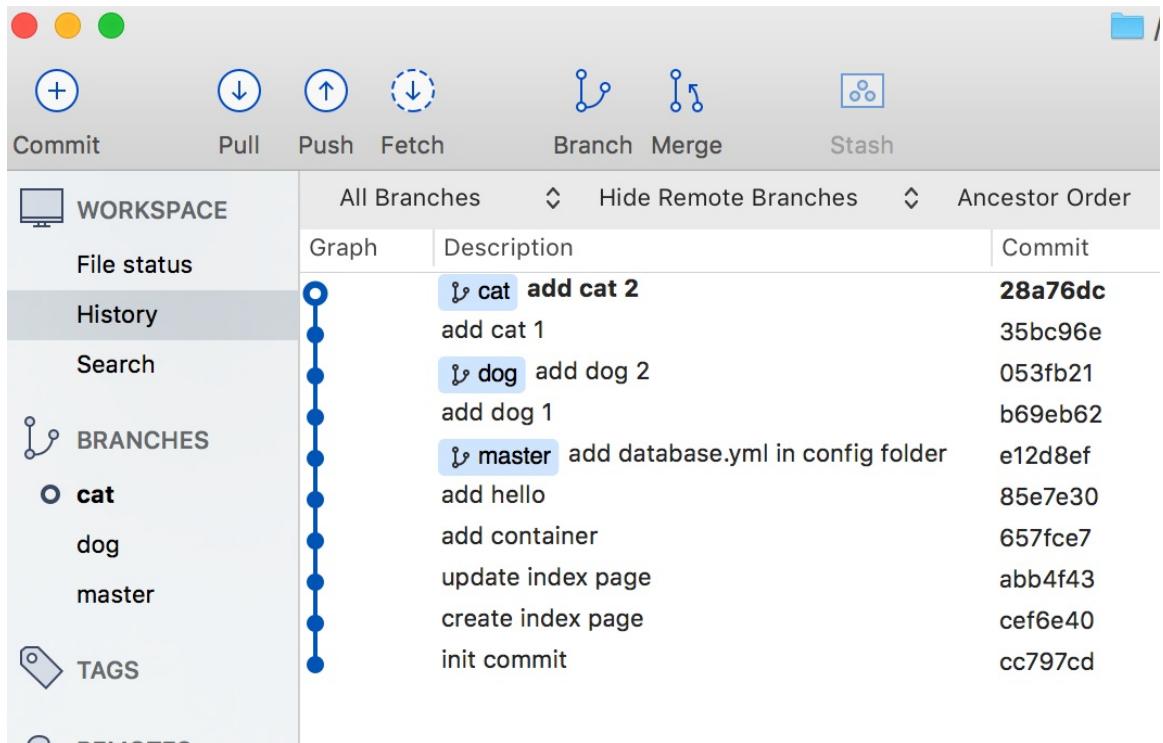
在上圖中，原本的那兩個 Commit（灰色），也就是 `c68537` 跟 `b174a5` 這兩個，他們的下場會是怎麼樣？

是也不會怎麼樣，反正他們就還是在 Git 的空間裡佔有一席之地，只是因為它已經沒有分支指著它，如果沒有特別去記他們這兩個 Commit 的 SHA-1 值，就會慢慢被邊緣化了吧。

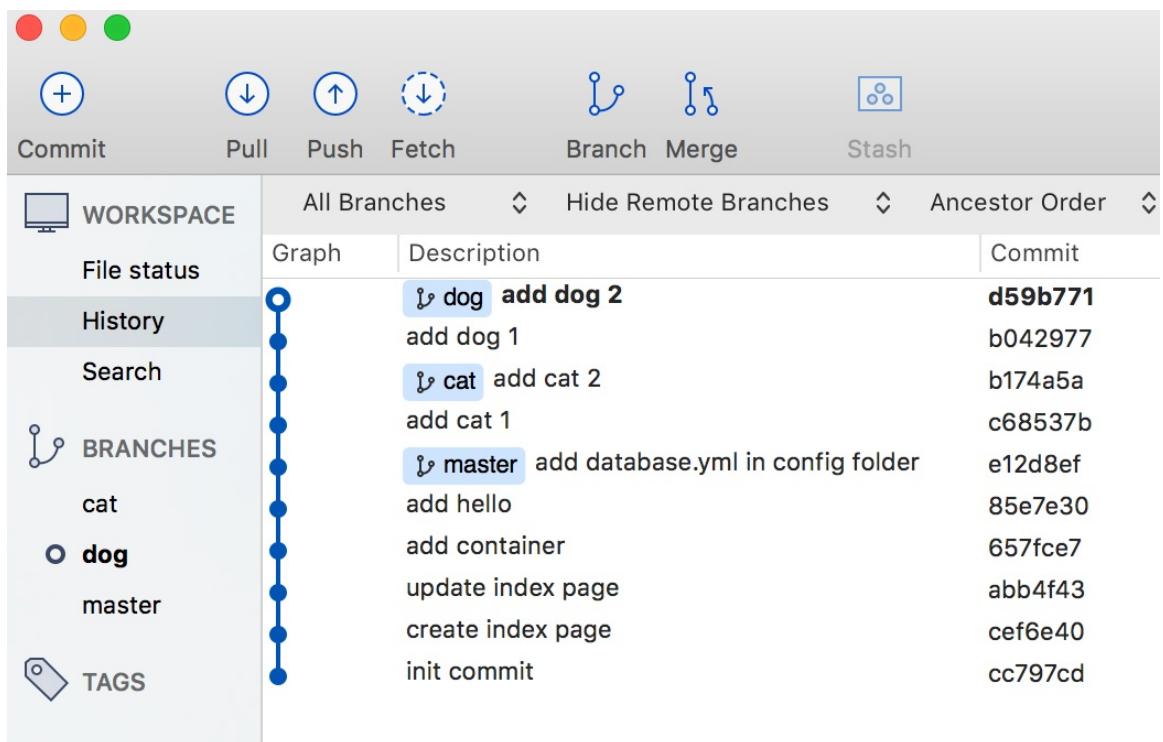
但，他們並沒有馬上被刪除喔，他們只是默默的待在那邊，直到有一天被 Git 的資源回收車載走。關於 Git 的資源回收機制，請參閱「[【冷知識】你知道 Git 有資源回收機制嗎？](#)」章節介紹。

誰 Rebase 誰有差嗎？

就以最後的檔案來說是沒什麼差別，但以歷史紀錄來說有差別，誰 Rebase 誰，會造成歷史紀錄上先後順序不同的差別。這是 `cat` 分支 Rebase `dog` 分支：



而這是 `dog` 分支 Rebase `cat` 分支：



這些 Commit 的歷史先後順序就明顯不同了。

【狀況題】怎麼取消 rebase？

如果是一般的合併，也許只要 `git reset HEAD^ --hard` 一行指令，拆掉這個合併的 Commit 大家就會退回到合併前的狀態。但是，從上面的結果可得知，Rebase 並沒有做出那個合併專用的 Commit，而是整串都串在一起了，就跟一般的 Commit 差不多。所以這時候如果執行 `git reset HEAD^ --hard`，只會拆掉最後一個 Commit，但並不會回到 Rebase 前的狀態。

使用 Reflog

第一個方法是使用 Reflog。

沒錯，又是它，其實 Reflog 會紀錄很多好用的東西。舉個例子來說，我剛剛把 `cat` 分支 Rebase 到 `dog` 分支上，所以目前的樣子長這樣：

```
$ git log --oneline
28a76dc (HEAD -> cat) add cat 2
35bc96e add cat 1
053fb21 (dog) add dog 2
b69eb62 add dog 1
e12d8ef (master) add database.yml in config folder
85e7e30 add hello
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

翻一下現在的 Reflog：

```
$ git reflog
28a76dc (HEAD -> cat) HEAD@{0}: rebase finished: returning to refs/heads/cat
28a76dc (HEAD -> cat) HEAD@{1}: rebase: add cat 2
35bc96e HEAD@{2}: rebase: add cat 1
053fb21 (dog) HEAD@{3}: rebase: checkout dog
b174a5a HEAD@{4}: checkout: moving from master to cat
e12d8ef (master) HEAD@{5}: checkout: moving from new_cat to master
b174a5a HEAD@{6}: checkout: moving from master to new_cat
e12d8ef (master) HEAD@{7}: checkout: moving from new_cat to master
b174a5a HEAD@{8}: checkout: moving from master to new_cat
...[略]...
```

看得出來最新的幾次紀錄都是在做 Rebase，但我看到這行：

```
b174a5a HEAD@{4}: checkout: moving from master to cat
```

看起來這應該是在開始做 Rebase 前的最後動作，所以就是它了！我使用 `reset` 指令硬切回去：

```
$ git reset b174a5a --hard
HEAD is now at b174a5a add cat 2
```

這就一來就會回到 Rebase 前的狀態了。

使用 ORIG_HEAD

在 Git 有另一個特別的紀錄點叫做 `ORIG_HEAD`，這個 `ORIG_HEAD` 會記錄「危險操作」之前 `HEAD` 的位置。例如分支合併或是 `Reset` 之類的都算是所謂的「危險操作」。透過這個紀錄點來取消這次 `Rebase` 相對的更簡單：

```
$ git rebase dog
First, rewinding head to replay your work on top of it...
Applying: add cat 1
Applying: add cat 2
```

成功重新計算 2 個 `Commit` 並接到 `dog` 分支上了，這時候可使用這個指令輕鬆的跳回 `Rebase` 前的狀態：

```
$ git reset ORIG_HEAD --hard
HEAD is now at b174a5a add cat 2
```

一切又都回來了！

使用 Rebase 時機？

小結一下，使用 `Rebase` 來合併分支的好處，就是它不像一般合併可能會產生額外的合併專用的 `Commit`，而且歷史順序可以依照誰 `Rebase` 誰而決定；但缺點就是它相對的比一般的合併來得沒那麼直覺，一個不小心可能會弄壞掉而且還不知道怎麼 `Reset` 回來，或是發生衝突的時候就會停在一半，對不熟悉 `Rebase` 的人來說是個困擾。關於發生衝突，將在「[合併發生衝突了，怎麼辦？](#)」章節再另外介紹。

通常在還沒有推（`Push`）出去但感覺得有點亂（或太瑣碎）的 `Commit`，我會先使用 `Rebase` 分支來整理完再推出去。但如同前面提到的，`Rebase` 等於是修改歷史，修改已經推出去的歷史可能會對其它人帶來困擾，所以對於已經推出去的內容，非必要的話請盡量不要使用 `Rebase`。

合併發生衝突了，怎麼辦？

Git 有能力幫忙檢查簡單的衝突，所以並不是改到同一個檔案就一定會發生衝突，但改到同一行就沒辦法了。假設我在 `cat` 分支修改了 `index.html` 的內容如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>首頁</title>
  </head>
  <body>
    <div class="container">
      <div>我是 Cat</div>
    </div>
  </body>
</html>
```

然後在 `dog` 分支剛好也修改了 `index.html`，內容如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>首頁</title>
  </head>
  <body>
    <div class="container">
      <div>我是 Dog</div>
    </div>
  </body>
</html>
```

這時候進行合併，不管是一般的合併或是使用 Rebase 進行合併，都會出現衝突，我們先使用一般的合併：

```
$ git merge dog
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git 發現那個 `index.html` 檔案有問題了，我們先看一下目前的狀態：

```
$ git status
On branch cat
You have unmerged paths.
```

```
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Changes to be committed:

  new file:  dog1.html
  new file:  dog2.html

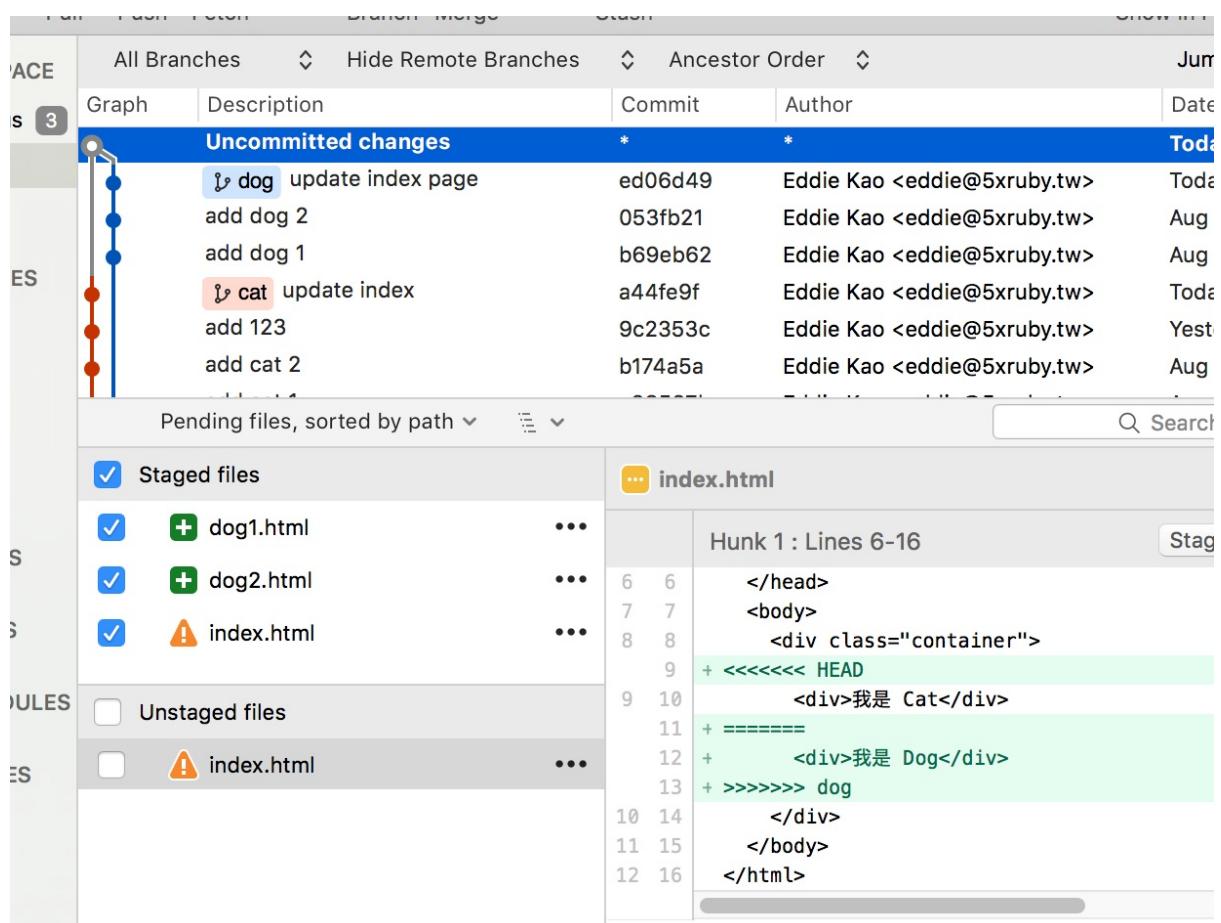
Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified: index.html
```

說明一下：

1. 對 `cat` 分支來說，`dog1.html` 跟 `dog2.html` 是新來的檔案，但已被放置至暫存區。
2. 但是 `index.html` 這個檔案因為兩邊都修改到了，所以 Git 把它標記成「both modified」狀態。

使用 SourceTree 來看：



可以看到那個有衝突的檔案用驚嘆號標記出來了。

解決問題

看看那個 `index.html` 的內容長什麼樣子：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>首頁</title>
  </head>
  <body>
    <div class="container">
<<<<< HEAD
      <div>我是 Cat</div>
=====
      <div>我是 Dog</div>
>>>>> dog
      </div>
    </body>
  </html>
```

Git 把有衝突的段落標記出來了，上半部是 `HEAD`，也就是目前所在的 `cat` 分支，中間是分隔線，接下是 `dog` 分支的內容。

那要解決這個問題？這問題看來是溝通不良造成的，所以遇到問題，當然是解決有問題的人... 不是，是把兩邊的人請過來討論一下，到底是該用誰的 code。經過一番討論後，決定還是要採納 `cat` 分支的內容，順便把那些標記修掉，最後內容如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>首頁</title>
  </head>
  <body>
    <div class="container">
      <div>我是 Cat</div>
    </div>
  </body>
</html>
```

修改完後，別忘了把這個檔案加回暫存區：

```
$ git add index.html
```

然後就可以 Commit，完成這一回合：

```
$ git commit -m "conflict fixed"
[cat a28a93c] conflict fixed
```

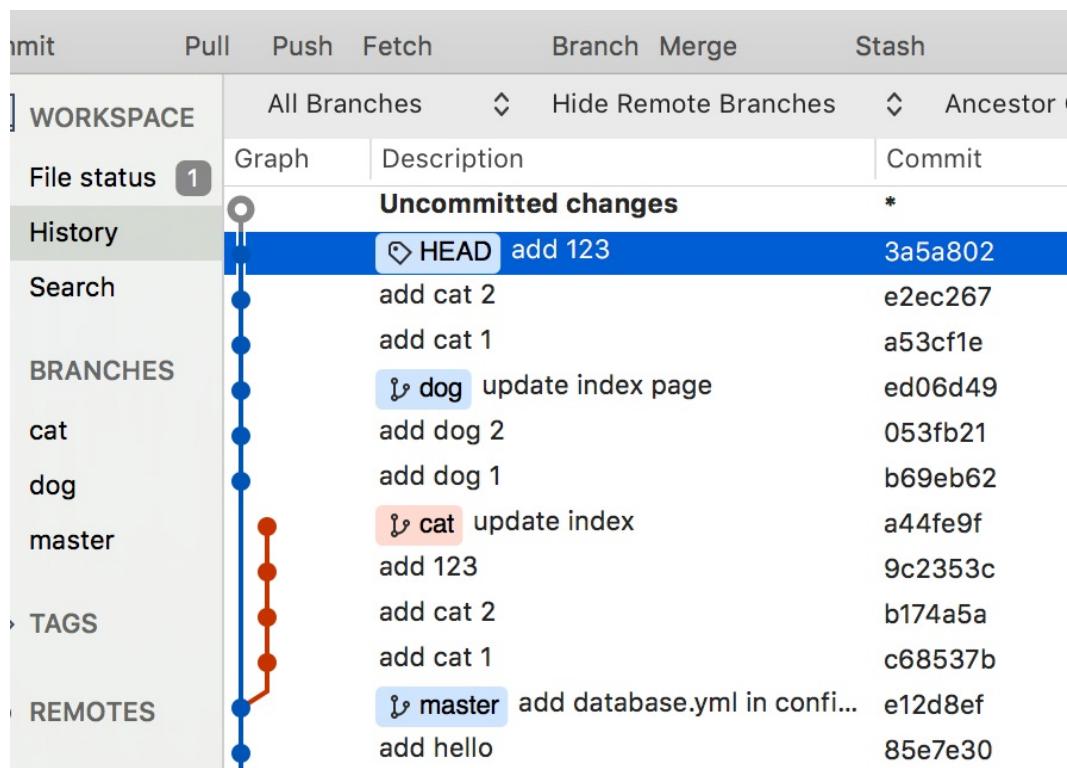
如果是使用 Rebase 的合併造成衝突？

衝突這回事，不管是一般合併或是 Rebase，會打架就是會打架，不會因為合併方式而就不會衝突。但 Rebase 的過程如果發生衝突會跟一般的合併不太一樣。例如：

```
$ git rebase dog
First, rewinding head to replay your work on top of it...
Applying: add cat 1
Applying: add cat 2
Applying: add 123
Applying: update index
Using index info to reconstruct a base tree...
M    index.html
Falling back to patching base and 3-way merge...
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
error: Failed to merge in the changes.
Patch failed at 0004 update index
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

這時候其實是卡在一半，從 SourceTree 可以看得更清楚：



HEAD 現在並沒有指著任何一個分支，它現在有點像是在修改歷史的時候卡在某個時空縫隙裡了（其實是 3a5a802 這個 Commit）。看一下目前的狀態：

```
$ git status
rebase in progress; onto ed06d49
You are currently rebasing branch 'cat' on 'ed06d49'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

    both modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

訊息寫著 `rebase in progress`，而且那個 `index.html` 的確也是被標記成「both modified」狀態。跟上面提到的方法一樣，把 `index.html` 有衝突的內容修正完成後，把它加回暫存區：

```
$ git add index.html
```

搞定，接著繼續完成剛剛中斷的 Rebase：

```
$ git rebase --continue
Applying: update index
```

這樣就算完成 Rebase 了。

那如果不是文字檔的衝突怎麼解？

上面的 `index.html` 因為是文字檔案，所以 Git 可以標記出發生衝突的點在哪些行，我們用肉眼都還能看得出來大概該怎麼解決，但如果是像圖片檔之類的二進位檔怎麼辦？例如在 `cat` 分支跟 `dog` 分支，同時都加了一張叫做 `cute_animal.jpg` 的圖片，合併的時候出現衝突的訊息：

```
$ git merge dog
warning: Cannot merge binary files: cute_animal.jpg (HEAD vs. dog)
Auto-merging cute_animal.jpg
CONFLICT (add/add): Merge conflict in cute_animal.jpg
Automatic merge failed; fix conflicts and then commit the result.
```

這下糟了，要把兩邊的人馬請過來，討論到底誰才是最可愛的動物。討論後決定貓才是這世上最可愛的動物，所以決定要用 `cat` 分支的檔案：

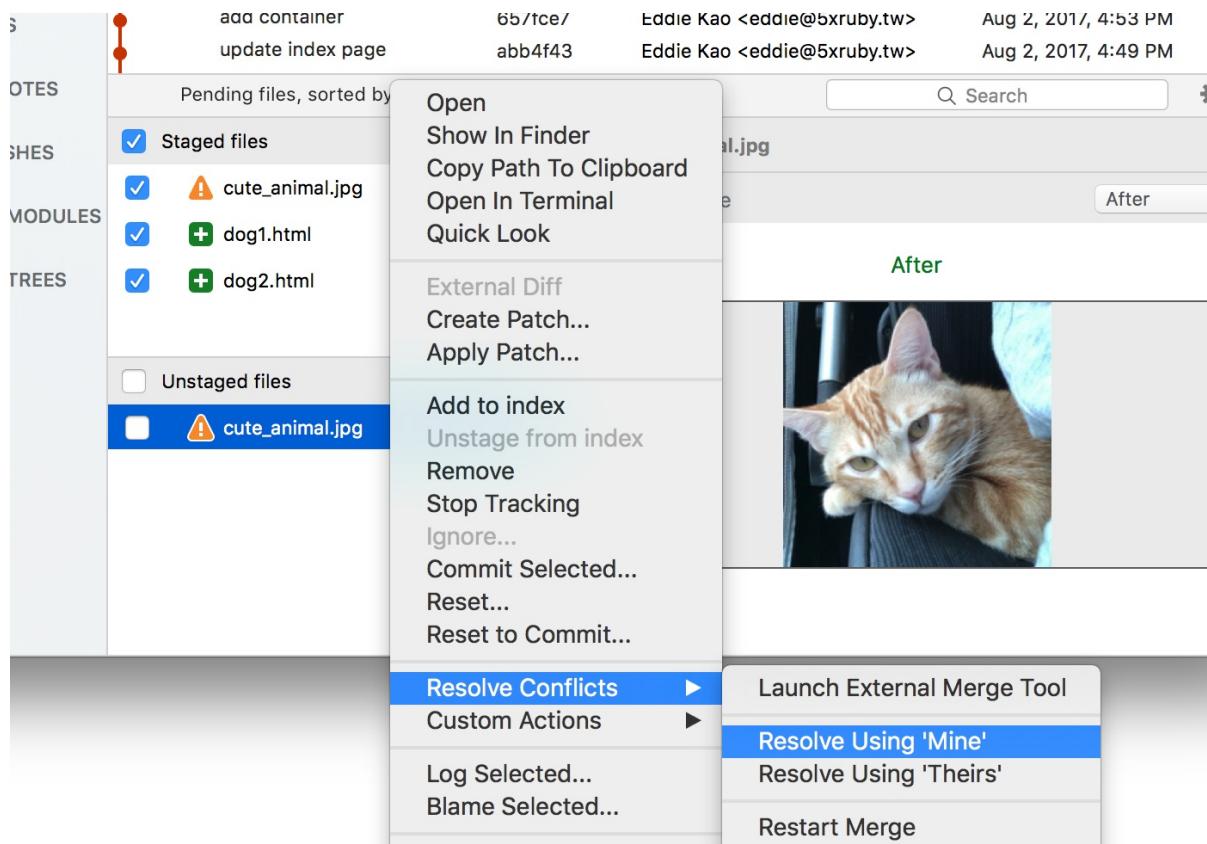
```
$ git checkout --ours cute_animal.jpg
```

如果是要用對方（`dog` 分支），則是使用 `--theirs` 參數：

```
$ git checkout --theirs cute_animal.jpg
```

決定之後，就跟前面一樣，加到暫存區，準備 Commit，然後結束這一回合。

如果使用 SourceTree，可在那個有衝突的檔案上按滑鼠右鍵，選擇「Resolve Conflicts」→「Resolve Using 'Mine'」等同於上面使用 `--ours` 參數的效果：



如果是選擇「Resolve Using 'Theirs'」則等同使用 `--theirs` 參數。

【冷知識】為什麼大家都說在 Git 開分支「很便宜」？

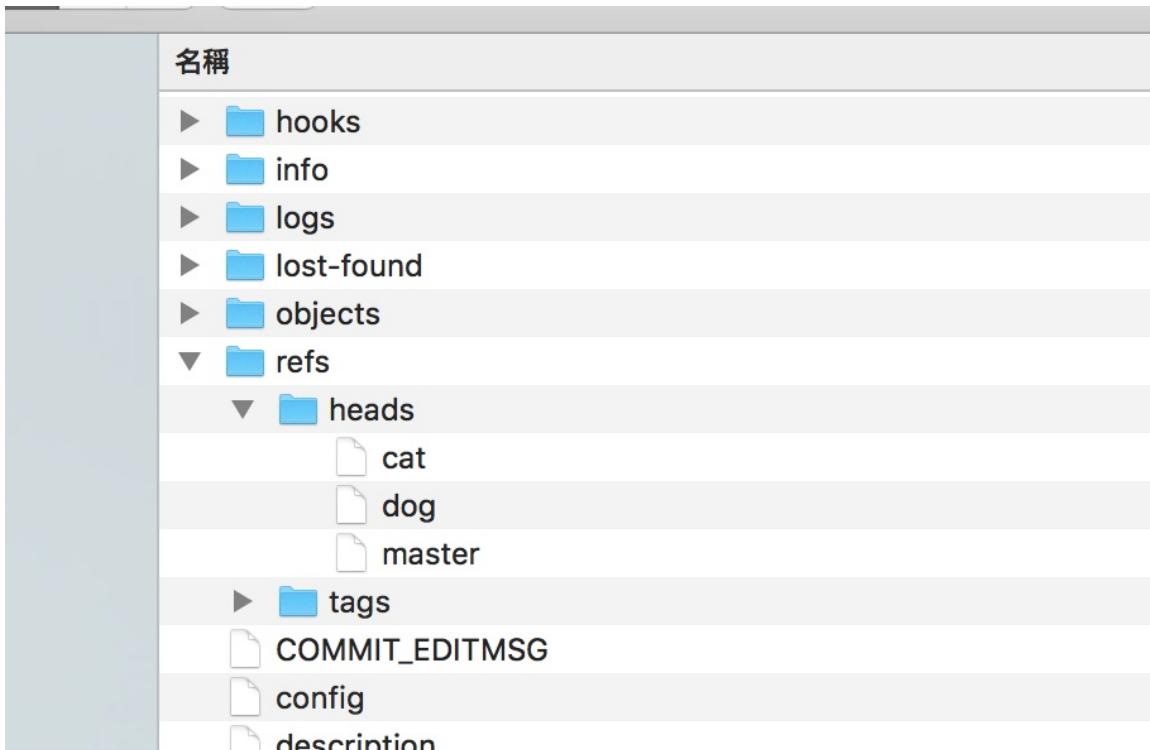
所謂的分支，其實就只是一個有 40 個字元的檔案而已，它藏在 .git 目錄裡面，讓我們進去挖出來給大家看看。先看一下現在有哪些分支：

```
$ git branch
  cat
  dog
* master
```

目前總共有 `cat` 、`dog` 以及 `master` 三個分支。

	All Branches	Description	Commit	Author
Graph	<code>↳ dog add dog 2</code> <code>add dog 1</code> <code>↳ cat add cat 2</code> <code>add cat 1</code> <code>↳ master add database.yml in config folder</code> <code>add hello</code> <code>add container</code> <code>update index page</code> <code>create index page</code>		053fb21 b69eb62 b174a5a c68537b e12d8ef 85e7e30 657fce7 abb4f43 cef6e40	Eddie Kao · Eddie Kao ·

分支的資訊放在哪裡？打開 `.git` 目錄，裡面有一個 `refs` 目錄，再往下有個 `heads` 子目錄：



所謂的分支其實就是放在這裡了！

小提示

- . 小數點開頭的目錄跟檔案通常預設會被作業系統標記成隱藏檔，所以可能會需要打開「檢視隱藏檔」的設定才看得到喔。

看一下這些檔案的內容：

```
$ cat .git/refs/heads/master  
e12d8ef0e8b9deae8bf115c5ce51dbc2e09c8904  
  
$ cat .git/refs/heads/cat  
dd8d48000140bcc66ed0aa5630b6072d5956d32e  
  
$ cat .git/refs/heads/dog  
919df8e360cf2482ff42dd0566f615263fa17214
```

這些數字好像在哪裡看過？你再比對一下上面 SourceTree 的那個畫面，就會發現其實這三個檔案的內容，就只是某個 Commit 的 SHA-1 值而已！

那如果把這些檔案刪掉會怎麼樣嗎？

那就把 `.git/refs/heads/dog` 這個檔案刪掉吧：

```
$ rm .git/refs/heads/dog
```

這時候的分支列表：

```
$ git branch
  cat
* master
```

那個 `dog` 分支就不見了。是的，不用懷疑，在 Git 的分支就只有這樣而已，不信的話，我們來把 `.git/refs/heads/cat` 這個檔案改名成 `bird` 看看：

```
$ mv .git/refs/heads/cat .git/refs/heads/bird
```

再回來看分支列表：

```
$ git branch
  bird
* master
```

原來的 `cat` 分支變成 `bird` 分支了。

所以，回來原來的問題，為什麼大家都說在 Git 開分支「很便宜」？所謂的分支不就是一個只有 40 個字元（某個 Commit 的 SHA-1 值）的檔案，所以在 Git 開一個分支是能多「貴」呢：)

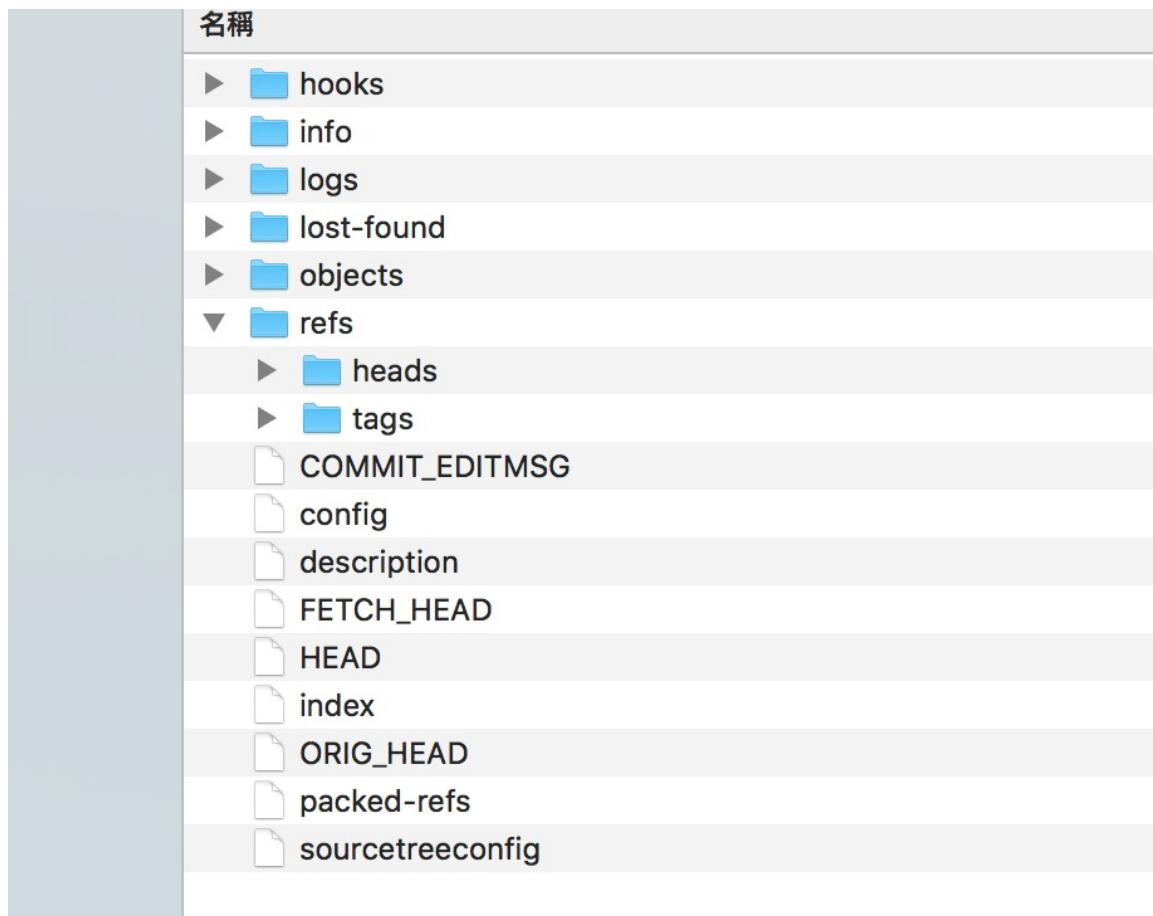
【冷知識】Git 怎麼知道現在是在哪一個分支？

當執行 `git branch` 指令的時候，可以看到目前所有的分支列表：

```
$ git branch
cat
dog
* master
```

看得出來目前共有 3 個分支，前面那個 `*` 號表示目前是處於 `master` 這個分支。那 Git 是在哪裡記錄這個資訊呢？

在 `.git` 目錄裡藏了很多有趣的東西，我們會在「【超冷知識】在 `.git` 目錄裡有什麼東西？」章節有更詳細的介紹。其中在 `.git` 目錄裡有一個名字叫做 `HEAD` 的檔案：



看看裡面是什麼東西：

```
$ cat .git/HEAD
ref: refs/heads/master
```

還記得 `refs/heads/master` 這個檔案的內容嗎？如果忘了可參閱「[【冷知識】為什麼大家都說在 Git 開分支「很便宜」？](#)」的說明。

其實 HEAD 的內容就這麼簡單，它就是記錄著目前指向的分支。我們來試著切換分支看看：

```
$ git checkout cat  
Switched to branch 'cat'
```

然後再看一次 `HEAD` 檔案的內容：

```
$ cat .git/HEAD  
ref: refs/heads/cat
```

在切換分支的時候，HEAD 的內容也會跟著變化。

【冷知識】HEAD 也有縮寫喔

從 Git 1.8.5 之後的版本，在使用 Git 的時候，可以用 `@` 來代替 HEAD，例如原本的指令是這樣：

```
$ git reset HEAD^
```

可以用 `@` 來替代：

```
$ git reset @^
```

但這樣有沒有比原來使用 HEAD 來得清楚，就見人見智了。

ORIG_HEAD 是什麼東西？

在 `.git` 目錄裡除了剛剛說的 HEAD 檔案之外，還有另一個叫做 `ORIG_HEAD` 的檔案，當你在做一些比較「危險」的操作（例如像 `merge`、`rebase` 或 `reset` 之類的），Git 就會把 HEAD 的狀態存放在這裡，讓你隨時可以跳回危險動作之前的狀態。

雖然 `git reflog` 指令也可以查到相關資訊，但 Reflog 的資料比較雜一點，這個 `ORIG_HEAD` 會更方便的讓你找到最近一次危險動作之前的 SHA-1 值。舉例來說：

```
$ git log --oneline  
b174a5a (HEAD -> cat) add cat 2  
c68537b add cat 1  
e12d8ef (master) add database.yml in config folder  
85e7e30 add hello
```

```
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

然後我故意使用 hard 模式執行 Reset 指令，倒退一步：

```
$ git reset HEAD^ --hard
HEAD is now at c68537b add cat 1
```

看一下 ORIG_HEAD 的內容：

```
$ cat .git/ORIG_HEAD
b174a5a95a75c963617409d2fdb514c210d86ba6
```

它記錄的這個內容，正是進行 reset 指令前的 SHA-1 值 b174a5a，所以如果想要取消剛剛這次 reset：

```
$ git reset ORIG_HEAD --hard
HEAD is now at b174a5a add cat 2
```

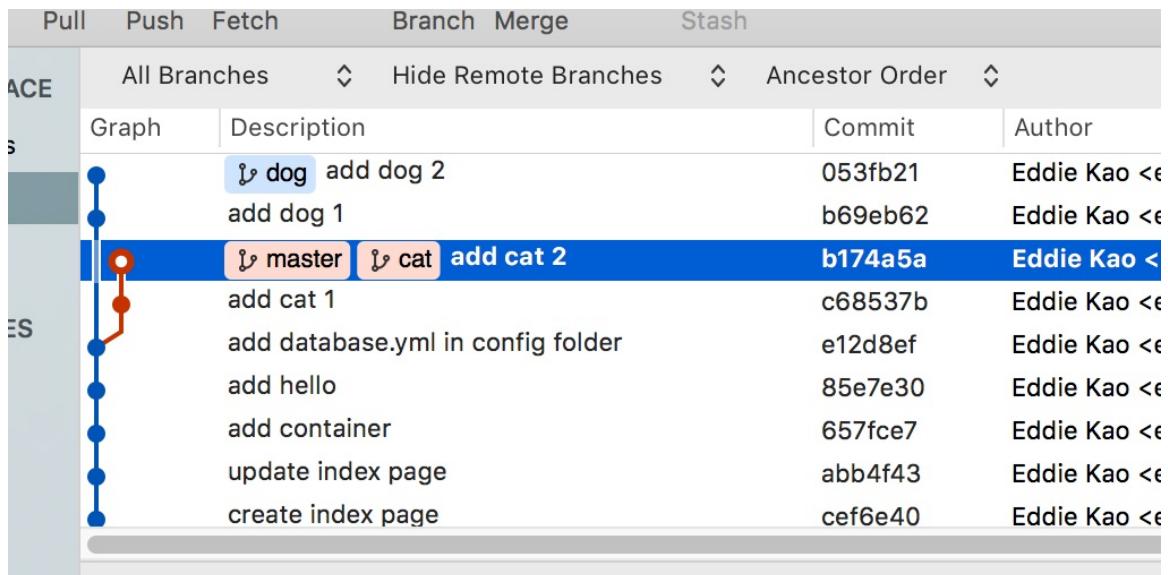
直接使用 reset 指令到 ORIG_HEAD 就能退回 reset 之前的狀態了。再來試一下合併，首先，我先切換到 master 分支：

```
$ git checkout master
Switched to branch 'master'
```

接著準備合併 cat 分支：

```
$ git merge cat
Updating e12d8ef..b174a5a
Fast-forward
 cat1.html | 0
 cat2.html | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat1.html
 create mode 100644 cat2.html
```

沒有意外的使用快轉模式（Fast Forward）進行合併。這時候的狀態應該像這樣：



看一下現在的 `ORIG_HEAD` 的內容：

```
$ cat .git/ORIG_HEAD
e12d8ef0e8b9deae8bf115c5ce51dbc2e09c8904
```

看得出來它指向哪裡嗎？它正指向著合併前的那個 Commit，所以如果想取消這次的合併：

```
$ git reset ORIG_HEAD --hard
HEAD is now at e12d8ef add database.yml in config folder
```

剛剛那次的合併就取消了。這個技巧用在取消 Rebase 合併的時候相當方便，詳情請見「[另一種合併方式（使用 rebase）](#)」章節說明。

【狀況題】我可以從過去的某個 Commit 再長一個新的分支出來嗎？

我想大家一定都有過「如果我在年輕的時候做了另一個選擇，現在一定會不一樣！」的想法。人生，很多事情不能重來，但我們可以用 Git 來補足這個缺憾（嘆？）。

回到過去，重新開始

假設我們現在的歷史紀錄是這樣的：

	All Branches	Hide Remote Branches	Ancestor Order	
	Graph	Description	Commit	Author
		↳ dog add dog 2	053fb21	Eddie Kao <eddie.kao@example.com>
		add dog 1	b69eb62	Eddie Kao <eddie.kao@example.com>
		↳ cat add cat 2	b174a5a	Eddie Kao <eddie.kao@example.com>
		add cat 1	c68537b	Eddie Kao <eddie.kao@example.com>
		↳ master add database.yml in config folder	e12d8ef	Eddie Kao <eddie.kao@example.com>
		add hello	85e7e30	Eddie Kao <eddie.kao@example.com>
		add container	657fce7	Eddie Kao <eddie.kao@example.com>
		update index page	abb4f43	Eddie Kao <eddie.kao@example.com>
		create index page	cef6e40	Eddie Kao <eddie.kao@example.com>
		init commit	cc797cd	Eddie Kao <eddie.kao@example.com>

我想要從 `add container` 那個 Commit (`657fce7`) 再做出新的分支，首先，你得先回到那個 Commit 的狀態，這時候使用的 `git checkout` 指令：

```
$ git checkout 657fce7
Note: checking out '657fce7'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 657fce7... add container
```

這段訊息跟你說，你正處於「detached HEAD」的狀態（請參閱「[【冷知識】斷頭（detached HEAD）是怎麼一回事？](#)」章節說明）。你可以從現在這個 Commit 開新的分支出去，讓我們直接使用 Checkout 指令的 `-b` 參數直接建立分支並自動切換過去：

```
$ git checkout -b bird
Switched to a new branch 'bird'
```

這樣就開好一個 `bird` 分支了。現在的狀況變成這樣：

Full	Push	Fetch	Branch	Merge	Stash
	All Branches		Hide Remote Branches		Ancestor Order
	Graph	Description			Commit
		↳ dog add dog 2			053fb21
		add dog 1			b69eb62
		↳ cat add cat 2			b174a5a
		add cat 1			c68537b
		↳ master add database.yml in config folder			e12d8ef
		add hello			85e7e30
		↳ bird add container			657fce7
		update index page			abb4f43
		create index page			cef6e40
		init commit			cc797cd

咦？不是開分支了？怎麼沒有「樹枝」的樣子？別忘了，分支只是一個像貼紙之類的東西，詳情請參閱「[對分支的誤解](#)」章節。

開好分支，你就可以從這個分支開始再繼續 Commit 了。

一行搞定

如果不想先飛過去再開分支，也是可以直接一行搞定：

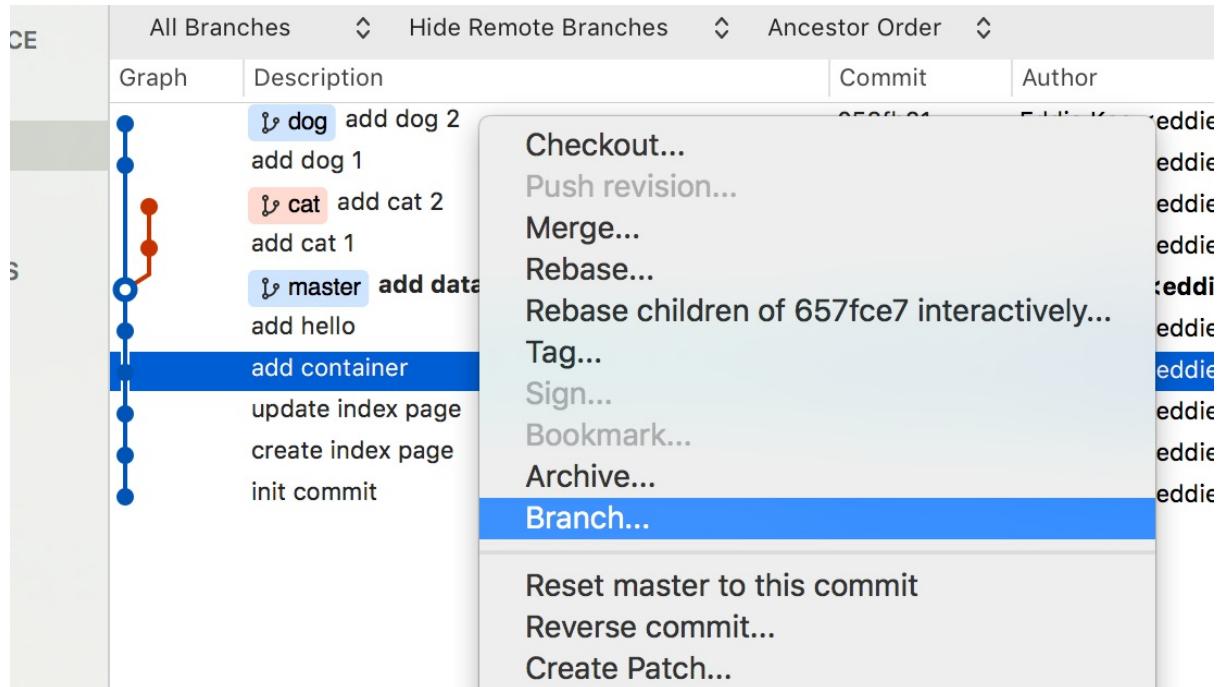
```
$ git branch bird 657fce7
```

意思就是「請幫我用在 `657fce7` 這個 Commit 上開一個叫做 `bird` 的分支」，其實更應該說「請幫我在 `657fce7` 這個 Commit 上幫我貼上一張 `bird` 的分支貼紙」。同時，使用 Checkout 指令配合 `-b` 參數也可以有開分支的效果，而且還會直接切換過去：

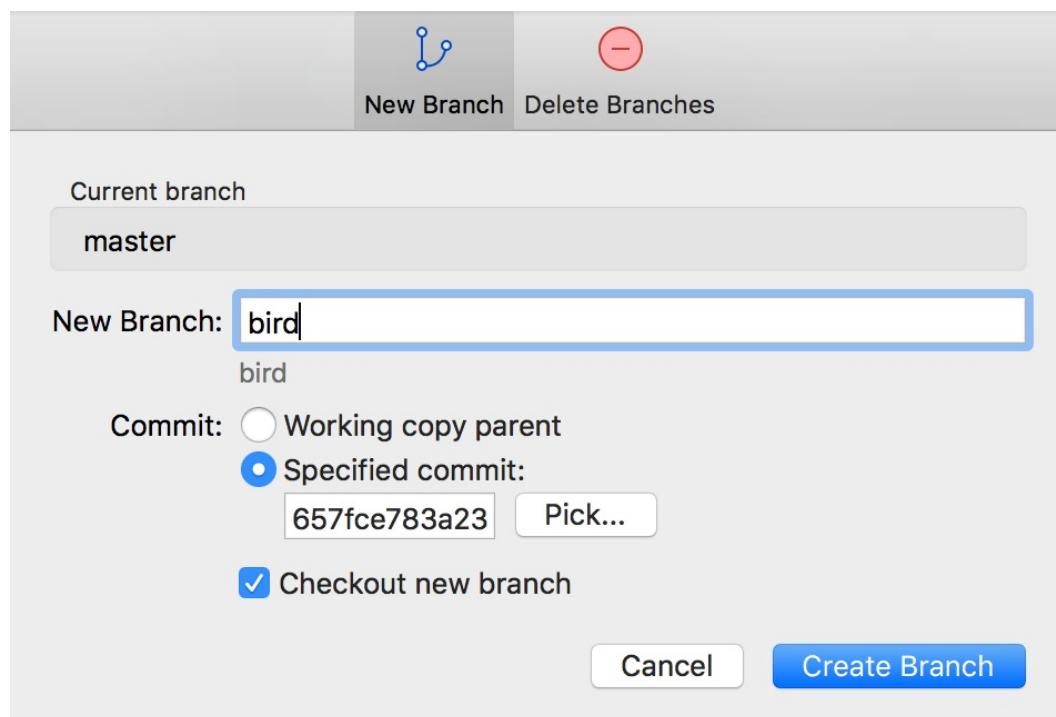
```
$ git checkout -b bird 657fce7
Switched to a new branch 'bird'
```

使用 SourceTree

使用 SourceTree 在指定的地方開分支滿方便的，只要在想開分支的 Commit 上按滑鼠右鍵，選擇「Branch...」功能：



填寫好新的分支的名字：



按下右下角的「Create Branch」按鈕即可完成。

【狀況題】我可以從過去的某個 Commit 再長一個新的分支出來嗎？

【狀況題】修改歷史訊息

要修改歷史訊息，在「[【狀況題】修改 Commit 紀錄](#)」章節曾提過可使用 `--amend` 參數來修改最後一次 Commit 的訊息，但這僅限於最後一次，如果想要修改其它更早的訊息，就得使用別的方法了。

前面曾經介紹過的 `git rebase` 指令，它有一個很厲害的互動模式，接下來這幾個章節都是介紹怎麼使用這個模式來修改過去的歷史。先看一下目前的狀況：

```
$ git log --oneline
27f6ed6 (HEAD -> master) add dog 2
2bab3e7 add dog 1
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

互動模式，啟動！

讓我們使用 `rebase` 指令來整理一下吧：

```
$ git rebase -i bb0c9c2
```

`-i` 參數是指要進入 Rebase 指令的「互動模式」，而後面的 `bb0c9c2` 是指這次的 Rebase 指令的應用範圍會「從現在到 `bb0c9c2` 這個 Commit」，也就是最一開始的那個 Commit。這個指令會跳出一個 Vim 編輯器：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
pick 1de2076 add cat 2
pick ca40fc9 add 2 cats
pick 2bab3e7 add dog 1
pick 27f6ed6 add dog 2

# Rebase bb0c9c2..27f6ed6 onto bb0c9c2 (6 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
```

```
#  
# These lines can be re-ordered; they are executed from top to bottom.  
#  
# If you remove a line here THAT COMMIT WILL BE LOST.  
#  
# However, if you remove everything, the rebase will be aborted.  
#  
# Note that empty commits are commented out
```

說明：

1. 上面的順序，跟 `git log` 指令的結果是相反的，但在 SourceTree 介面是一樣的喔。
2. 前面的 `pick` 的意思是「保留這次的 Commit，不做修改」，其它指令在稍後會再介紹。

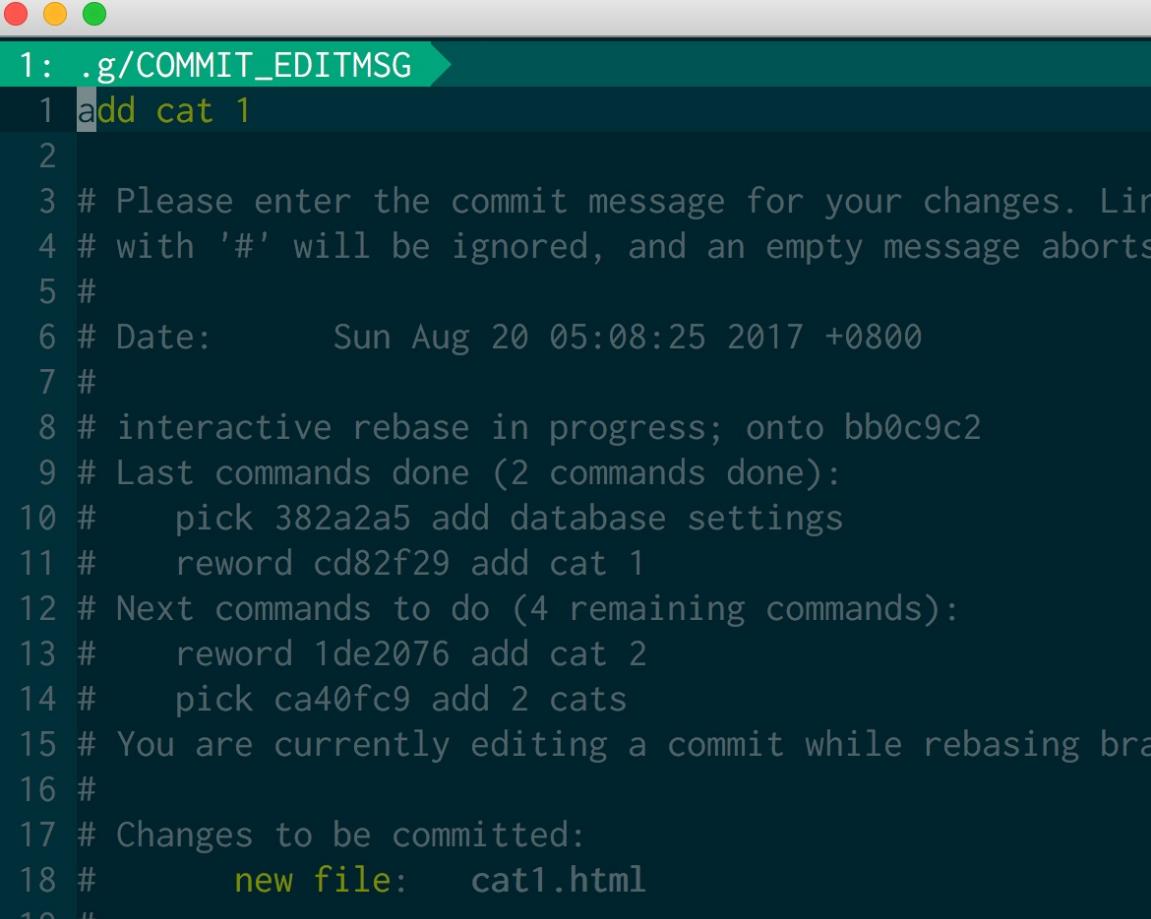
這裡，我把這兩行的內容：

```
pick cd82f29 add cat 1  
pick 1de2076 add cat 2
```

前面的 `pick` 改成 `reword`，或是懶得打字也可以只用 `r` 就好：

```
reword cd82f29 add cat 1  
reword 1de2076 add cat 2
```

表示待會我要來修改這兩次 Commit 的訊息。存檔並離開之後，立馬就會再跳另一個 Vim 編輯器畫面：

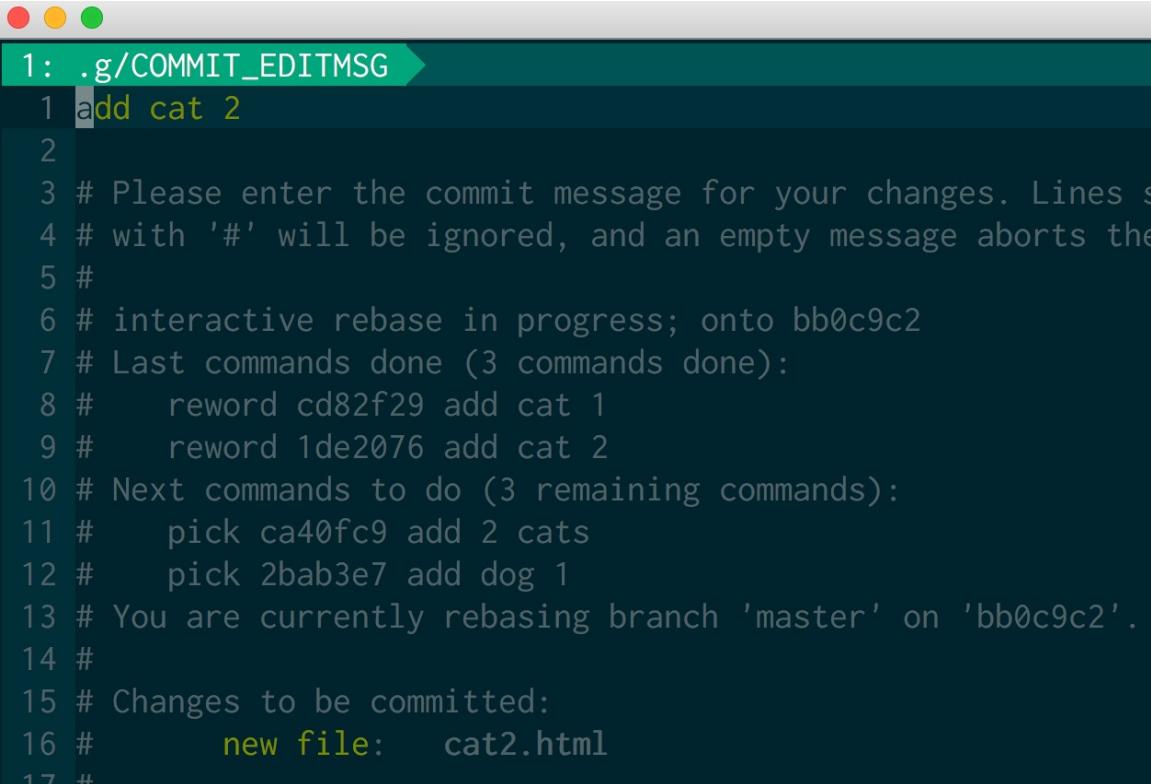


```

1: .g/COMMIT_EDITMSG
1 add cat 1
2
3 # Please enter the commit message for your changes. Lines starting
4 # with '#' will be ignored, and an empty message aborts the
5 #
6 # Date:      Sun Aug 20 05:08:25 2017 +0800
7 #
8 # interactive rebase in progress; onto bb0c9c2
9 # Last commands done (2 commands done):
10 #    pick 382a2a5 add database settings
11 #    reword cd82f29 add cat 1
12 # Next commands to do (4 remaining commands):
13 #    reword 1de2076 add cat 2
14 #    pick ca40fc9 add 2 cats
15 # You are currently editing a commit while rebasing branch 'master'.
16 #
17 # Changes to be committed:
18 #       new file:  cat1.html
19 #

```

還記得剛剛我們說要對那兩次的 Commit 修改訊息嗎？這就是第一次的 `reword`。我把內容改成 `add cat "kitty"`，並存檔、離開，它又陰魂不散的跳出第二個 Vim 編輯器畫面：



```

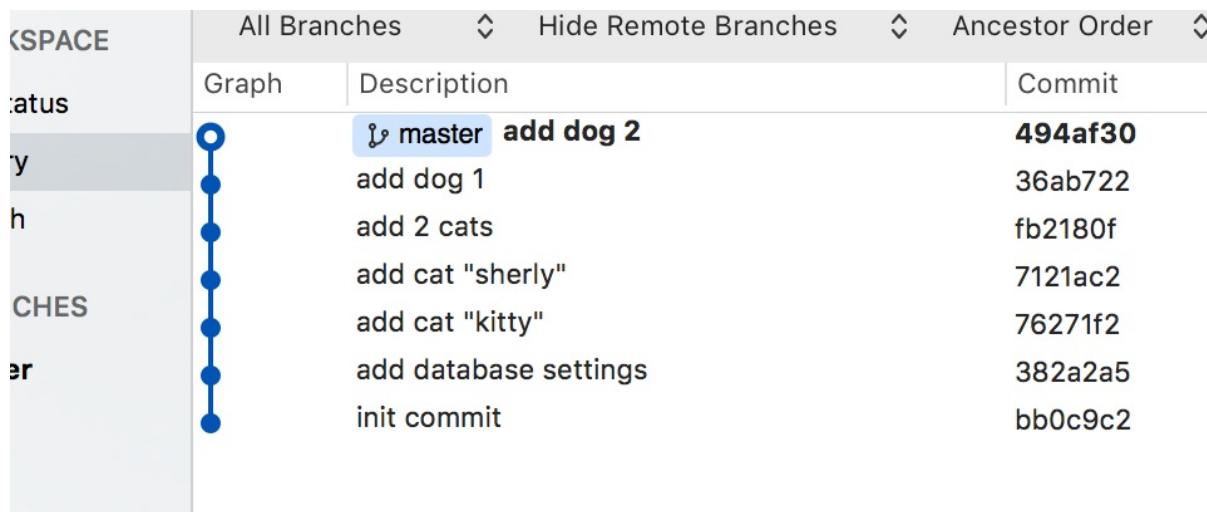
1: .g/COMMIT_EDITMSG
1 add cat 2
2
3 # Please enter the commit message for your changes. Lines starting
4 # with '#' will be ignored, and an empty message aborts the
5 #
6 # interactive rebase in progress; onto bb0c9c2
7 # Last commands done (3 commands done):
8 #    reword cd82f29 add cat 1
9 #    reword 1de2076 add cat 2
10 # Next commands to do (3 remaining commands):
11 #    pick ca40fc9 add 2 cats
12 #    pick 2bab3e7 add dog 1
13 # You are currently rebasing branch 'master' on 'bb0c9c2'.
14 #
15 # Changes to be committed:
16 #       new file:  cat2.html
17 #

```

這是第二次的 `reword`，這回我把內容編輯成 `add cat "sherly"`，存檔、離開之後，Git 就會完成剩下的工作：

```
$ git rebase -i bb0c9c2
[detached HEAD 76271f2] add cat "kitty"
Date: Sun Aug 20 05:08:25 2017 +0800
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 cat1.html
[detached HEAD 7121ac2] add cat "sherly"
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 cat2.html
Successfully rebased and updated refs/heads/master.
```

這時用 SourceTree 看一下歷史紀錄：



就這樣把那兩次的紀錄修掉囉。

只是改訊息，不會怎樣吧？

看起來好像只有改訊息，但這個並不是只有單純的改字這麼單純。如果你仔細看，那兩次 Commit 的 SHA-1 值都變了，原本是 `cd82f29` 跟 `1de2076`，現在變成 `76271f2` 跟 `7121ac2`，這兩次的 Commit 根本就是全新的 Commit 物件了。

在「[另一種合併方式（使用 rebase）](#)」章節也曾介紹到在進行 Rebase 的時候，Commit 物件並不是剪下、貼上而已，而因為要接的前一個 Commit 不同（其實時間也不同），所以會重新計算並做出一顆新的 Commit。

這裡也是一樣，看起來只是改字，但因為 Commit 物件的訊息也會影響 SHA-1 的計算，因為訊息改了，所以 Git 會做出新的 Commit 物件來替代原本舊的 Commit。

不只這樣，在剛剛這個例子裡，因為這兩顆 Commit 物件換掉了，在它之後的 Commit 因為前面的歷史改了，所以後面整串的 Commit 全部都重新做出新的 Commit 出來替代舊的 Commit。

看過卡通「多啦 A 夢」的人應該都知道，當改變了過去的歷史之後，可能會造出新的平行時空出來，大概就跟剛剛這個概念差不多吧。

啊，那想取消剛剛這次 Rebase 的話...

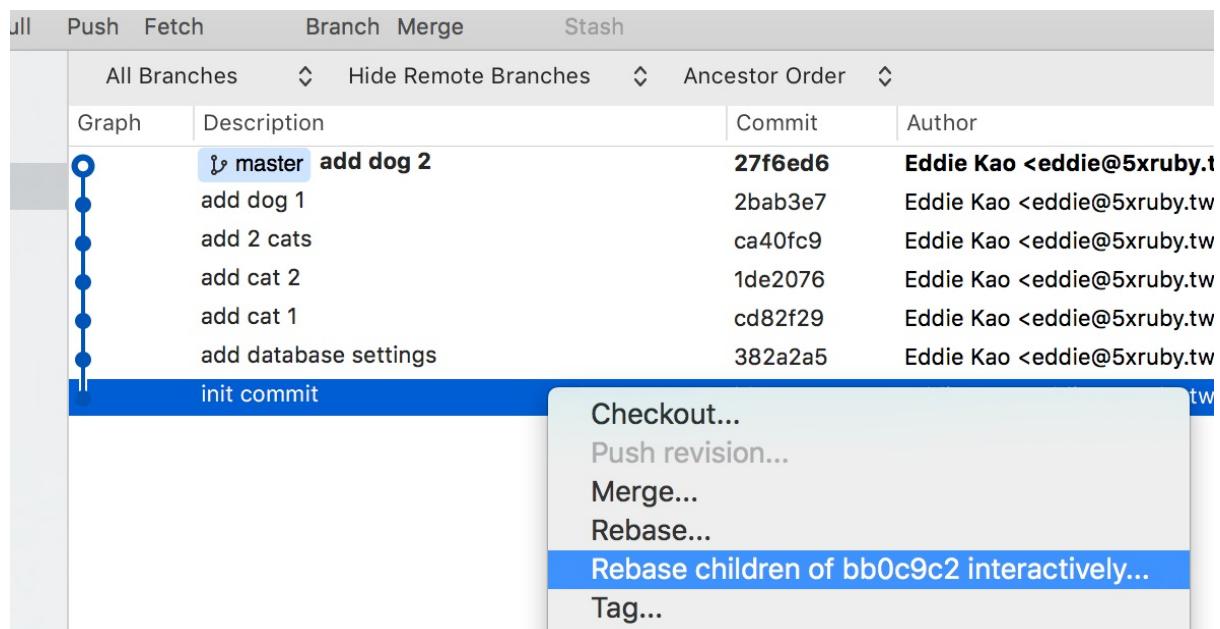
在「[另一種合併方式（使用 rebase）](#)」跟「[【冷知識】Git 怎麼知道現在是在哪一個分支？](#)」這兩個章節都有介紹過關於 `ORIG_HEAD` 這東西，如果想要取消這回的 Rebase 的話，只要這樣：

```
$ git reset ORIG_HEAD --hard
```

就會回到 Rebase 之前囉。

使用 SourceTree

使用 SourceTree 來修改歷史訊息會比使用指令來得簡單一些，就選擇你想要修改的範圍，在指定的 Commit 上按滑鼠右鍵，選擇「Rebase children of SHA-1 interactively...」：

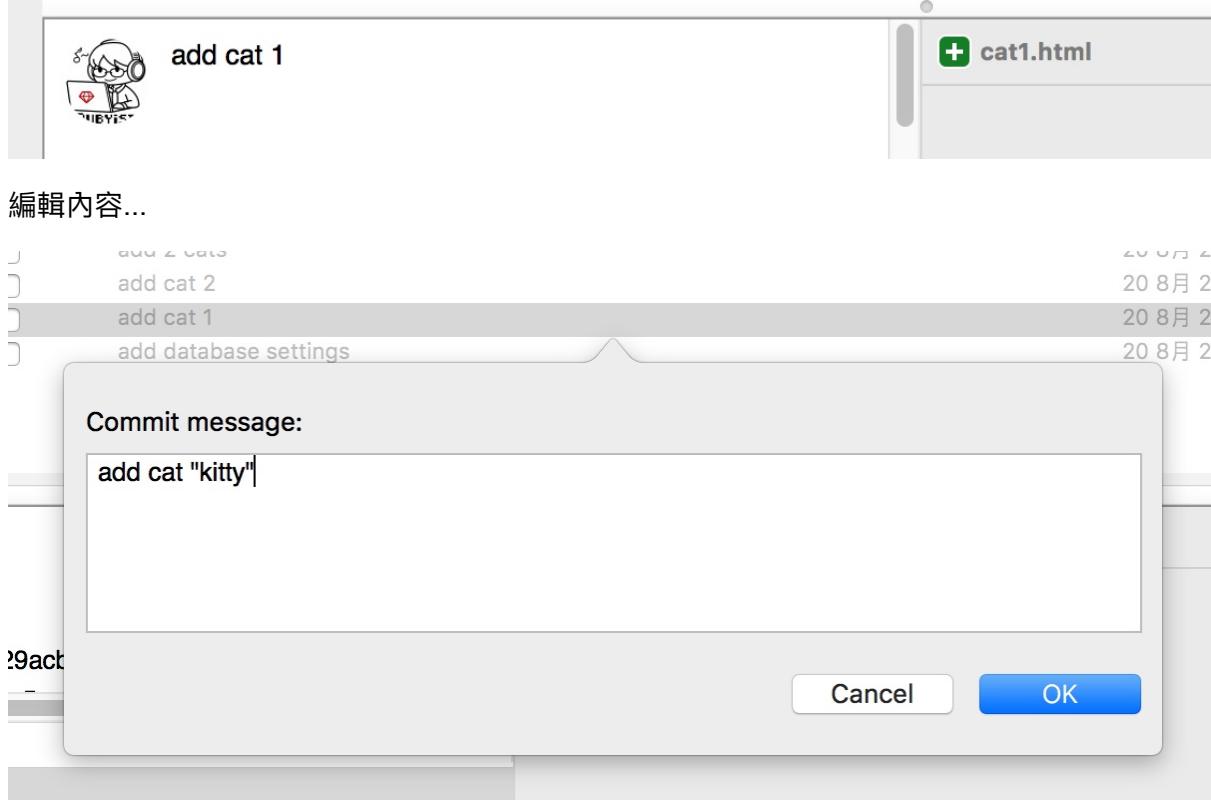


接著找到你想要修改訊息的 Commit 上，按滑鼠右鍵選擇「Edit message...」：

Reorder and amend commits:

Changeset	Amend Commit?	Description
27f6ed6	<input type="checkbox"/>	add dog 2
2bab3e7	<input type="checkbox"/>	add dog 1
ca40fc9	<input type="checkbox"/>	add 2 cats
1de2076	<input type="checkbox"/>	add cat 2
cd82f29	<input checked="" type="checkbox"/>	add cat 1
382a2a5	<input type="checkbox"/>	add database settings

[Edit message...](#)
[Squash with previous comm](#)
[Delete commit...](#)



編輯內容...

add cat 2	20 8月 2
add cat 1	20 8月 2
add database settings	20 8月 2

Commit message:

add cat "kitty"

Cancel
OK

可以再繼續選擇其它 Commit 繼續進行修改訊息，待全部都完成之後，按下右下角的 OK 鈕，即可開始進行 Rebase。

【狀況題】把多個 Commit 合併成一個 Commit

有時候 Commit 的太過「瑣碎」，舉個例子來說：

```
$ git log --oneline
27f6ed6 (HEAD -> master) add dog 2
2bab3e7 add dog 1
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

在 `cd82f29` 跟 `1de2076` 這兩個 Commit 都只有各加一個檔案（分別是 `cat1.html` 跟 `cat2.html`），`2bab3e7` 跟 `27f6ed6` 也一樣，都只各加了一個檔案而已。如果想把這幾個 Commit 合併成一個，會讓 Commit 看起來更乾淨一些。同樣可以使用互動模式的 Rebase 來處理：

```
$ git rebase -i bb0c9c2
```

接著一樣再次出現 Vim 編輯器視窗，內容如下：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
pick 1de2076 add cat 2
pick ca40fc9 add 2 cats
pick 2bab3e7 add dog 1
pick 27f6ed6 add dog 2

# Rebase bb0c9c2..27f6ed6 onto bb0c9c2 (6 commands)
#
# Commands:
# ...[略]...
```

這裡我用的指令是 `squash`，把上面的內容修改成這樣：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
squash 1de2076 add cat 2
squash ca40fc9 add 2 cats
pick 2bab3e7 add dog 1
squash 27f6ed6 add dog 2
```

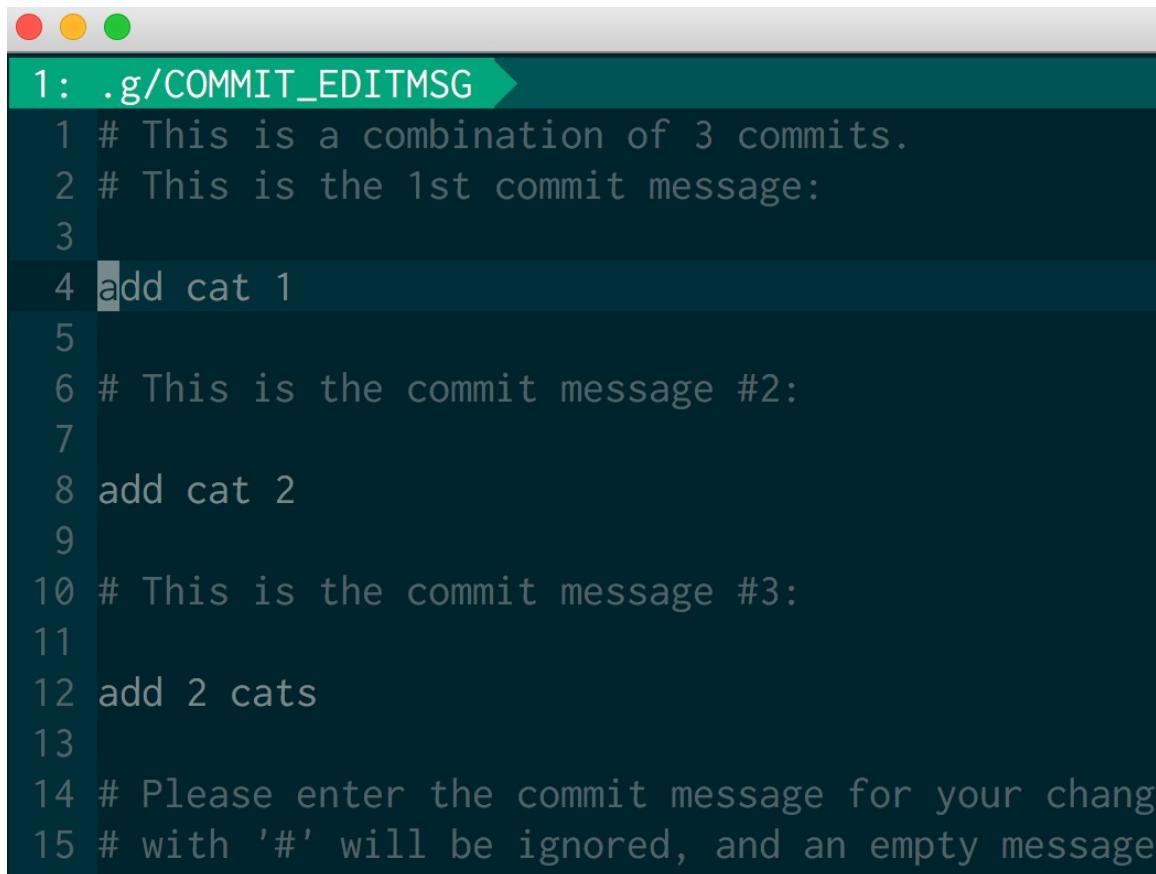
注意！

在互動模式的紀錄由上而下是從最舊到最新，跟 `git log` 指令所呈現的結果是相反的。

上面這樣的修改表示接下來會發生這些事：

- 最後一行的 `27f6ed6` 會跟前一個 Commit `2bab3e7` 進行合併，也就是 `add dog 1` 跟 `add dog 2` 這個 Commit 會合在一起。
- 倒數第三號的 `ca40fc9` 會跟前一個 Commit `1de2076` 合併，但因為 `1de2076` 又會再往前一個 Commit `cd82f29` 合併，所以整個跟 `cat` 有關的這三個 Commit 會併成同一個。

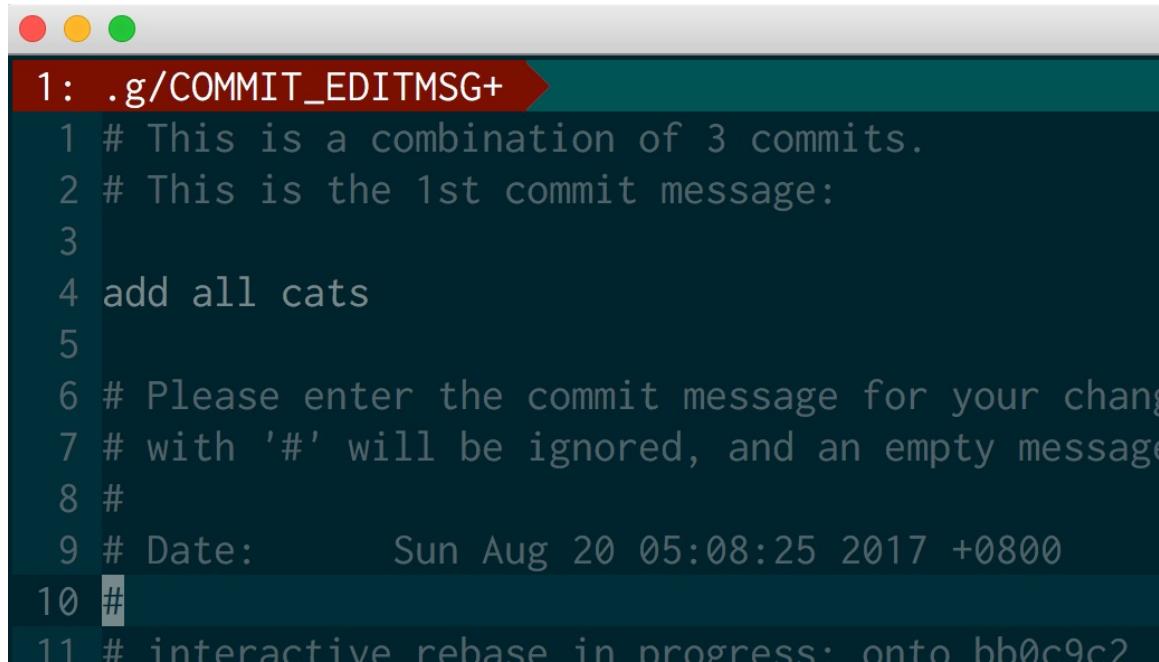
存檔並離開 Vim 編輯器後，它會開始進行 Rebase，而在 Squash 的過程中，它還會跳出 Vim 編輯器讓你編輯一下訊息：



The screenshot shows a terminal window with a dark theme. The title bar says "1: .g/COMMIT_EDITMSG". The main area contains the following text:

```
1: .g/COMMIT_EDITMSG
1 # This is a combination of 3 commits.
2 # This is the 1st commit message:
3
4 add cat 1
5
6 # This is the commit message #2:
7
8 add cat 2
9
10 # This is the commit message #3:
11
12 add 2 cats
13
14 # Please enter the commit message for your change
15 # with '#' will be ignored, and an empty message
```

我把訊息改成「add all cats」：

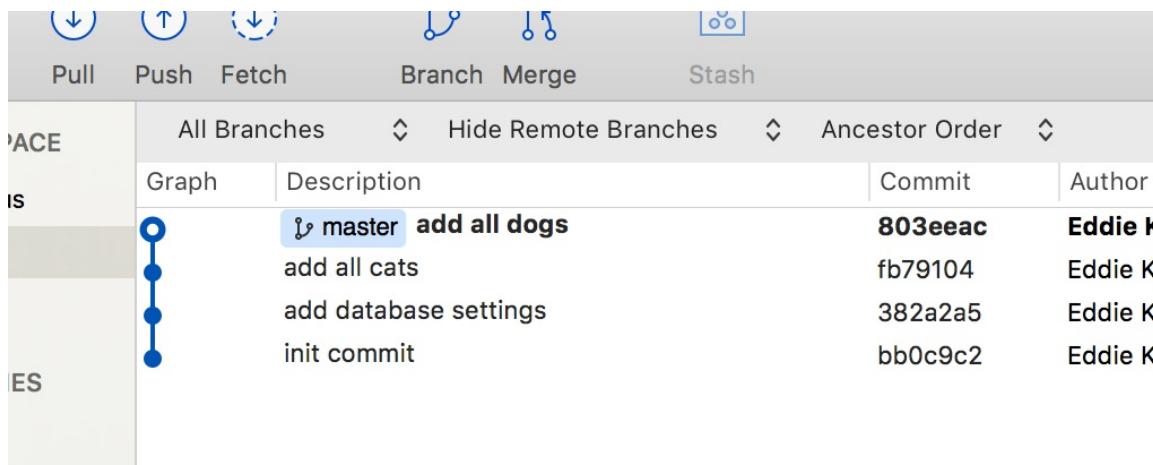


```
1: .g/COMMIT_EDITMSG+
1 # This is a combination of 3 commits.
2 # This is the 1st commit message:
3
4 add all cats
5
6 # Please enter the commit message for your change
7 # with '#' will be ignored, and an empty message
8 #
9 # Date:      Sun Aug 20 05:08:25 2017 +0800
10 #
11 # interactive rebase in progress: onto bb0c9c2
```

同樣的，在另一次的 Squash 也會再編輯一次 Commit 訊息，我把它改成「add all dogs」。整個 Rebase 的訊息如下：

```
$ git rebase -i bb0c9c2
[detached HEAD fb79104] add all cats
Date: Sun Aug 20 05:08:25 2017 +0800
4 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 cat1.html
create mode 100644 cat2.html
create mode 100644 cat3.html
create mode 100644 cat4.html
[detached HEAD 803eeac] add all dogs
Date: Sun Aug 20 05:09:53 2017 +0800
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dog1.html
create mode 100644 dog2.html
Successfully rebased and updated refs/heads/master.
```

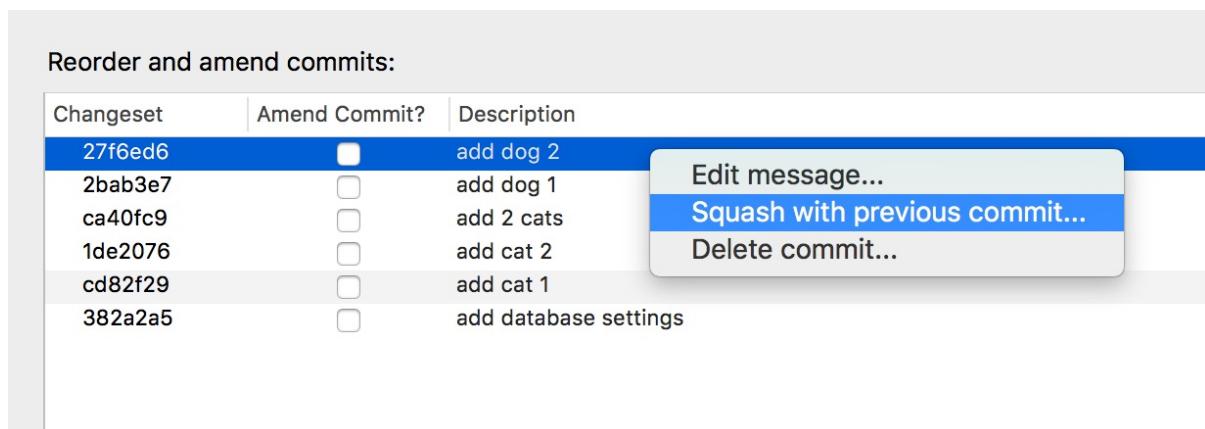
這時候的歷史紀錄就變成這樣了：



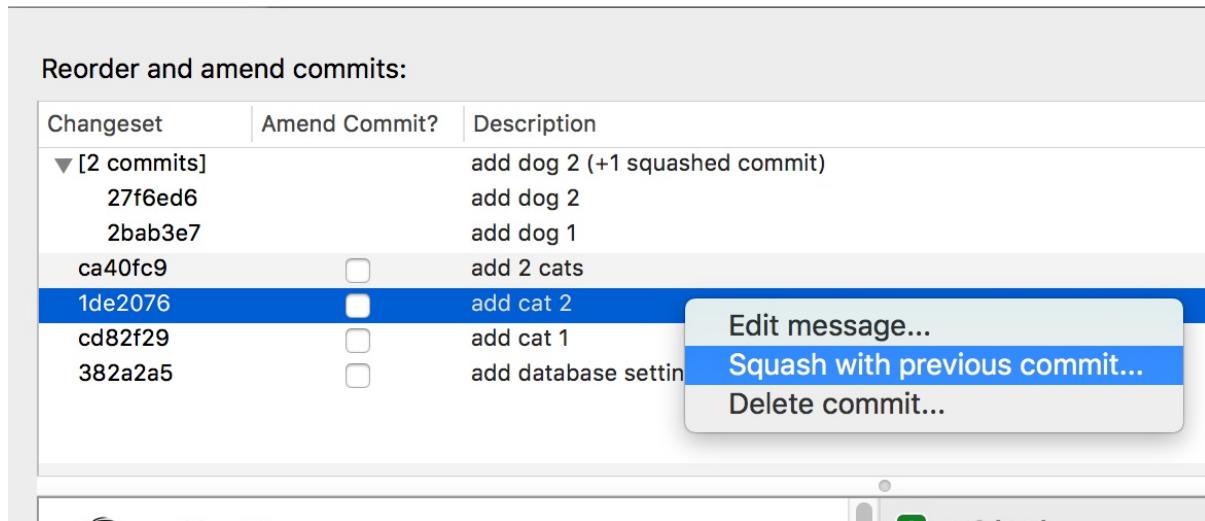
這樣就把剛剛那些貓貓狗狗的，全部整理成兩個 Commit 了。

使用 SourceTree

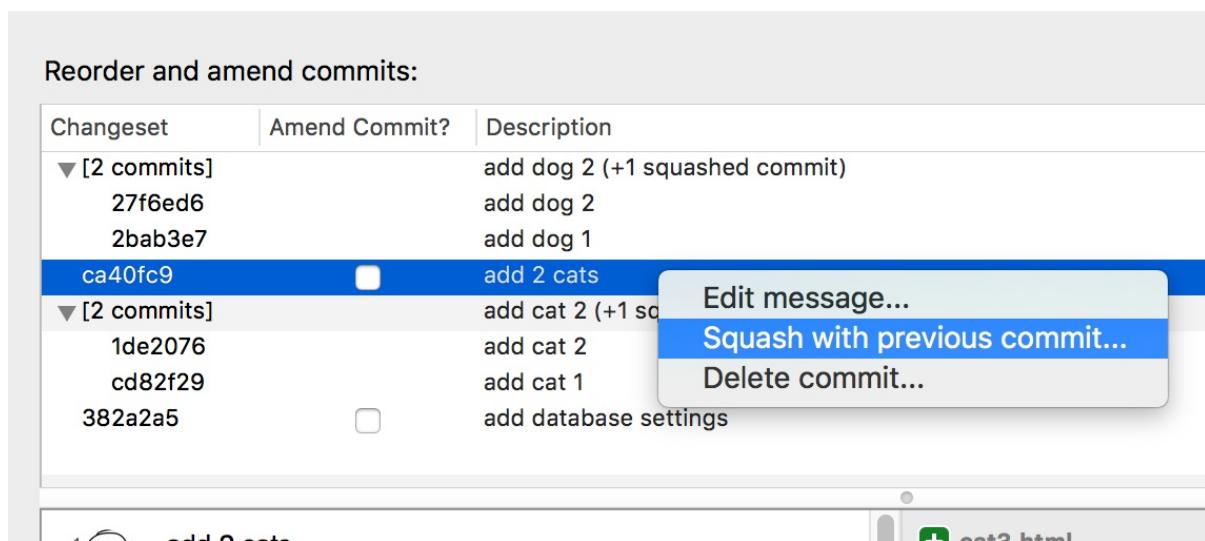
使用 SourceTree 在歷史紀錄上的 Commit 上按滑鼠右鍵，選擇「Rebase children of SHA-1 interactively...」，進入互動模式。這時，我先在 `add dog 2` 這個 Commit 上按滑鼠右鍵並選擇「Squash with previous commit...」：



接著是在 `add cat 2` 上做一樣的事：



再來是 `add 2 cats` :



注意！

因為這邊是要合併三個 Commit，如果上面這兩個步驟反過來，你會發現沒辦法順利的「Squash」，這時只要按下 `Cmd + z` 就可以回到上一步再重來一次。

接著，編輯剛剛「濃縮」的這兩個 Commit 的訊息，這是編輯完成的 Commit 訊息之後的樣子：

Reorder and amend commits:

Changeset	Amend Commit?	Description
▼ [2 commits]		add all dogs
27f6ed6		add dog 2
2bab3e7		add dog 1
▼ [3 commits]		add all cats
1de2076		add cat 2
cd82f29		add cat 1
ca40fc9		add 2 cats
382a2a5	<input type="checkbox"/>	add database settings

There are a total of 5 commits.

cat2.html

全部完成之後，按下右下角的 OK 鈕就會開始進行 Rebase，這樣就可以把多個 Commit 合併成一個了。

【狀況題】把一個 Commit 拆解成多個 Commit

跟上個章節「[【狀況題】把多個 Commit 合併成一個 Commit](#)」相反，有時候覺得單次 Commit 的檔案太多了，可能會想把它拆得細一點，同樣也可使用互動模式的 Rebase 來操作。這是目前的歷史紀錄：

```
$ git log --oneline
27f6ed6 (HEAD -> master) add dog 2
2bab3e7 add dog 1
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

`ca40fc9` 這個 Commit 一口氣增加了兩個檔案，我想把它拆成兩個 Commit，每個 Commit 分別都只有一個檔案就好。起手式跟上個章節一樣：

```
$ git rebase -i bb0c9c2
```

一樣跳出一個 Vim 編輯器。這次，我把我要拆的那個 Commit 的 `pick` 改成 `edit`：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
pick 1de2076 add cat 2
edit ca40fc9 add 2 cats
pick 2bab3e7 add dog 1
pick 27f6ed6 add dog 2
```

Rebase 在執行到 `ca40fc9` 這個 Commit 的時候就會停下來：

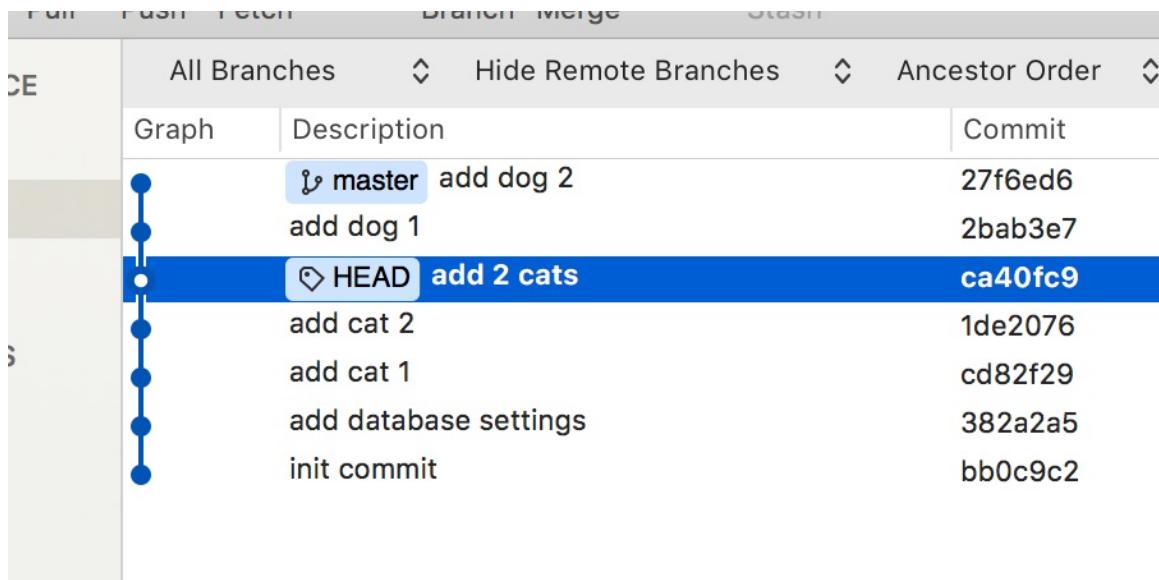
```
$ git rebase -i bb0c9c2
Stopped at ca40fc9... add 2 cats
You can amend the commit now, with

  git commit --amend

Once you are satisfied with your changes, run

  git rebase --continue
```

看目前的狀態的話會像這樣：



這時，因為我們要把目前這個 Commit 拆成兩個 Commit，還記得怎麼拆 Commit 嗎？沒錯，就是使用 Reset 指令：

```
$ git reset HEAD^
```

如果忘記，請回「[【狀況題】剛才的 Commit 後悔了，想要拆掉重做...](#)」章節復習一下。看一下目前的狀態：

```
$ git status
interactive rebase in progress; onto bb0c9c2
Last commands done (4 commands done):
  pick 1de2076 add cat 2
  edit ca40fc9 add 2 cats
  (see more in file .git/rebase-merge/done)
Next commands to do (2 remaining commands):
  pick 2bab3e7 add dog 1
  pick 27f6ed6 add dog 2
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'master' on 'bb0c9c2'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    cat3.html
    cat4.html

nothing added to commit but untracked files present (use "git add" to track)
```

可以看到 `cat3.html` 跟 `cat4.html` 都被拆出來放在工作目錄並且是處於 `Untracked` 狀態。那，還記得怎麼 Commit 檔案嗎？就是用 `add + commit` 二段式指令：

```
$ git add cat3.html
```

把檔案加到暫存區，接下來進行 Commit：

```
$ git commit -m "add cat 3"  
[detached HEAD 8e79d0e] add cat 3  
 1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 cat3.html
```

還有另一個檔案，依樣畫葫蘆：

```
$ git add cat4.html  
$ git commit -m "add cat 4"  
[detached HEAD 06ee3f6] add cat 4  
 1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 cat4.html
```

同樣的，如果忘記怎麼 Commit 檔案，請再參考「[把檔案交給 Git 控管](#)」章節說明。

這樣就把剛剛那個 Commit 拆成兩個 Commit 了。但別忘了現在還是處於 Rebase 狀態，所以要讓 Rebase 繼續跑完喔：

```
$ git rebase --continue  
Successfully rebased and updated refs/heads/master.
```

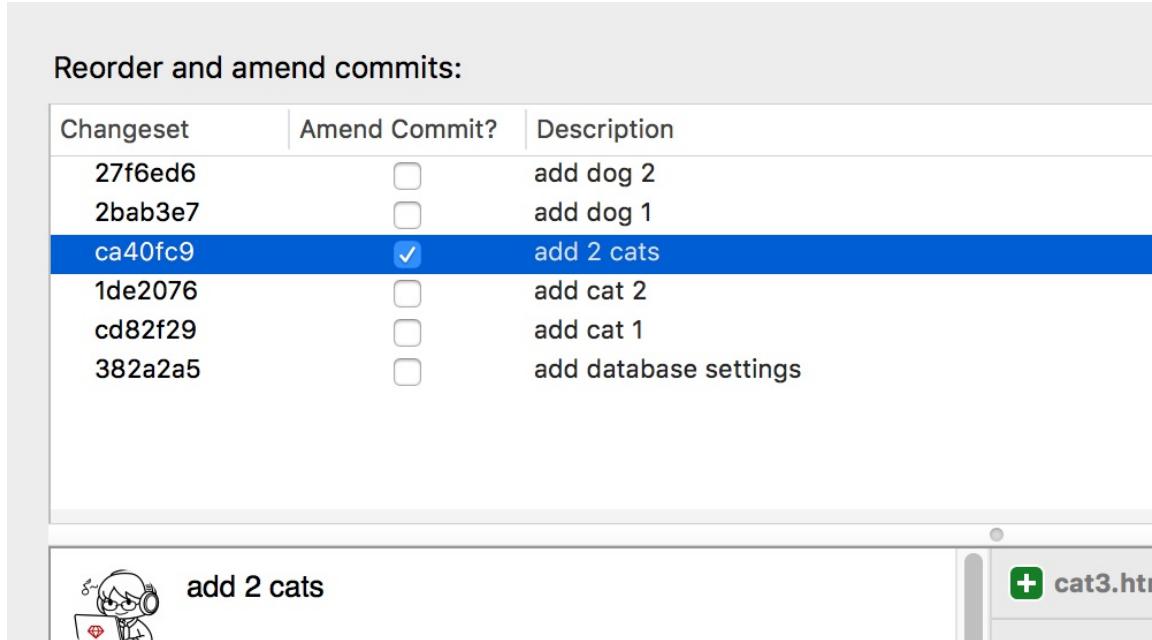
看一下現在的歷史紀錄：

	Pull	Push	Fetch	Branch	Merge	Stash	
KSPACE	All Branches		Graph	Description		Ancestor Order	
status	Graph	Description		Commit	Author		
ry		↳ master add dog 2		e6850a7	Eddie Kao		
:h		add dog 1		4ca9154	Eddie Kao		
ICIES		add cat 4		06ee3f6	Eddie Kao		
er		add cat 3		8e79d0e	Eddie Kao		
		add cat 2		1de2076	Eddie Kao		
		add cat 1		cd82f29	Eddie Kao		
		add database settings		382a2a5	Eddie Kao		
OTES		init commit		bb0c9c2	Eddie Kao		

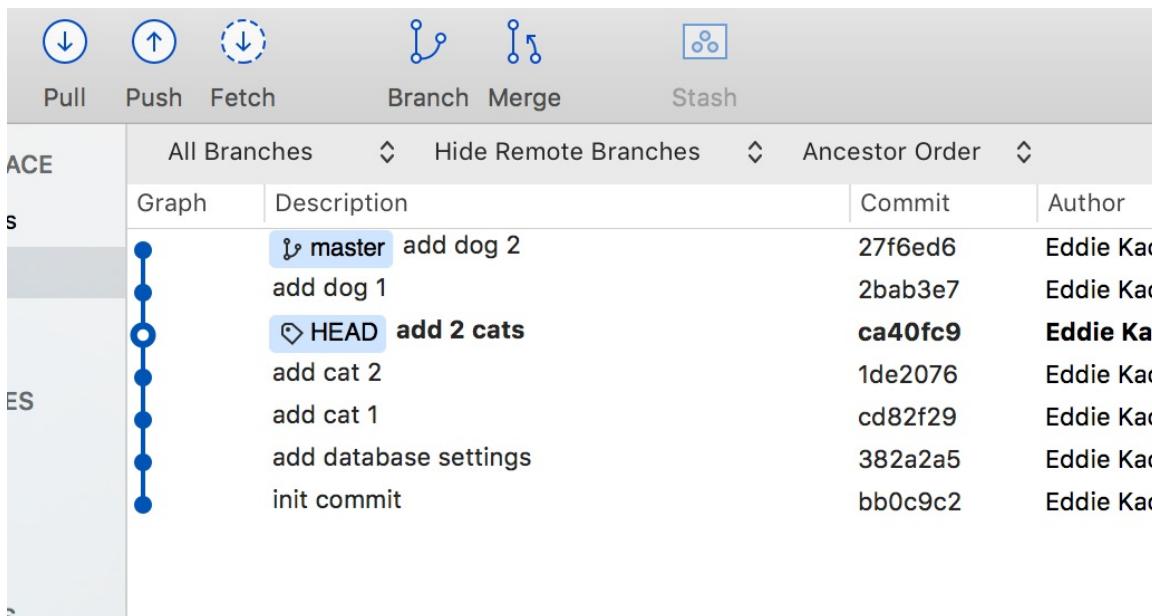
原來的 `add 2 cats` 已經被拆成 `add cat 3` 跟 `add cat 4` 了，而且各別都只有一個檔案囉。

使用 SourceTree

這個使用 SourceTree 就沒有方便的介面可以處理了。一樣先叫出互動模式的 Rebase 視窗，然後在要拆的那個 Commit 上打勾：

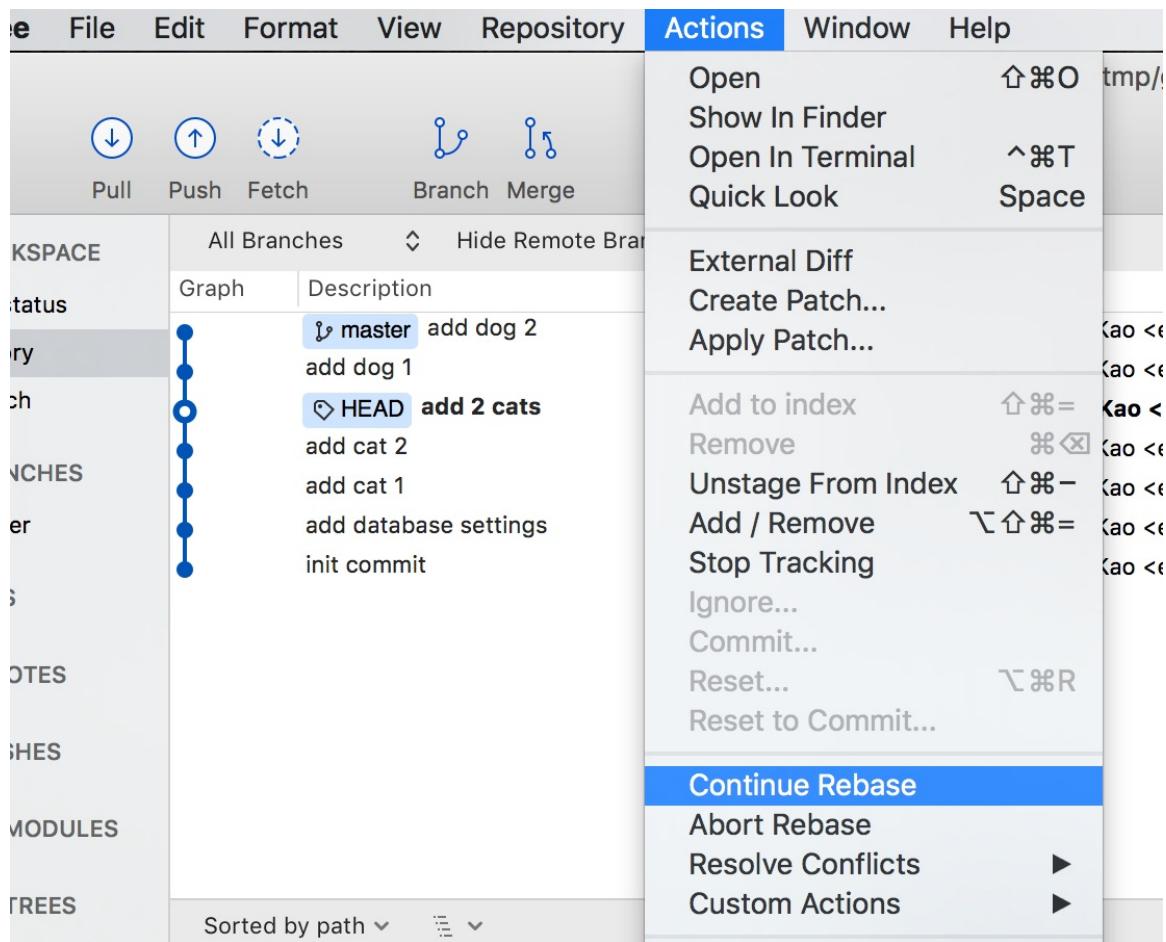


打勾就是代表待會在這個 Commit 要停下來編輯的意思。按下右下角 OK 鈕便會開始執行 Rebase 指令，接著你會看到這個畫面：



HEAD 的確停在 `add 2 cats` 那個 Commit 了。接下來呢.. 因為 SourceTree 似乎沒有提供可以做這件事情的介面，所以，就乖乖打開終端機，開始上面那些動作吧 :)

喔，忘了說，如果做完的話，在 SourceTree 的功能選單→「Action」→「Continue Rebase」可以繼續完成剩下的 Rebase 指令：



跟 `git rebase --continue` 指令是一樣的功效。

【狀況題】想要在某些 Commit 之間再加新的 Commit

偶爾你可能會想在某些 Commit 之間再增加一些其它的 Commit，假設目前的歷史紀錄如下：

```
$ git log --oneline
27f6ed6 (HEAD -> master) add dog 2
2bab3e7 add dog 1
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

在 `ca40fc9 (add 2 cats)` 跟 `2bab3e7 (add dog 1)` 這兩個 Commit 之間，我想再增加兩個 Commit。這個使用的技巧其實跟上一章「[【狀況題】把一個 Commit 拆解成多個 Commit](#)」有點像，都是先停在某個 Commit 之後再來做事。所以起手式一樣是這樣：

```
$ git rebase -i bb0c9c2
```

再次提醒大家，在 Rebase 狀態的 Commit 列表跟平常我們看的紀錄是反過來的，所以如果你想在某兩個 Commit 之間再增加 Commit，要注意停下來的那個點是不是正確的點。例如我想加在 `add 2 cats` 跟 `add dog 1` 之間：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
pick 1de2076 add cat 2
edit ca40fc9 add 2 cats
pick 2bab3e7 add dog 1
pick 27f6ed6 add dog 2
```

注意，是停在 `add 2 cats` 喔，所以我把那個 Commit 改成 `edit`，接著繼續執行 Rebase：

```
$ git rebase -i bb0c9c2
Stopped at ca40fc9... add 2 cats
You can amend the commit now, with

    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue
```

接下來，我很快的加兩個 Commit：

```
$ touch bird1.html
$ git add bird1.html
$ git commit -m "add bird 1"
[detached HEAD 549bd92] add bird 1
 1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 bird1.html

$ touch bird2.html
$ git add bird2.html
$ git commit -m "add bird 2"
[detached HEAD e13837e] add bird 2
 1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 bird2.html
```

如果眼尖的朋友可能會注意到在 Rebase 的過程中常出現「detached HEAD」字樣，如果忘記它是什麼意思，可以參閱「[【狀況題】我可以從過去的某個 Commit 再長一個新的分支出來嗎？](#)」章節說明。

加好 2 個 Commit 之後，就繼續剛剛中斷的 Rebase 吧：

```
$ git rebase --continue
Successfully rebased and updated refs/heads/master.
```

這樣就在指定的位置中間增加新的 Commit 了：

		All Branches	Hide Remote Branches	Ancestor Order
	Graph	Description	Commit	Author
CE		master add dog 2	e94ebf1	Eddie Kao <
S		add dog 1	4644333	Eddie Kao <
		add bird 2	e13837e	Eddie Kao <
		add bird 1	549bd92	Eddie Kao <
		add 2 cats	ca40fc9	Eddie Kao <
		add cat 2	1de2076	Eddie Kao <
		add cat 1	cd82f29	Eddie Kao <
		add database settings	382a2a5	Eddie Kao <
		init commit	bb0c9c2	Eddie Kao <

如果使用 SourceTree，基本上跟上一章「[【狀況題】把一個 Commit 拆解成多個 Commit](#)」章節介紹的差不多，也是勾選要中斷的 Commit，然再回到終端機進行操作。搞定之後可在 SourceTree 的功能選單→「Action」→「Continue Rebase」便可完成剩下的 Rebase 指令。

【狀況題】想要刪除某幾個 Commit 或是調整 Commit 的順序

調整 Commit 順序

要在 Git 裡調整 Commit 的順序其實滿簡單的，假設這是目前的歷史紀錄：

```
$ git log --oneline
27f6ed6 (HEAD -> master) add dog 2
2bab3e7 add dog 1
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

我想讓所有跟 cat 有關的 Commit 都移到 dog 相關的 Commit 的後面，一樣的 Rebase 起手式：

```
$ git rebase -i bb0c9c2
```

這時候跳出來的編輯器的 Commit 內容是這樣：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
pick 1de2076 add cat 2
pick ca40fc9 add 2 cats
pick 2bab3e7 add dog 1
pick 27f6ed6 add dog 2
```

別忘了，在 Rebase 狀態看到的這個紀錄是跟我們平常看的紀錄是反過來的，越新的 Commit 在越下面。接下來我只要這樣移動一下：

```
pick 382a2a5 add database settings
pick 2bab3e7 add dog 1
pick 27f6ed6 add dog 2
pick cd82f29 add cat 1
pick 1de2076 add cat 2
pick ca40fc9 add 2 cats
```

是的，就是只要做這樣的修改，存檔、離開後，Rebase 就會做它的工作：

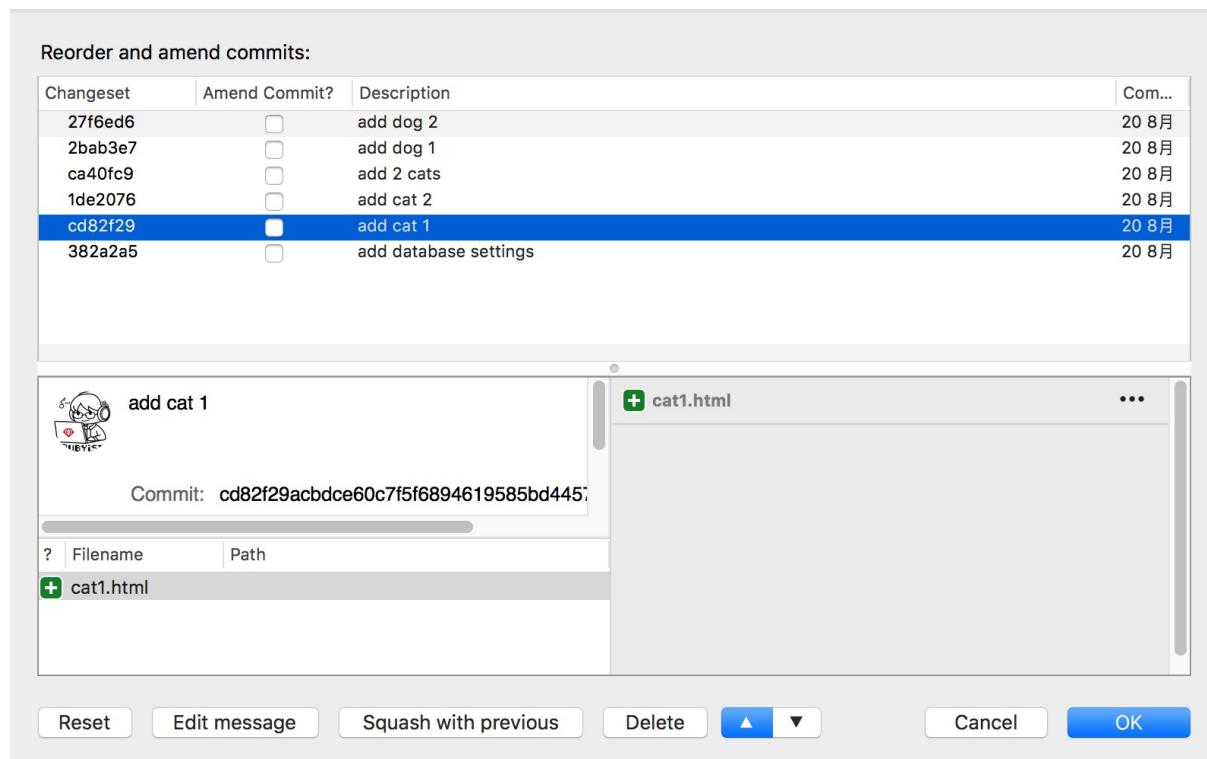
```
$ git rebase -i bb0c9c2
Successfully rebased and updated refs/heads/master.
```

搞定！現在的歷史紀錄就變這樣了：

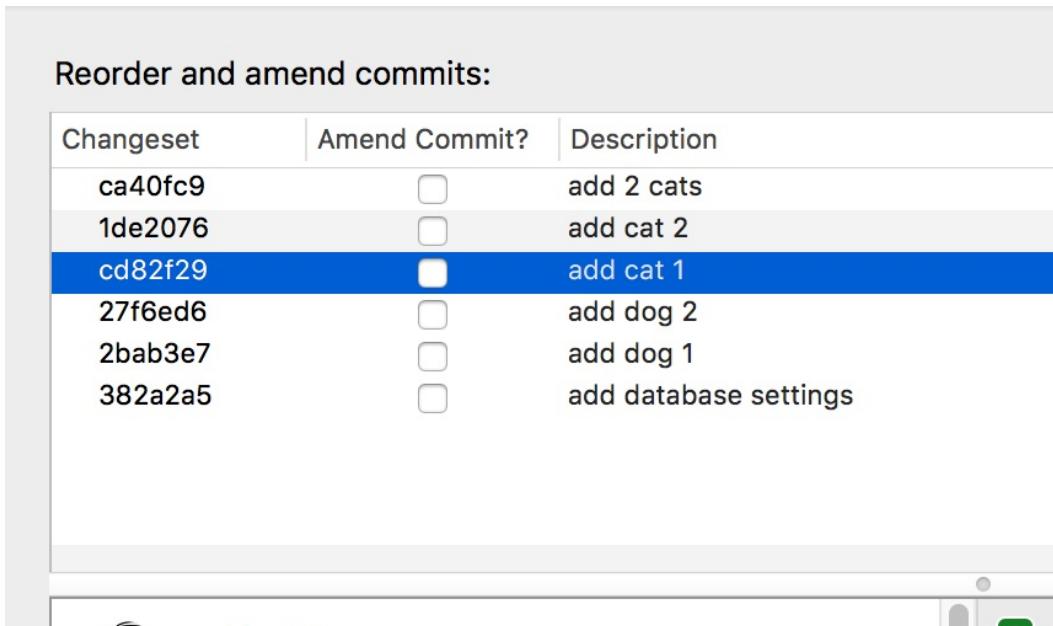
```
$ git log --oneline
a2df4b2 (HEAD -> master) add 2 cats
a8e28c5 add cat 2
1e51a3d add cat 1
9f6a6a5 add dog 2
5a14212 add dog 1
382a2a5 add database settings
bb0c9c2 init commit
```

所有 `cat` 相關的 Commit 都搬到 `dog` 後面了。

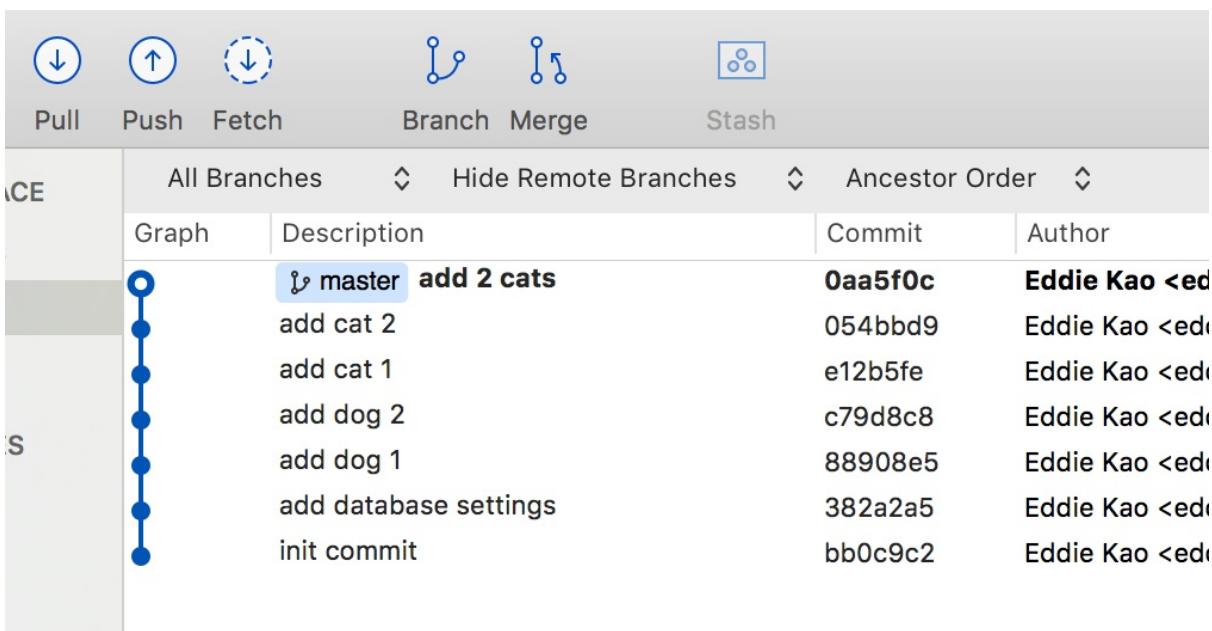
使用 SourceTree 也挺容易的，用同樣的方式叫出 Rebase 視窗：



這時候可以用滑鼠拖拉順序，或是在視窗的上方有可以調整上下的按鈕，拉好期望的順序後：



按下 OK 鍵，便會進行 Rebase 指令：



輕輕鬆鬆就搞定了！

刪除 Commit

要刪除 Commit 更簡單了，在 Rebase 的過程中，把原本的 `pick` 改成 `drop`，或甚至直接把那行刪掉也可以。例如原本的 Commit 是這樣：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
pick 1de2076 add cat 2
pick ca40fc9 add 2 cats
```

```
pick 2bab3e7 add dog 1
pick 27f6ed6 add dog 2
```

如果我想把跟 dog 有關的 Commit 都刪掉，只要把最後兩行刪掉：

```
pick 382a2a5 add database settings
pick cd82f29 add cat 1
pick 1de2076 add cat 2
pick ca40fc9 add 2 cats
```

存檔、離開後便開始進行 Rebase 指令：

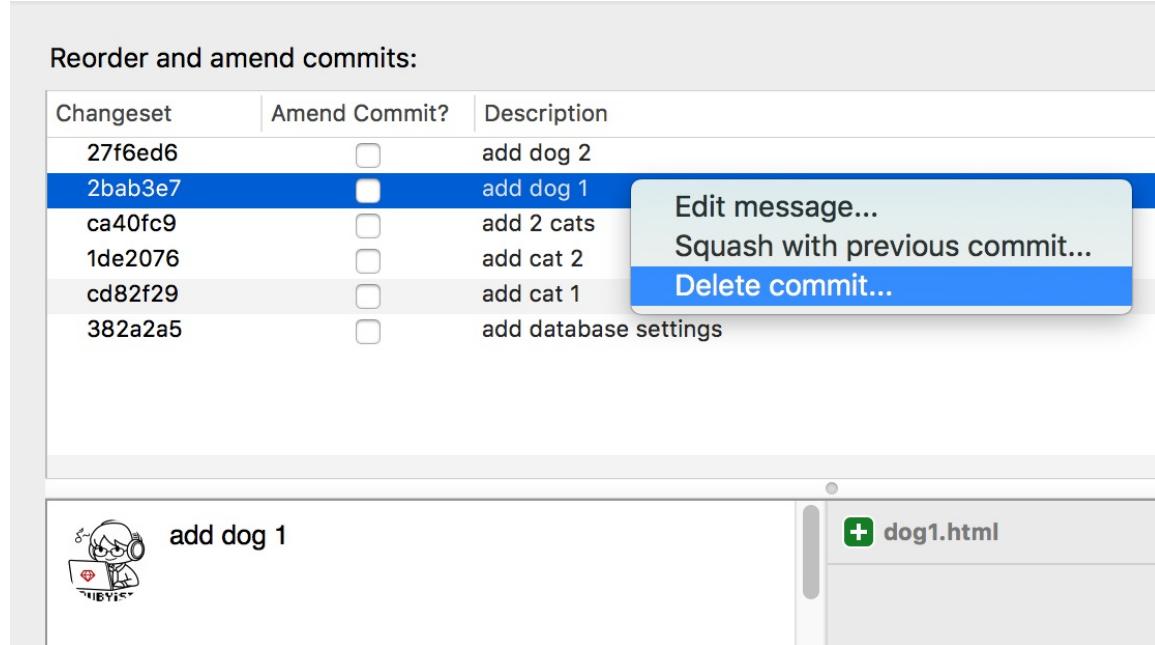
```
$ git rebase -i bb0c9c2
Successfully rebased and updated refs/heads/master.
```

來看一下現在的 Commit 紀錄：

```
$ git log --oneline
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

剛剛那 2 個 Commit 就被刪除了。

使用 SourceTree 的話，一樣先叫出 Rebase 視窗，在打算刪除的 Commit 上按滑鼠右鍵，選擇「Delete commit...」功能：



可以把打算刪除的都先標記起來：

Reorder and amend commits:

Changeset	Amend Commit?	Description
27f6ed6	<input type="checkbox"/>	add dog 2
2bab3e7	<input type="checkbox"/>	add dog 1
ca40fc9	<input type="checkbox"/>	add 2 cats
1de2076	<input type="checkbox"/>	add cat 2
cd82f29	<input type="checkbox"/>	add cat 1
382a2a5	<input checked="" type="checkbox"/>	add database settings

按下 OK 鈕之後便會開始執行 Rebase 指令：

Graph	Description	Commit	Author
master	add 2 cats	ca40fc9	Eddie Kao <eddie>
	add cat 2	1de2076	Eddie Kao <eddie>
	add cat 1	cd82f29	Eddie Kao <eddie>
	add database settings	382a2a5	Eddie Kao <eddie>
	init commit	bb0c9c2	Eddie Kao <eddie>

剛剛標記成刪除的 Commit 就被刪掉了。

後遺症？

那些穿越時空電影的劇情都會警告時空旅行者不要隨便修改歷史，想想看，如果你搭時光機回到你爸媽那個年代，然後想辦法拆散他們兩人，然後你還會是你嗎？

同樣的，不管是調整 Commit 的順序，或是刪除某個 Commit，都一樣要注意相依性問題。例如某個 Commit 修改 `index.html` 的內容，結果你把這個 Commit 移到建立 `index.html` 的那個 Commit 之前；或是你刪除了某個建立 `welcome.html` 檔案的 Commit，但後面的 Commit 都需要 `welcome.html` 這個檔案...像這樣的操作一定會出問題的，使用 Rebase 指令的時候要特別注意。

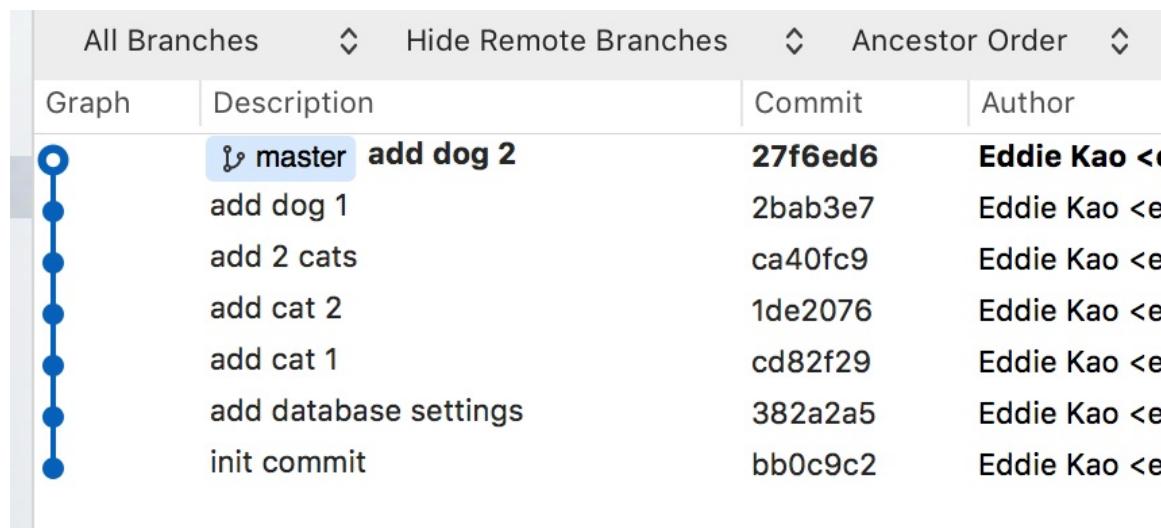
Reset、Revert 跟 Rebase 指令有什麼差別？

在進入這個主題之前，先看一下 Revert 這個指令。

如果想要拆除已經完成的 Commit，在「[【狀況題】剛才的 Commit 後悔了，想要拆掉重做...](#)」章節介紹過可使用 Reset 指令來處理，在前幾章也有介紹過使用 Rebase 來修改歷史紀錄，例如把多個 Commit 併成一個、把一個拆成多個、刪除 Commit、調整 Commit 順序或是在指定的 Commit 之間再加入新的 Commit 等，接下來要介紹的 Revert 指令，也是另一可以用來「後悔」的指令。

使用 Revert 指令

來看一下怎麼使用 Revert 指令。假設目前的歷史紀錄是這樣：



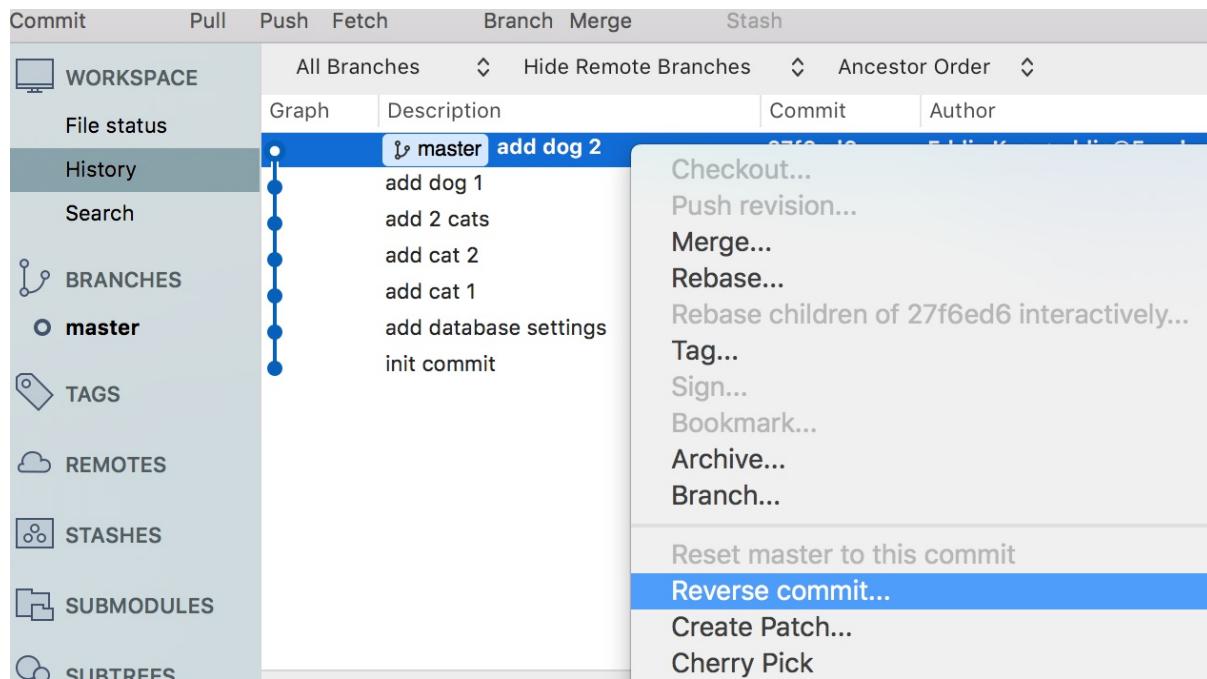
The screenshot shows a SourceTree interface with a commit history table. The table has four columns: Graph, Description, Commit, and Author. The commits are listed from newest at the top to oldest at the bottom. The 'Graph' column shows a vertical line of circles representing the commit chain, with the last commit ('add dog 2') highlighted in blue.

All Branches	Hide Remote Branches	Ancestor Order	
Graph	Description	Commit	Author
	master add dog 2	27f6ed6	Eddie Kao <e...
	add dog 1	2bab3e7	Eddie Kao <e...
	add 2 cats	ca40fc9	Eddie Kao <e...
	add cat 2	1de2076	Eddie Kao <e...
	add cat 1	cd82f29	Eddie Kao <e...
	add database settings	382a2a5	Eddie Kao <e...
	init commit	bb0c9c2	Eddie Kao <e...

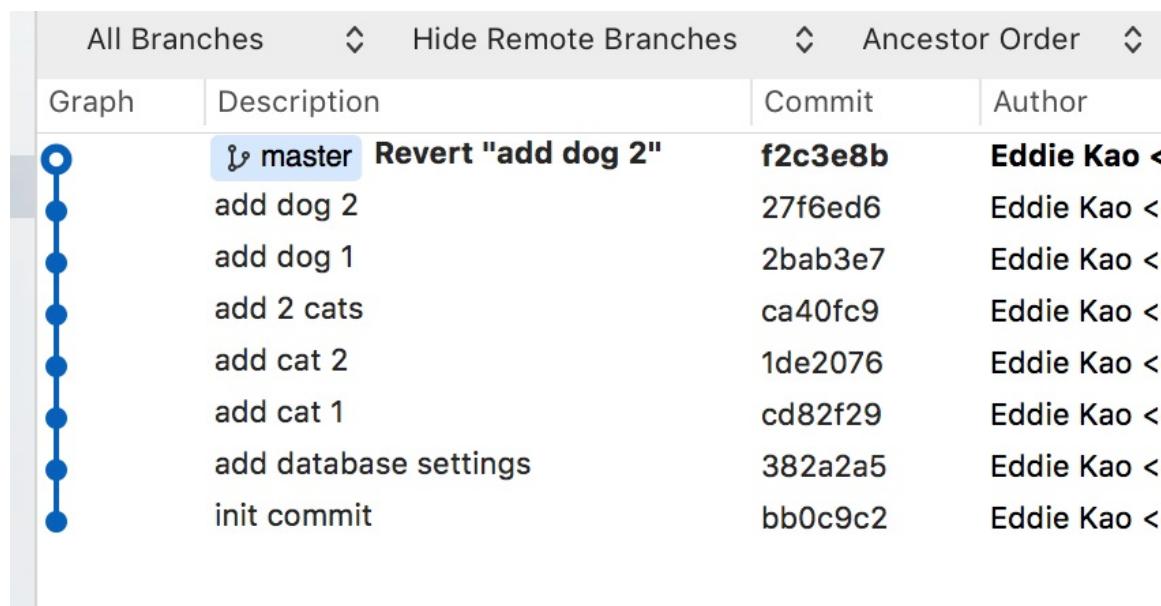
如果想要取消最後這次的 Commit（`add dog 2`），可以這樣做：

```
$ git revert HEAD --no-edit
[master f2c3e8b] Revert "add dog 2"
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 dog2.html
```

加上後面的 `--no-edit` 參數，表示不編輯 Commit 訊息。使用 SourceTree 可以在想要取消的 Commit 上按滑鼠右鍵，然後選擇「Revert Commit...」



這樣就把最後一次 Commit 的內容給刪掉了...等等，檔案是不見了沒錯，但是好像有點怪怪的：



Commit 怎麼增加了？而且原來的那個 `add dog 2` 的 Commit 也還在？原來，Revert 的指令是「再做一個新的 Commit，來取消你不要的 Commit」的概念，所以 Commit 數量才會增加。

那怎麼取消 Revert？

如果做出來的這個 Revert 你也不想要了，有幾個方式來處理它：

再開一次 Revert

是的，你可以再開一個新的 Revert，來 Revert 剛剛這個 Revert（有點饒口）：

```
$ git revert HEAD --no-edit
[master e209455] Revert "Revert "add dog 2"""
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dog2.html
```

但剛剛被刪掉的 `dog2.html` 又復活了，但這樣 Commit 又變多了…

直接使用 Reset

如果是要「砍掉」這個 Revert，其實只要直接使用 Reset 指令就行了：

```
$ git reset HEAD^ --hard
```

這樣就可以回到 Revert 之前的狀態了。

什麼時候使用 Revert 指令？

如果是自己一個人做的專案，用 Revert 指令其實有點過於「禮貌」了，大部份都是直接使用 Reset 就好。但如果對於多人共同協作的專案，也許因為團隊開發的政策，你不一定有機會可以使用 Reset 指令，這時候就可以 Revert 指令來做出一個「取消」的 Commit，對其它人來說也不算是「修改歷史」，而是新增一個 Commit，只是剛好這個 Commit 是跟某個 Commit 反向的操作而已。

所以，這三個指令有什麼差別？

用個表格歸納一下：

指令	改變歷史紀錄	說明
Reset	是	把目前的狀態設定成某個指定的 Commit 的狀態，通常適用於尚未推出去的 Commit。
Rebase	是	不管是新增、修改、刪除 Commit 都相當方便，用來整理、編輯還沒有推出去的 Commit 相當方便，但通常也只適用於尚未推出去的 Commit。
Revert	否	新增一個 Commit 來反轉（或說取消）另一個 Commit 的內容，原本的 Commit 依舊還是會保留在歷史紀錄中。雖然會因此而增加 Commit 數，但通常比較適用於已經推出去的 Commit，或是不允許使用 Reset 或

Rebase 之修改歷史紀錄的指令的場合。

使用標籤

標籤是什麼？

在 Git，「標籤（tag）」是一個指向某一個 Commit 的指標。咦？這好像跟分支（Branch）一樣不是嗎？是的，他們還滿像的，但也有一些不太一樣的地方，在下一章「[【冷知識】標籤跟分支有什麼不一樣？](#)」會再另外說明。

什麼時候使用標籤？

通常在開發軟體有完成特定的里程碑，例如軟體版號 `1.0.0` 或是 `beta-release` 之類的，這時候就很適合使用標籤做標記。

標籤有兩種

跟斯斯一樣，標籤也有兩種，一種是輕量標籤（lightweight tag），一種是有附註標籤（annotated tag），但不管哪一種，都可以把它當做貼紙一樣看待，它就是貼在某個 Commit 上的東西。

輕量標籤（lightweight tag）

輕量標籤的使用方法相當簡單，只要直接指定打算貼上去的那個 Commit 紿它就行了。假設目前的 Commit 紀錄如下：

```
$ git log --oneline
db3bbec (HEAD -> master) add fish
930feb3 add pig
51d54ff add lion and tiger
27f6ed6 add dog 2
2bab3e7 add dog 1
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

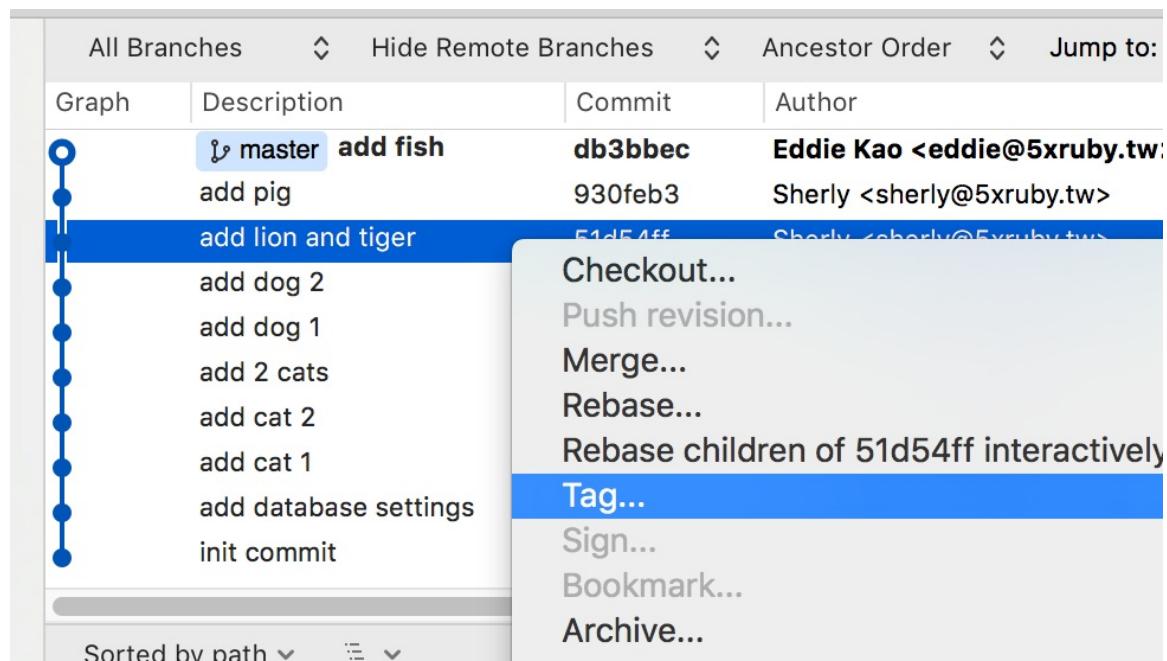
我想在 `add lion and tiger` 這個 Commit (`51d54ff`) 打上一個 `big_cats` 的標籤：

```
$ git tag big_cats 51d54ff
```

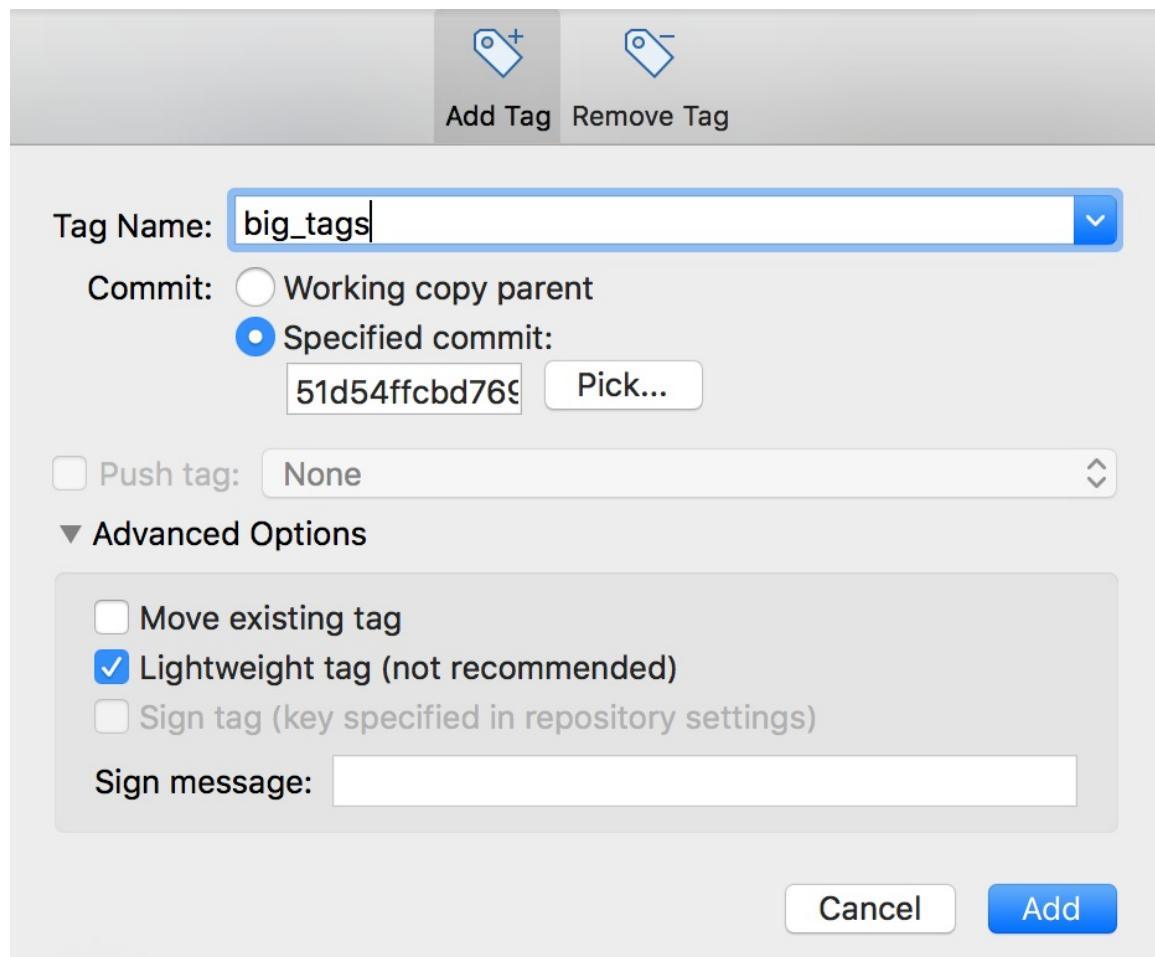
這樣就貼好標籤了！如果只使用 `git tag big_cats` 而沒有加上後面 Commit 的 SHA-1 值，會把標籤貼在目前所在的這個 Commit 上。看看現在的狀態：

```
$ git log --oneline
db3bbec (HEAD -> master) add fish
930feb3 add pig
51d54ff (tag: big_cats) add lion and tiger
27f6ed6 add dog 2
2bab3e7 add dog 1
ca40fc9 add 2 cats
1de2076 add cat 2
cd82f29 add cat 1
382a2a5 add database settings
bb0c9c2 init commit
```

可以看到在有一個標籤 `big_cats` 指向 `51d54ff` 這個 Commit。如果使用 SourceTree 來操作，可以在你打貼標籤的地方按滑鼠右鍵，選擇「Tag...」功能：



填上標籤的名字，但這邊需要特別點開下方的「Advanced Options」，展開後有一項「Lightweight tag (not recommended)」的選項打勾：



這樣就能建立一個輕量標籤了。因為輕量標籤僅是一個指向某個 Commit 的指標，沒有含有其它的資訊，所以 Git 比較推薦使用有附註的標籤（annotated tag）。

有附註標籤（annotated tag）

以上面的 `add lion and tiger` 那個 Commit (`51d54ff`) 為例，如果要建立一個有附註的標籤：

```
$ git tag big_cats 51d54ff -a -m "Big Cats are comming"
```

這樣就行了，那個 `-a` 參數就是請 Git 幫你建立有附註的標籤，而後面的 `-m` 則是跟我們在做一般的 Commit 一樣輸入的訊息，如果沒有使用 `-m` 參數，會自動跳出一個 Vim 編輯器出來。而在 SourceTree 上要加上有附註的標籤也很容易，跟一般的輕量標籤一樣的流程，但不要勾選「Lightweight tag (not recommended)」的選項就好了。

這是 Git 官方文件對這兩種標籤的說明：

Annotated tags are meant for release while lightweight tags are meant for private or temporary object labels.

有附註標籤主要用來做像是軟體版號之類的用途，而輕量標籤則是用來個人使用或是暫時標記用途。簡單的說，有附註標籤的好處就是有更多關於這張標籤的資訊，例如是誰在什麼時候貼這張標籤，以及為什麼要貼這張標籤。所以如果你不是很在乎這些資訊，用一般的輕量標籤也是沒什麼問題。

兩種標籤的差別

這兩種標籤的第一個差別，就是訊息量的不同，如果是一般的輕量標籤：

```
$ git show big_cats
commit 51d54ffcbd76902f2f580cf5638305eaaf6acde5
Author: Sherly <sherly@5xruby.tw>
Date:   Tue Aug 22 01:10:54 2017 +0800

    add lion and tiger

diff --git a/cute_animals/lion.html b/cute_animals/lion.html
new file mode 100644
index 0000000..e69de29
diff --git a/cute_animals/tiger.html b/cute_animals/tiger.html
new file mode 100644
index 0000000..e69de29
```

只有標籤指向的那個 Commit 的訊息。而這是有附註的標籤：

```
$ git show big_cats

tag big_cats
Tagger: Eddie Kao <eddie@5xruby.tw>
Date:   Tue Aug 22 03:39:37 2017 +0800

Big Cats are comming

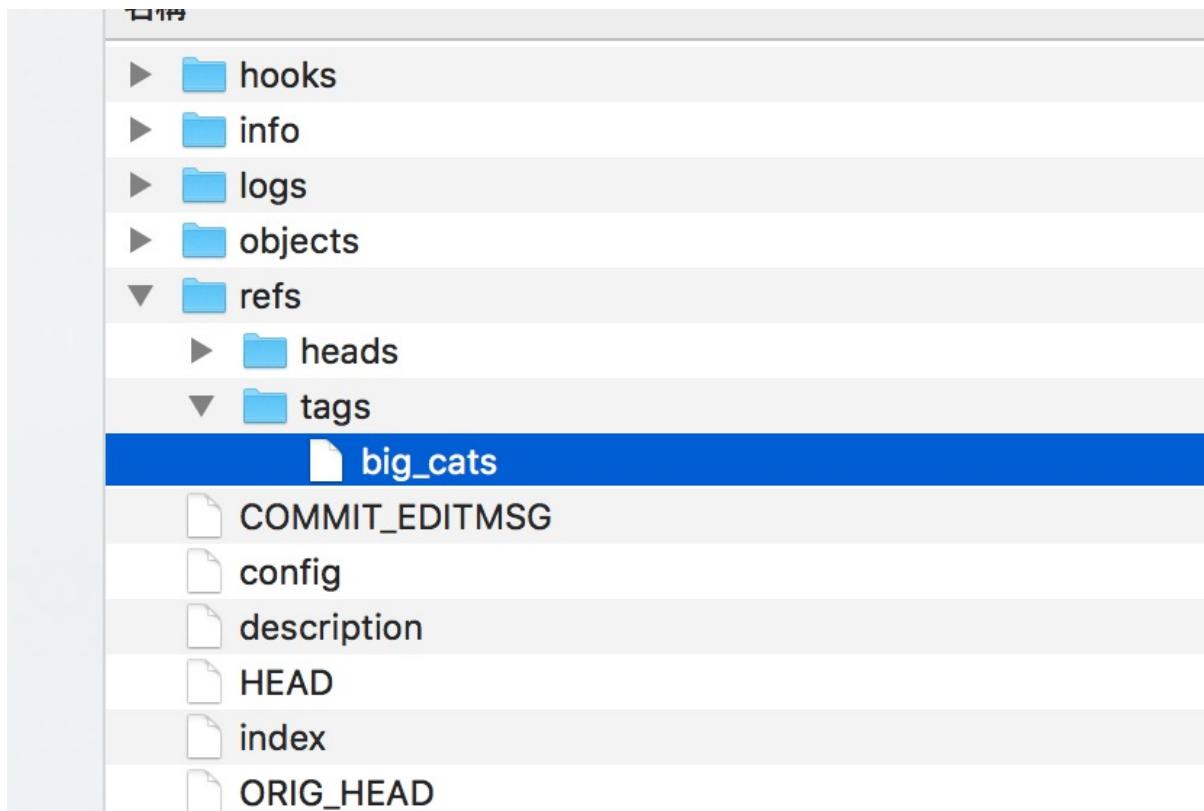
commit 51d54ffcbd76902f2f580cf5638305eaaf6acde5
Author: Sherly <sherly@5xruby.tw>
Date:   Tue Aug 22 01:10:54 2017 +0800

    add lion and tiger

diff --git a/cute_animals/lion.html b/cute_animals/lion.html
new file mode 100644
index 0000000..e69de29
diff --git a/cute_animals/tiger.html b/cute_animals/tiger.html
new file mode 100644
index 0000000..e69de29
```

有附註的標籤比一般的輕量標籤多了一些資訊，可以清楚的看得出來是誰在什麼時候打了這張標籤。

不管是哪種標籤，跟分支一樣都是以檔案方式存在 `.git/refs/tags` 目錄下：



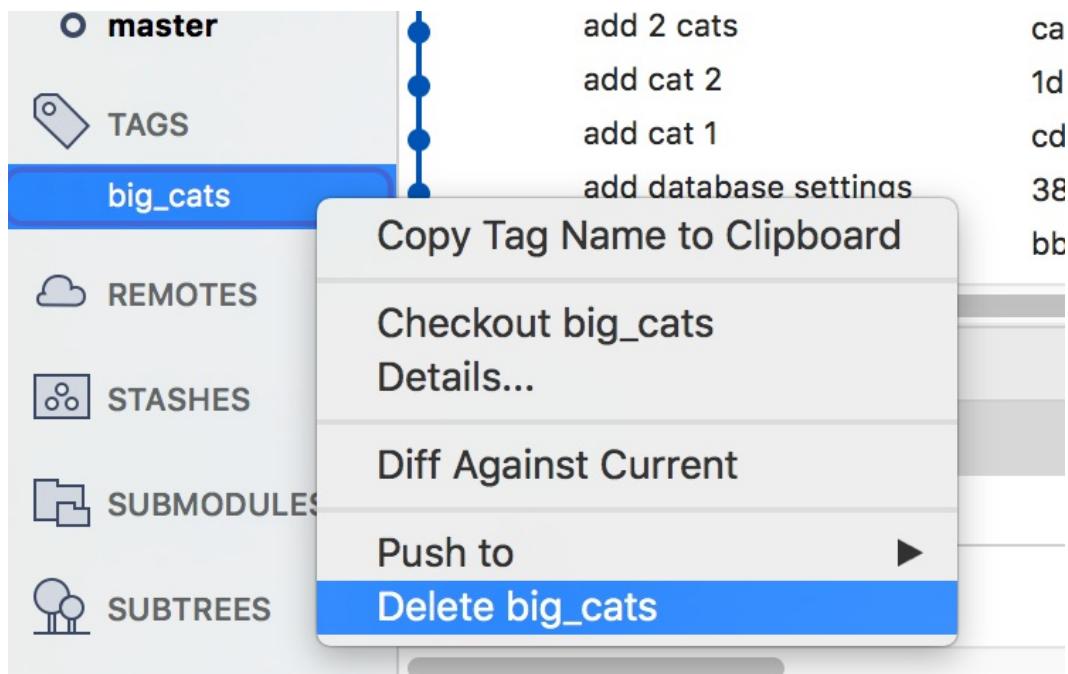
檔案的內容也跟分支一樣，是一個 40 個字元的 SHA-1 值，指向某個地方。但差別是，輕量標籤指向的是某一個 Commit，但附註標籤是指向某個 Tag 物件，而這個 Tag 物件才再指向那個 Commit。關於 Tag 物件，在「[【超冷知識】在 .git 目錄裡有什麼東西？Part 1](#)」章節有詳細說明。

刪除標籤

不管是哪一種標籤，標籤基本上就是一張貼紙的概念，撕掉一張貼紙並不會造成 Commit 或檔案不見。要刪除只要給它 `-d` 參數就行了：

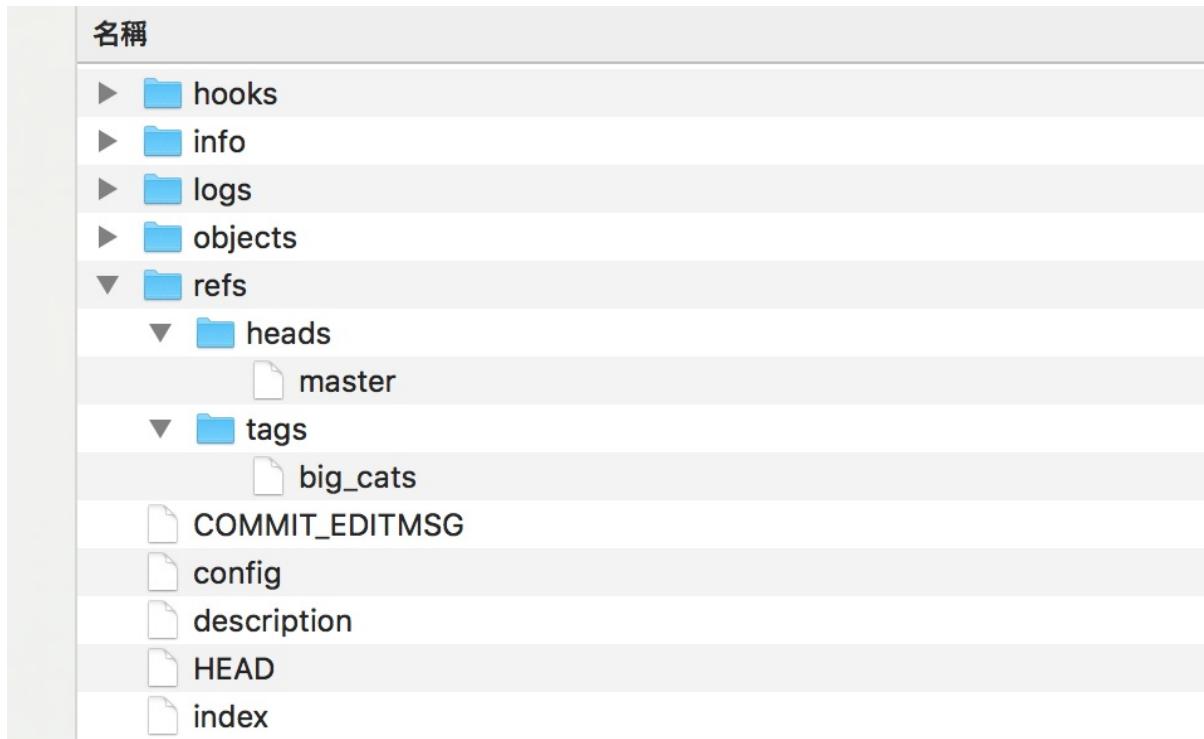
```
$ git tag -d big_cats
Deleted tag 'big_cats' (was 8ee0144)
```

使用 SourceTree 的話，則可在左邊側邊選單，找到那個分支，按滑鼠右鍵選擇最下面「Delete..」的就行了：



【冷知識】標籤跟分支有什麼不一樣？

其實標籤跟分支是很像的東西，來把它們從 .git 目錄裡翻出來看看：



標籤跟分支都是一種指標，也都是放在 `.git/refs` 目錄下，只是分支是在 `heads` 目錄，標籤則是在 `tags` 目錄。

裡面的內容也長得很像，都是 40 個字元的 SHA-1 值：

```
$ cat .git/refs/heads/master  
db3bbec63301d1c638e828c9a38a29314c8a0c44  
  
$ cat .git/refs/tags/big_cats  
552a844022bad7f24c5e6e3b0fc2528c8ec86df7
```

在被刪除的時候，也不會影響到被指到的那個物件。

其實標籤跟分支真正的差別，是「分支會隨著 Commit 而移動，但標籤不會」。在「[對分支的誤解](#)」章節曾介紹過，當 Git 往前推進一個 Commit 的時候，它所在的分支會跟著往前移動。但標籤一旦貼上去之後，不管 Commit 怎麼前進，標籤還是留在原來貼的那個位置上。

所以你可以把分支看成「會移動的標籤」。

在電影《海角七號》有一句經典台詞：「留下來，或我跟你走」，用在這邊差不多是「留下來的是標籤，跟你走的是分支」的概念吧。

【狀況題】手邊的工作做到一半，臨時要切換到別的任務

在公司工作或多或少有經歷過這樣的情境：

你手邊的工作做到一半...

老闆：「那個誰誰誰，網站掛了，你趕快先來修一下這個功能...」

先不管心情好不好，既然老闆都開口了，為了生活只好暫時先乖乖聽話，先把手邊的進度放旁邊...

那就先 Commit 目前的進度吧！

簡單的做法，也是我常會做的做法，就是先不管那麼多，先把目前所有的修改先存下來。假設這是目前的狀況，而且我正在 `cat` 分支進行功能開發：

Graph	Description	Commit
●	↳ dog add dog 2	053fb21
●	add dog 1	b69eb62
●	↳ cat add cat 2	b174a5a
●	add cat 1	c68537b
●	↳ master add database.yml in config folder	e12d8ef
●	add hello	85e7e30
●	add container	657fce7
●	update index page	abb4f43
●	create index page	cef6e40
●	init commit	cc797cd

```
$ git add --all
$ git commit -m "not finish yet"
[cat 9bf1f43] not finish yet
 2 files changed, 1 insertion(+)
  create mode 100644 cat3.html
```

然後就可以切到有問題的分支先進行功能修復，待完成之後再切回原來做一半的 `cat` 分支，然後再 Reset 一下，把剛剛做一半的東西拆回來繼續做：

```
$ git reset HEAD^
```

然後就可以接著繼續做了。

使用 Stash

遇到這個情況，剛剛那個先 Commit，之後再 Reset 回來的做法是一種做法，另一種做法是使用 Git 的 Stash 功能。先看一下目前的狀態：

```
$ git status
On branch cat
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   cat1.html
    modified:   cat2.html
    modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

目前的狀態正在修改 `cat1.html`、`cat2.html` 以及 `index.html` 這幾個檔案。因為老闆急著召喚要去做別的事，這時候可使用 `git stash` 指令，把這些修改先「存」起來：

```
$ git stash
Saved working directory and index state WIP on cat: b174a5a add cat 2
```

注意！

Untracked 狀態的檔案預設沒辦法被 Stash，需要額外使用 `-u` 參數。

看一下目前的狀態：

```
$ git status
On branch cat
nothing to commit, working tree clean
```

好像跟剛 Commit 完一樣的乾淨了。

剛剛那些檔案存到哪去了？讓我們看一下：

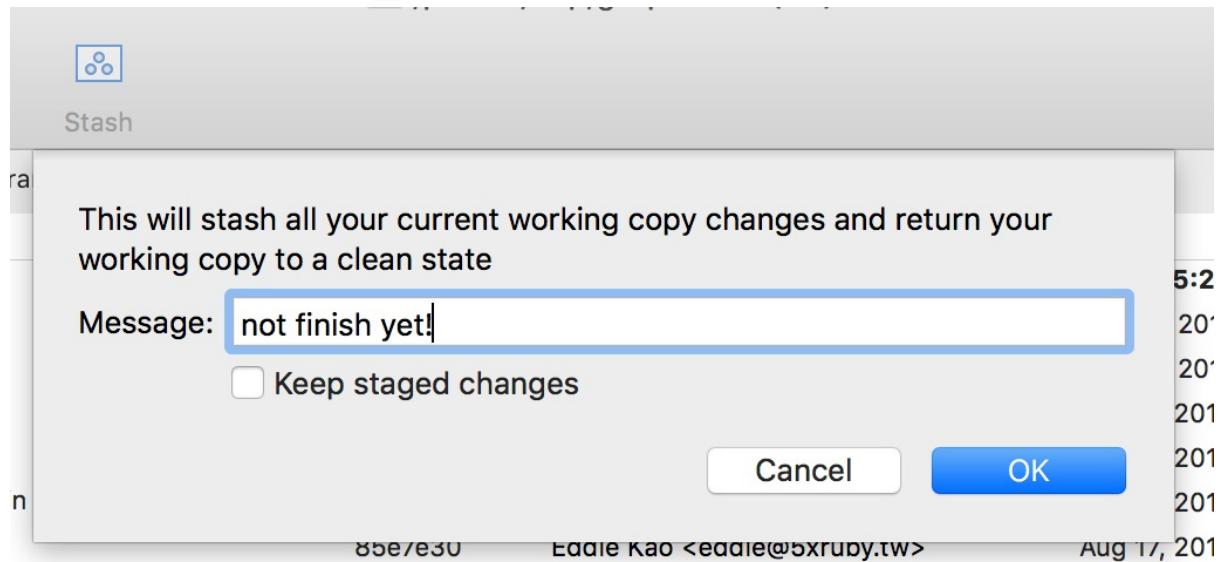
```
$ git stash list
stash@{0}: WIP on cat: b174a5a add cat 2
```

看起來目前只有一份狀態被存起來，最前面的 `stash@{0}` 是這個 Stash 的代名詞，而後面的 `WIP` 字樣是「Work In Progress」，就是工作進行中的意思。Stash 可以放很多份，例如我再放一份到 Stash：

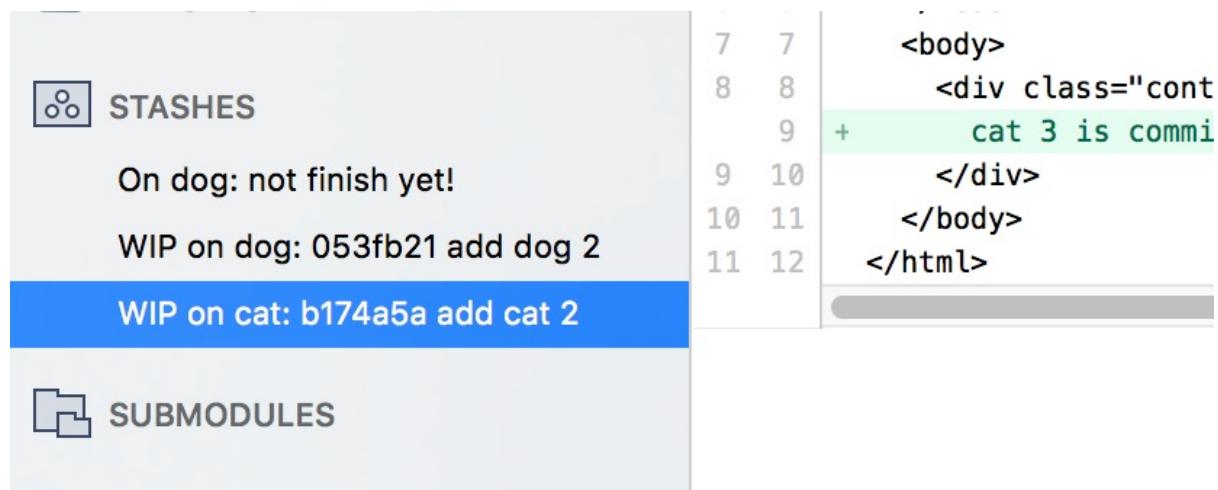
```
$ git stash list  
stash@{0}: WIP on dog: 053fb21 add dog 2  
stash@{1}: WIP on cat: b174a5a add cat 2
```

剛剛原來 `stash@{0}` 的現在變成 `stash@{1}` 了。

使用 SourceTree 也可以做這件事，在上方選單有一顆「Stash」按鈕，按下之後便會出現對話框



填寫後按下 OK 即可完成。在左邊側邊欄可以看到一個「STASHES」選單，裡面會放著目前所有的 Stash 的內容：



把 Stash 撿回來用

好啦，剛剛的修復任務完成了，可以繼續把剛剛存起來的東西再拿出來了。再次看一下目前的 Stash 列表：

```
$ git stash list
stash@{0}: On dog: not finish yet!
stash@{1}: WIP on dog: 053fb21 add dog 2
stash@{2}: WIP on cat: b174a5a add cat 2
```

這個 `stash@{2}` 應該是我剛剛最一開始做一半的進度，所以我要把它撿回來做：

```
$ git stash pop stash@{2}
On branch cat
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   cat1.html
    modified:   cat2.html
    modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{2} (80091001b2022e0fb3f8c7ee6cffcefa207d00be)
```

說明如下：

1. 使用 `pop` 指令，可以把某個 Stash 拿出來並套用在目前的分支上。套用成功之後，那個套用過的 Stash 就會被刪除。
2. 如果後面沒有指定要 pop 哪一個 Stash，會從編號最小的，也就是 `stash@{0}` 開始拿（也就是最後疊上去的那次）。

如果那個 Stash 確定不要，可以使用 `drop` 指令：

```
$ git stash drop stash@{0}
Dropped stash@{0} (87390c02bbfc8cf7a38fb42f6f3a357e51ce6cd1)
```

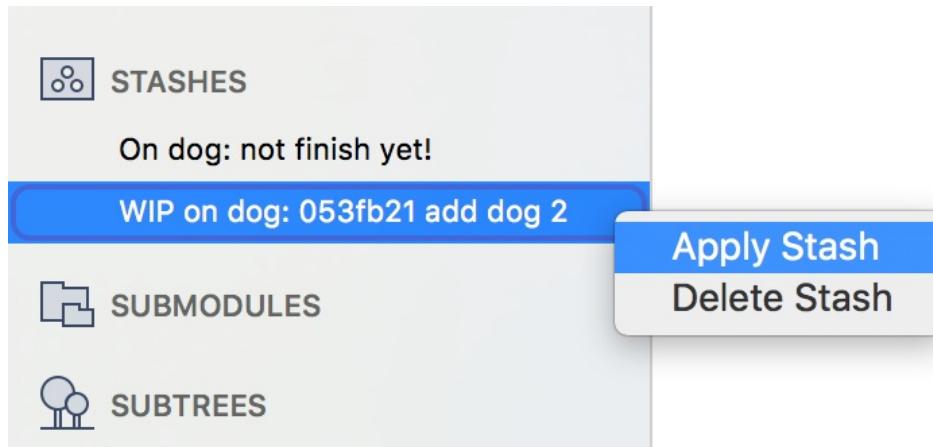
這樣就可以把那個 Stash 從列表裡刪掉了。

要把 Stash 撿回來用，除了 `pop` 之外，另一個指令是 `apply`：

```
$ git stash apply stash@{0}
```

這是會把 `stash@{0}` 這個 Stash 拿來套用在現在的分支上，但 Stash 不會刪除，還是會留在 Stash 列表上。所以你可把 `pop` 指令看成「`apply Stash + drop Stash`」。

使用 SourceTree 要套用 Stash 的話，就是在指定的 Stash 上按滑鼠右鍵，然後選擇「`Apply Stash`」功能即可：



所以，使用哪一種比較好？

其實都可，看每個人的習慣，還有對 Git 的熟悉程度會有不同的答案。如果你問我，我會跟你說我比較喜歡第一種，就是先 Commit 之後再 Reset 回來繼續做，對我個人而言，我覺得這樣相對比使用 Stash 來得直覺一些。

【狀況題】不小心把帳號密碼放在 Git 裡了，想把它刪掉...

你不小心把帳號密碼寫在某次的 Commit 裡，而且已經 Push 出去了...

如果已經 Push 出去，請不要先想要怎麼用 Git 來解決，請先直接改密碼再說！

改完密碼了嗎？那接下來我們再來看看怎麼解決這件事吧。

砍掉重練大法

這招是屬於「逃避雖然可恥但是有用」的做法，把因為所有的 Commit 紀錄都在 `.git` 目錄裡，所以可以這樣：

1. 把 `.git` 目錄刪掉。
2. 把那個密碼檔刪掉或修掉。
3. 重新 Commit。

不要以為我是在開玩笑，這也是一招，而且是比較不需要什麼技術的招式，缺點就是之前的 Commit 紀錄都不見了。所以這個專案如果只有你自己一個人做，而且也不在乎之前的 Commit 紀錄的話，這也是一個選擇。

使用 `filter-branch` 指令

如果你不想使用砍掉重練大法，另一個選項就是使用 Git 的 `filter-branch` 指令。假設目前的 Commit 紀錄是這樣：

Graph	Description	Commit	Author
	↳ master add fish	db3bbec	Eddie Kao <eddie@5...
	add pig	930feb3	Sherly <sherly@5...
	add lion and tiger	51d54ff	Sherly <sherly@5...
	add dog 2	27f6ed6	Eddie Kao <eddie@5...
	add dog 1	2bab3e7	Eddie Kao <eddie@5...
	add 2 cats	ca40fc9	Eddie Kao <eddie@5...
	add cat 2	1de2076	Eddie Kao <eddie@5...
	add cat 1	cd82f29	Eddie Kao <eddie@5...
	add database settings	382a2a5	Eddie Kao <eddie@5...
	init commit	bb0c9c2	Eddie Kao <eddie@5...

在這個例子裡，我從第二個 Commit (382a2a5) 開始就已經把資料庫的密碼加進去了。

假設我想要把 config/database.yml 這個檔案從每個 Commit 裡把它拿掉，比較直覺但稍微比較辛苦的做法是使用 Rebase 指令，然後一個一個 Commit 去編輯。Git 有一個叫做 filter-branch 的指令，這個指令不算太常見，但它可以一次大量的修改 Commit。

```
$ git filter-branch --tree-filter "rm -f config/database.yml"
Rewrite db3bbec63301d1c638e828c9a38a29314c8a0c44 (9/10) (1 seconds passed, remaining 0 predicted)
Ref 'refs/heads/master' was rewritten
```

說明：

1. filter-branch 這個指令可以讓你根據不同的 filter，一個一個 Commit 的來處理它。
2. 這裡使用了 --tree-filter 這個 filter，它可以讓你在 Checkout 到每個 Commit 的時候執行你指定的指令，執行完後再自動幫你重新再 Commit。以上面這個例子來說，便是執行「強制刪除 config/database.yml 檔案」這個指令。
3. 因為刪除了某個檔案，所以在那之後的 Commit 全部都會重新計算，也就是說等於產生一份新的歷史紀錄了。

東西一旦加到 Git 裡，真的要把它刪掉其實沒那麼容易，我們在下一章「【冷知識】怎麼樣把檔案真正的從 Git 裡移掉？」章節會再做更多的說明。

啊！我又後悔了，怎麼回復剛剛 filter-branch 造成的效果？

使用 Git 的好處，就是後悔隨時可以重來。其實在進行 `filter-branch` 指令的時候，Git 會幫你把之前的狀態備份一份在 `.git/refs/original/refs/heads` 這個目錄裡（其實說是備份，也只是備份開始進行 `filter-branch` 之前的那個 HEAD 的 SHA-1 值而已）。所以你可以從這個檔案把 SHA-1 值找出來然後再 hard Reset 回去，或是直接這樣也可以：

```
$ git reset refs/original/refs/heads/master --hard  
HEAD is now at db3bbec add fish
```

這樣就都回來了。

如果已經推出去了...

老實說，推出去的東西就跟潑出去的水一樣收不回來，你能做的就是使用 `git push -f` 指令，重新強制推一份你剛剛 `filter-branch` 過的 Commit 上去。在「[Push 上傳到 GitHub](#)」章節會有更多關於 Push 相關的介紹。

【狀況題】如果你只想要某個分支的某幾個 Commit ?

先來看看目前的狀況：

All Branches	Show Remote Branches	Ancestor Order	
Graph	Description	Commit	Author
fish	↳ fish add shark	fc8c4dc	Eddie Kao <eddi>
	add whale	fd23e1c	Eddie Kao <eddi>
	add dolphin	6a498ec	Eddie Kao <eddi>
	add gold fish	f4f4442	Eddie Kao <eddi>
dog	↳ dog add dog 2	053fb21	Eddie Kao <eddi>
	add dog 1	b69eb62	Eddie Kao <eddi>
cat	↳ cat add cat 2	b174a5a	Eddie Kao <eddi>
	add cat 1	c68537b	Eddie Kao <eddi>
master	↳ master add database.yml in config folder	e12d8ef	Eddie Kao <eddi>
	add hello	85e7e30	Eddie Kao <eddi>
	add container	657fce7	Eddie Kao <eddi>
	update index page	abb4f43	Eddie Kao <eddi>
	create index page	cef6e40	Eddie Kao <eddi>
	init commit	cc797cd	Eddie Kao <eddi>

說明：

1. 目前加上 `master` 共有四個分支。
2. 目前正在 `cat` 分支上。

然後我發現 `fish` 分支做得不錯，但裡面不是所有的 Commit 想要，例如我只想要 `add dolphin` 跟 `add whale` 這兩個 Commit，其它的不是很有興趣...

檢別的分支的 Commit 過來合併

Git 裡有個 `cherry-pick` 指令，可以只檢某些 Commit 來用，例如只想檢 `add dolphin` 這個 Commit (`6a498ec`) :

```
$ git cherry-pick 6a498ec
[cat 8562ee3] add dolphin
Date: Tue Aug 22 09:42:16 2017 +0800
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dolphin.html
```

檢視一下紀錄：

```
$ git log --oneline
8562ee3 (HEAD -> cat) add dolphin
b174a5a add cat 2
c68537b add cat 1
e12d8ef (master) add database.yml in config folder
85e7e30 add hello
657fce7 add container
abb4f43 update index page
cef6e40 create index page
cc797cd init commit
```

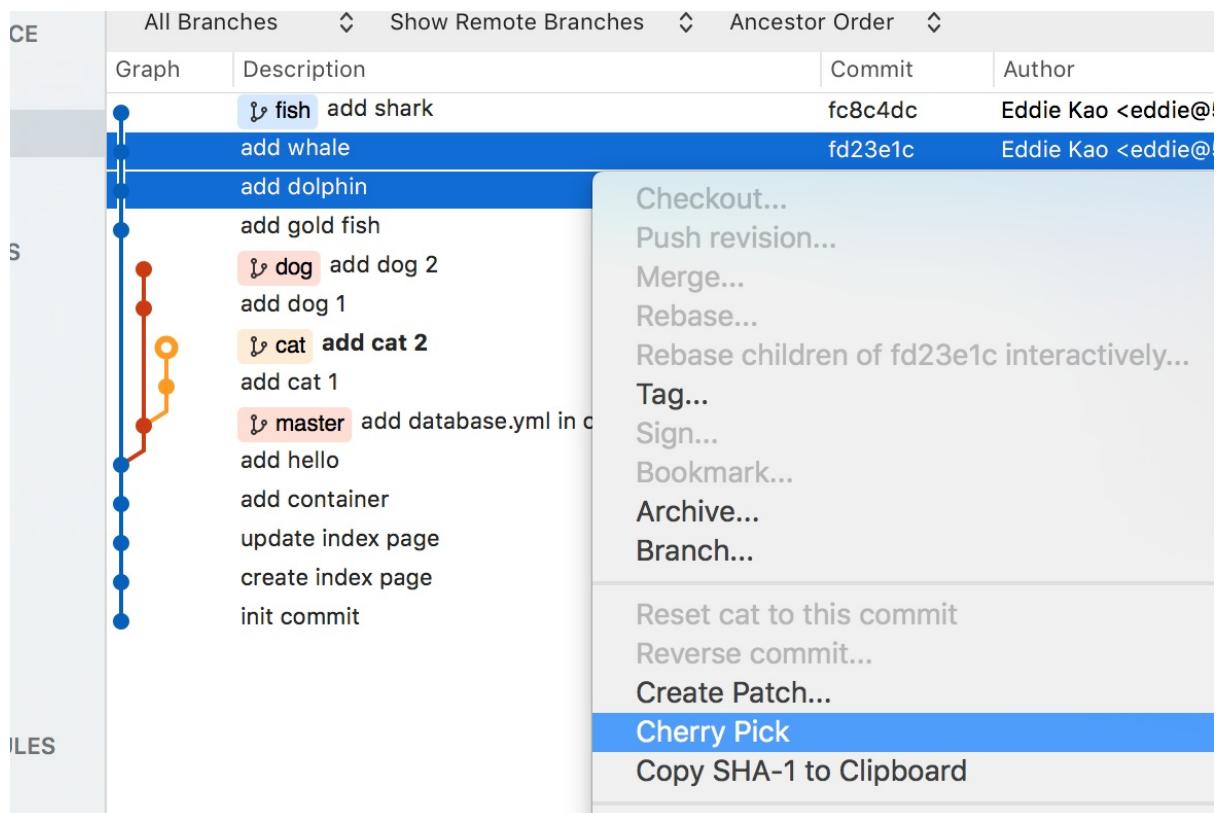
那個 `add dolphin` 就被檢進來了。當然，它並不是把原來的 Commit 剪過來這邊貼上，而是比較像是複製 Commit 的內容過來，因為要接到現有的分支上所以需要重新計算，產生一個新的 Commit。當然，原本在 `fish` 的那個 `add dolphin` 的 Commit 還是在原來的地方。

一次檢好幾個...

```
$ git cherry-pick fd23e1c 6a498ec f4f4442
[cat 5b76277] add whale
Date: Tue Aug 22 09:44:50 2017 +0800
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 whale.html
[cat de503b3] add dolphin
Date: Tue Aug 22 09:42:16 2017 +0800
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dolphin.html
[cat 6bf4b32] add gold fish
Date: Tue Aug 22 09:41:51 2017 +0800
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 gold-fish.html
```

這樣就可以一口氣檢三個 Commit 過來合併。

使用 SourceTree 做這件事也是很容易，只要先選好幾個想要檢過來的，在上面按滑鼠右鍵，選擇「Cherry Pick」功能：



按下確認鍵就完成了。

撿過來但先不合併

在使用 `cherry-pick` 指令的時候，如果加上 `--no-commit` 參數，撿過來的 Commit 不會直接合併，而是會先放在暫存區：

```
$ git cherry-pick 6a498ec --no-commit

$ git status
On branch cat
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   dolphin.html
```

【冷知識】怎麼樣把檔案真正的從 Git 裡移掉？

砍掉重練

就是把整個 .git 目錄砍掉，整理好之後再重建，但這招這個應該不是我們這邊要討論的重點。

使用 Rebase 或 filter-branch 指令來整理

如果 Commit 的數量小，使用 Rebase 應該就足以進行編輯、重整。在「[【狀況題】不小心把帳號密碼放在 Git 裡了，想把它刪掉…](#)」章節也曾介紹過 `filter-branch` 指令，它可以大範圍的對每個 Commit 執行某個指令，並於修改完成後會自動再重新 Commit。所以如果是要把檔案從 Git 裡拔掉，用 `filter-branch` 指令應該是相對較方便的。

舉例來說，目前的 Commit 紀錄是這樣：

All Branches	Show Remote Branches	Ancest
Graph	Description	Commit
○	master add dog 2	27f6ed6
●	add dog 1	2bab3e7
●	add 2 cats	ca40fc9
●	add cat 2	1de2076
●	add cat 1	cd82f29
●	add database settings	382a2a5
●	init commit	bb0c9c2

我想要把所有 Commit 的 `config/database.yml` 這個檔案刪掉：

```
$ git filter-branch --tree-filter "rm -f config/database.yml"
Rewrite 27f6ed6da50dbe5adbb68102266a91dc097ad3f (7/7) (0 seconds passed, remaining 0 predicted)
Ref 'refs/heads/master' was rewritten
```

這樣看起來好像刪掉了...假的，其實那些東西都還在，隨時都可以取消剛剛這個指令，把剛剛被刪的檔案救回來：

```
$ git reset refs/original/refs/heads/master --hard  
HEAD is now at 27f6ed6 add dog 2
```

全部斷乾淨！

再重頭來一次：

```
$ git filter-branch -f --tree-filter "rm -f config/database.yml"  
Rewrite 27f6ed6da50dbe5adbb68102266a91dc097ad3f (7/7) (1 seconds passed, remaining 0 predicted)  
Ref 'refs/heads/master' was rewritten
```

跟前面不太一樣，這次多加了 `-f` 參數，是因為要強制覆寫 filter-branch 的備份點。這邊使用 `filter-branch` 指令把檔案從工作目錄裡移掉，這時候 `database.yml` 的確已不見，但還有好幾個跟資源回收有關的事情需要處理一下：

```
$ rm .git/refs/original/refs/heads/master
```

這個檔案還對剛剛做的 `filter-branch` 動作念念不忘（也就是備份點啦），隨時可以透過它再跳回去，所以先斷這條線。

再來，念念不忘的還有 Reflog，所以它也要清一下：

```
$ git reflog expire --all --expire=now
```

這個指令是要求 Reflog 現在立刻過期（不然預設要等 30 天）。接著再用 `git fsck` 指令就可以看到很多 Unreachable 的物件了：

```
$ git fsck --unreachable  
Checking object directories: 100% (256/256), done.  
unreachable tree c8da8b6accf7029a2fb89eed130365822692b603  
unreachable commit ca40fc9b31c777b1d3434453448c945fa2ffae11  
unreachable commit cd82f29acbdce60c7f5f6894619585bd445797b5  
unreachable tree 9e941fe91d47bf5174bd5a3d3e73ff257598b0ca  
unreachable tree 5e01e02411507c504c77bca53c508a3174c9a06f  
unreachable tree 607f055180d1195c81e0534d264d131d5abfdc27  
unreachable commit 1de207637a6eed2cc86507dca37a38c7a932e53c  
unreachable tree a21100f9f3aae37858cc84fd402663992ccca681  
unreachable commit 27f6ed6da50dbe5adbb68102266a91dc097ad3f  
unreachable tree a618ce33da8d21bca841f18e6432fcabf15d4477  
unreachable commit 2bab3e7aff03a30ed9f53b5a7d3e02e1c0fc8c7c  
unreachable tree 70c6b4db190a452b22c28998d7c2487efb8026b2  
unreachable commit 382a2a5cec96b94e9c5cb42bf92b4b236f4ad8ac
```

最後，啟動 Git 的資源回收機制，請垃圾車來立刻把它們載走：

```
$ git gc --prune=now
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), done.
Total 14 (delta 5), reused 0 (delta 0)
```

檢查一下：

```
$ git fsck
Checking object directories: 100% (256/256), done.
Checking objects: 100% (14/14), done.
```

看來垃圾都載走了。我們試試看用偷吃步能不能 Reset 得回去：

```
$ git reset 27f6ed6 --hard
fatal: ambiguous argument '27f6ed6': unknown revision or path not in the working tree.
Use '--' to separate paths from revisions, like this:
'git <command> [<revision>...] -- [<file>...]'
```

看來不行了，Git 已經找不到原來的那個 SHA-1 值了。

提醒一下，如果這些內容已經推出去的話，別忘了最後再加一步 `git push -f` 把線上的紀錄蓋掉喔。

小結

檔案進了 Git 就跟得罪方丈一樣，想走沒那麼容易，需要全部都斷乾淨才行。關於 Git 的資源回收的相關介紹，請參閱「[【冷知識】你知道 Git 有資源回收機制嗎？](#)」章節。

【冷知識】你知道 Git 有資源回收機制嗎？

在 Git 裡，每當把檔案加至暫存區的時候，Git 便會根據檔案內容製作出 Blob 物件；每當完成 Commit，便會跟著產生所需的 Tree 物件以及 Commit 物件。（細節可參閱「[【超冷知識】在.git 目錄裡有什麼東西？Part 1](#)」章節）。

隨著物件越來越多，當條件滿足的時候，Git 會自動觸發資源回收機制（Garbage Collection）來整理這些物件，同時也會把 Unreachable 狀態的物件清掉。這樣除了讓這些備份的檔案體積縮小外，也讓物件的檢索更有效率。

讓它變成沒人愛的邊緣人！

實際操作一次，看看這個回收機制是怎麼運作的。首先，我先開一個專案，然後很簡單的做三次 Commit：

```
$ echo "cat 1" > cat1.html  
$ git add cat1.html  
$ git commit -m "add cat 1"  
[master (root-commit) 654b2fc] add cat 1  
 1 file changed, 1 insertion(+)  
 create mode 100644 cat1.html  
  
$ echo "cat 2" > cat2.html  
$ git add cat2.html  
$ git commit -m "add cat 2"  
[master f6bba64] add cat 2  
 1 file changed, 1 insertion(+)  
 create mode 100644 cat2.html  
  
$ echo "cat 3" > cat3.html  
$ git add cat3.html  
$ git commit -m "add cat 3"  
[master eb396df] add cat 3  
 1 file changed, 1 insertion(+)  
 create mode 100644 cat3.html
```

現在有三次 Commit 了。因為接下我打算使用 `git reset` 來取消最新一次的 Commit，在砍之前我想先看一下這個 Commit 最後一眼：

```
$ git cat-file -p eb396dfbdd7ab4ca6618fe643ada8cf60ca67472
tree 4c0c24949985ce2c87d99860b27791984d183d44
parent f6bba648073582fe4c361a8eadb088222ee3bb70
author Eddie Kao <eddie@5xruby.tw> 1503604211 +0800
committer Eddie Kao <eddie@5xruby.tw> 1503604211 +0800

add cat 3
```

應該沒有意外，它指向某個 Tree 物件，以及前一個 Commit 物件 (parent)。當然，那個 Tree 物件也有指著 Blob 物件。把這些資訊準備好後，就動手砍吧：

```
$ git reset HEAD^ --hard
HEAD is now at f6bba64 add cat 2
```

因為 hard Reset 指令，所以我們會預期原來的那個 Commit eb396d 應該會變成沒人愛的 Unreachable 狀態，來看看是不是這樣：

```
$ git fsck --unreachable
Checking object directories: 100% (256/256), done.
```

咦？不是應該會列出一些被清掉的東西在這裡嗎？原來，其實那個剛被 Reset 的 Commit 並不是真的沒人愛，至少現在還有 Reflog 愛它：

```
$ git reflog
f6bba64 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
eb396df HEAD@{1}: commit: add cat 3
f6bba64 (HEAD -> master) HEAD@{2}: commit: add cat 2
654b2fc HEAD@{3}: commit (initial): add cat 1
```

還記得 Reflog 嗎？如果忘記怎麼操作請參閱「[【狀況題】不小心使用 hard 模式 Reset 了某個 Commit，救得回來嗎？](#)」章節說明。

就是因為 Reflog 還對這個剛被砍掉的 Commit 還念念不忘，所以這顆 Commit 物件以及它底下的 Tree 物件以及 Blob 物件也暫時還活著。這時候，你可以默默的等 Reflog 過期（預設 30 天），然後剛這些被砍的傢伙就會真的變成沒人愛了。或是，直接手動讓 Reflog 過期：

```
$ git reflog expire --all --expire=now
```

這樣一來，Reflog 的紀錄就立馬過期了。再回來看看剛才刪掉的內容：

```
$ git fsck --unreachable
Checking object directories: 100% (256/256), done.
unreachable blob e124d9bf474d2caddade45fd7bd84a4f9b7f3bbe
unreachable commit eb396dfbdd7ab4ca6618fe643ada8cf60ca67472
unreachable tree 4c0c24949985ce2c87d99860b27791984d183d44
```

這三個 Unreachable 的傢伙，就是剛剛那一串 Commit 物件 → Tree 物件 → Blob 物件，因為已經沒人記得它們了，所以整串都變 Unreachable 狀態。不過即使它們變成 Unreachable 狀態，如果垃圾車沒來載走，它就永遠會在這裡。同樣，你可以慢慢等 Git 自己發動資源回收機制（物件個數要到一定量以上才會啟動），或是直接手動啟動：

```
$ git gc
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), done.
Total 6 (delta 0), reused 0 (delta 0)
```

看看載走了沒：

```
$ git fsck --unreachable
Checking object directories: 100% (256/256), done.
Checking objects: 100% (6/6), done.
unreachable blob e124d9bf474d2caddade45fd7bd84a4f9b7f3bbe
unreachable commit eb396dfbdd7ab4ca6618fe643ada8cf60ca67472
unreachable tree 4c0c24949985ce2c87d99860b27791984d183d44
```

怎麼還在啊！垃圾車不是都開來了嗎？原來是，還需要跟垃圾車說「現在給我載走」才行，不然它也是會等到過期才會載走：

```
$ git gc --prune=now
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), done.
Total 6 (delta 0), reused 6 (delta 0)
```

再確認一下：

```
$ git fsck --unreachable
Checking object directories: 100% (256/256), done.
Checking objects: 100% (6/6), done.
```

終於載走了，Git 的物件設計真的有點頑強啊！

事實上，`git gc` 指令會默默的呼叫 `git prune` 指令來清除 Unreachable 的物件，但 `git prune` 指令也是要給它設定到期日，所以剛才的指令：

```
$ git gc --prune=now
```

實際上等於這樣：

```
$ git gc  
$ git prune --expire=now
```

還可以怎麼做出這樣的邊緣物件？

知道資源回收怎麼運作之後，想想看，還有什麼情況下會產生這樣的邊緣物件？

情境一：在 Commit 前猶豫不決...

如果照正常的 Commit 流程，所有物件生成之後應該都會有人指向它，Tag 物件指向 Commit 物件，Commit 物件指向 Tree 物件，Tree 指向 Blob 物件（可參閱「[【超冷知識】在 .git 目錄裡有什麼東西？Part 1](#)」章節）。首先我先做一個檔案出來並加到暫存區：

```
$ echo "沒人愛" > no_body.html  
$ git add no_body.html
```

當把檔案加到暫存區之後，看看這時候的物件列表：

```
$ git ls-files -s  
100644 38d6c0201e33c5780192dc22b2e1017de8ab5563 0      cat1.html  
100644 56f551541c7c6b634525a141282e7cf9311d313d 0      cat2.html  
100644 e124d9bf474d2caddade45fd7bd84a4f9b7f3bbe 0      cat3.html  
100644 74b07973854e394cd8658d95a481709ebf00e3d0 0      no_body.html
```

Git 便會為它產生一個 Blob 物件（74b079）。這時候如果使用這個指令：

```
$ git rm --cached no_body.html  
rm 'no_body.html'
```

這時候使用 `git ls-files` 雖然看不到，但 74b079 這個物件並沒有消失，它還是活在 `.git/objects` 目錄裡。其實 Git 的物件一旦長出來之後，除非手動進 `.git/objects` 目錄處理掉，不然它就是會在那邊了。也就是說，這個 Blob 物件不像 Commit 物件可能還會有 Reflog 指著它，所以它立馬就會變 Unreachable 狀態：

```
$ git fsck --unreachable  
Checking object directories: 100% (256/256), done.  
unreachable blob 74b07973854e394cd8658d95a481709ebf00e3d0
```

確認一下內容：

```
$ git cat-file -p 74b07973854e394cd8658d95a481709ebf00e3d0
沒人愛
```

就是它沒錯！

其實如果在 Commit 之前，檔案 `git add` 之後再修改然後再執行 `git add` 指令，也會產生 Unreachable 物件。甚至像 `git commit --amend` 這樣單純只是改訊息的指令，也都會產生 Unreachable 的物件。

情境二：被刪除的 Tag 物件

這裡指的 Tag 物件是指有附註的標籤（Annotated Tag）。Tag 物件原本是指向某個 Commit，但當使用 `git tag -d TAG_NAME` 指令刪除之後，這個物件就變成 Unreachable 物件了。我先建立一個有附註的標籤：

```
$ git tag -a no_one_care -m "沒人愛"
```

然後動手把它砍掉：

```
$ git tag -d no_one_care
Deleted tag 'no_one_care' (was 5b4628e)
```

看一下狀況：

```
$ git fsck --unreachable
Checking object directories: 100% (256/256), done.
Checking objects: 100% (9/9), done.
unreachable tag 5b4628ea42b2f4f5892705b03f64eb8ca4180280
```

因為它不像 Commit 可能還會有 Reflog 對它念念不忘，已經沒有其它物件或指標指向這個 Tag 物件，所以它立馬就變 Unreachable 狀態了。

在做 Rebase 的時候

Rebase 的過程請參閱「[另一種合併方式（使用 rebase）](#)」章節介紹。在 Rebase 的過程，Git 是把 Commit 複製一份到新的 base 分支上並且重新計算，同時最後分支跟 HEAD 也都會移過去，所以原來的 Commit 就變成沒人愛了。

綜合以上種種，其實在 Git 裡面，沒人愛的 Unreachable 的物件是滿常見的，不過 Git 會幫你處理這些雜事，不需要你擔心。更多關於 Git 資源回收內容，可再參閱參閱「[【超冷知識】在.git 目錄裡有什麼東西？Part 2](#)」章節補充。

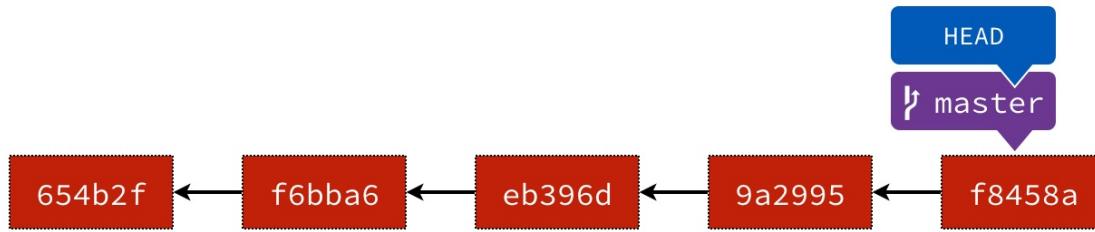
Dangling 跟 Unreachable 物件有什麼不同？

在執行 `git fsck` 的時候沒有帶參數的時候，偶爾會看到 "dangling" 的字樣。這跟前面提到的 Unreachable 物件有一點點不同：

1. Unreachable 物件：沒有任何物件或指標指著它，所以如同字面說的「無法到達」。雖然它「無法到達」，但它可以指著其它物件。
2. Dangling 物件：跟 Unreachable 物件一樣，沒有任何物件或指標指著它，就完全是懸在天邊的一顆物件。

Dangling 物件可以算是 Unreachable 物件的子集合，它也是一種 Unreachable 物件，所以在進行 GC 時候也會一起收走。

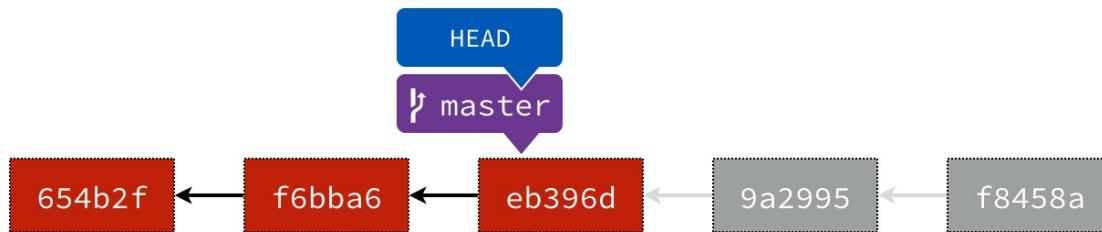
但什麼時候會產生這樣的情況？舉例來說，我在剛剛的例子又再多加了兩次 Commit，現在的樣子會像這樣：



這時，我一次 Reset 兩個 Commit：

```
$ git reset HEAD~2 --hard
```

也就是說，有兩顆 Commit 被拆下來了：



因為 `f8458a` 跟 `9a2995` 這兩個 Commit 都已經無法存取（其實還是可以啦），所以這兩個 Commit 都算是 Unreachable 狀態，連同底下整串的 Tree 物件跟 Blob 物件也都是：

```
$ git fsck --unreachable --no-reflogs
Checking object directories: 100% (256/256), done.
Checking objects: 100% (9/9), done.
unreachable tree 22e0e94b88e1d86c3cc8636add45b0bc78284722
```

```
unreachable blob 844e3541c6e899189cebe5210f8fe98f75b6e4f7
unreachable blob b0f0ad00157231befa3382df164044684756cf3b
unreachable commit f8458af074073915d89ec8c37e8fcf6636cba28b
unreachable commit 9a299576041a0cffdd38d7b29780788f23e095e9
unreachable tree b6c1eb2f863b8a1ce8bd653324d78a49e1c2fd9c
```

但這時候的 Dangling 物件只有這一個：

```
$ git fsck --no-reflogs
Checking object directories: 100% (256/256), done.
Checking objects: 100% (9/9), done.
dangling commit f8458af074073915d89ec8c37e8fcf6636cba28b
```

只有 `f8458a` 這個 Commit 物件是 Dangling 狀態，但另一個 Commit `9a2995` 不是，那是因為 `f8458a` 目前是完全沒有人指著它，而 `9a2995` 雖然也是 Unreachable，但至少還有 `f8458a` 指著它。

綜合以上介紹，檔案只要進了 Git 就不容易真的把它刪掉，詳情可參閱「[【冷知識】怎麼樣把檔案真正的從 Git 裡移掉？](#)」章節說明。

【冷知識】斷頭 (detached HEAD) 是怎麼一回事？

這個標題有點驚人，其實不是恐怖片的那種斷頭啦。在「[【冷知識】HEAD 是什麼東西？](#)」章節曾經介紹過，HEAD 是一個「指向某一個分支的指標」，你可以把 HEAD 當做「目前所在的分支」看待。

正常情況下，HEAD 會指向某一個分支，而分支會指向某一個 Commit。但 HEAD 偶爾會發生「沒有指到某個分支」的情況，這個狀態的 HEAD 便稱之「detached HEAD」。

可能發生這個狀態的原因有：

1. 使用 Checkout 指令直接跳到某個 Commit，而那個 Commit 剛好目前沒有分支指著它。
2. Rebase 的過程其實也是處於不斷的 detached HEAD 狀態。
3. 切換到某個遠端分支的時候。

在這個狀態可以做什麼？

其實這個狀態沒什麼特別的，就只是 HEAD 剛好指向某個沒有分支指著的 Commit 罷了，所以一樣可以跟平常一樣的操作 Git，一樣可以進行 Commit。這是原本的歷史紀錄：

All Branches	Hide Remote Branches	Ancestor Order	
Graph	Description	Commit	Author
	↳ master add dog 2	27f6ed6	Eddie Kao <ed...
	add dog 1	2bab3e7	Eddie Kao <ed...
	add 2 cats	ca40fc9	Eddie Kao <ed...
	add cat 2	1de2076	Eddie Kao <ed...
	add cat 1	cd82f29	Eddie Kao <ed...
	add database settings	382a2a5	Eddie Kao <ed...
	init commit	bb0c9c2	Eddie Kao <ed...

我使用 Checkout 指令切換至 `add cat 1` 那個 Commit：

```
$ git checkout cd82f29
Note: checking out 'cd82f29'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
```

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at cd82f29... add cat 1
```

或是直接在 SourceTree 的 Commit 上點兩下，也可以有一樣的效果。現在的狀態是這樣：

All Branches	Hide Remote Branches	Ancestor Order	
Graph	Description	Commit	Author
●	master add dog 2	27f6ed6	Eddie Kao <ed...
●	add dog 1	2bab3e7	Eddie Kao <ed...
●	add 2 cats	ca40fc9	Eddie Kao <ed...
●	add cat 2	1de2076	Eddie Kao <ed...
●	HEAD add cat 1	cd82f29	Eddie Kao <ed...
●	add database settings	382a2a5	Eddie Kao <ed...
●	init commit	bb0c9c2	Eddie Kao <ed...

這時候我試著進行一次 Commit：

```
$ touch no-head.html  
$ git add no-head.html  
$ git commit -m "add a no-head file"  
[detached HEAD b6d204e] add no-head file  
 1 file changed, 0 insertions(+), 0 deletions(-)  
 create mode 100644 no-head.html
```

這時候的狀態變這樣：

All Branches	Hide Remote Branches	Ancestor Order	
Graph	Description	Commit	Author
	HEAD add no-head file	b6d204e	Eddie Kao <e...
	master add dog 2	27f6ed6	Eddie Kao <e...
	add dog 1	2bab3e7	Eddie Kao <e...
	add 2 cats	ca40fc9	Eddie Kao <e...
	add cat 2	1de2076	Eddie Kao <e...
	add cat 1	cd82f29	Eddie Kao <e...
	add database settings	382a2a5	Eddie Kao <e...
	init commit	bb0c9c2	Eddie Kao <e...

有覺得有點眼熟嗎？是的，這就跟「[【狀況題】我可以從過去的某個 Commit 再長一個新的分支出來嗎？](#)」章節介紹的差不多，只是，現在這個 Commit (b6d204e) 還「沒有名字」，更正確的說是還沒有分支指向它，目前僅有 HEAD 指著它而已。

這有什麼影響嗎？影響就是當我的 HEAD 回到其它分支之後，這個 Commit 就不容易被找到（除非你有記下這個 Commit 的 SHA-1 值），如果一直沒人來找它，過久了之後就會被 Git 啟動的資源回收機制給收掉了。

所以，如果還想留下這個 Commit，就給它一個分支指著它就行了。如果你剛好就在這個 Commit 上的話：

```
$ git branch tiger
```

或是明確的跟 Git 說幫你建立一個分支指向某個 Commit：

```
$ git branch tiger b6d204e
```

雖然剛建完分支，當下的狀態還是處於 detached HEAD，不過不用太擔心，這個 Commit 以後就可以透過 tiger 這個分支來找到它了。你也可以使用 Checkout 指令配合 -b 參數，建立分支後直接切換：

```
$ git checkout -b tiger b6d204e
Switched to a new branch 'tiger'
```

為什麼切換到遠端的分支也會是這個狀態？

前面提到當 HEAD 沒有指到某個分支的時候，它會呈現 detached 狀態。事實上，更正確的說，應該是說「當 HEAD 沒有指到某個『本地』的分支」就會呈現這個狀態。舉例來說，有一個我剛從自己的 GitHub 帳號 Clone 下來的專案，我使用 `git branch` 指令檢視目前的分支：

```
$ git branch --remote
origin/HEAD -> origin/master
origin/master
origin/refactoring
```

使用 `--remote` 或 `-r` 參數可以顯示遠端的分支，當我試著切換到 `origin/refactoring` 這個分支的時候：

```
$ git checkout origin/refactoring
Note: checking out 'origin/refactoring'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 1ec82d7... refactored
```

OOPS！變成 detached HEAD 狀態了。要切換到遠端分支而不呈現 detached HEAD 狀態，可以加上 `--track` 或 `-t` 參數：

```
$ git checkout -t origin/refactoring
Branch refactoring set up to track remote branch refactoring from origin.
Switched to a new branch 'refactoring'
```

這樣就切過去了，那個 `-t` 參數是指會在本機建立一個名為追蹤分支 (tracking branch) 的東西。或是簡單一點直接把前面的 `origin` 拿掉：

```
$ git checkout refactoring
Branch refactoring set up to track remote branch refactoring from origin.
Switched to a new branch 'refactoring'
```

也會有一樣的效果。關於遠端分支的內容，會在「遠端共同協作」的相關章節再說明。

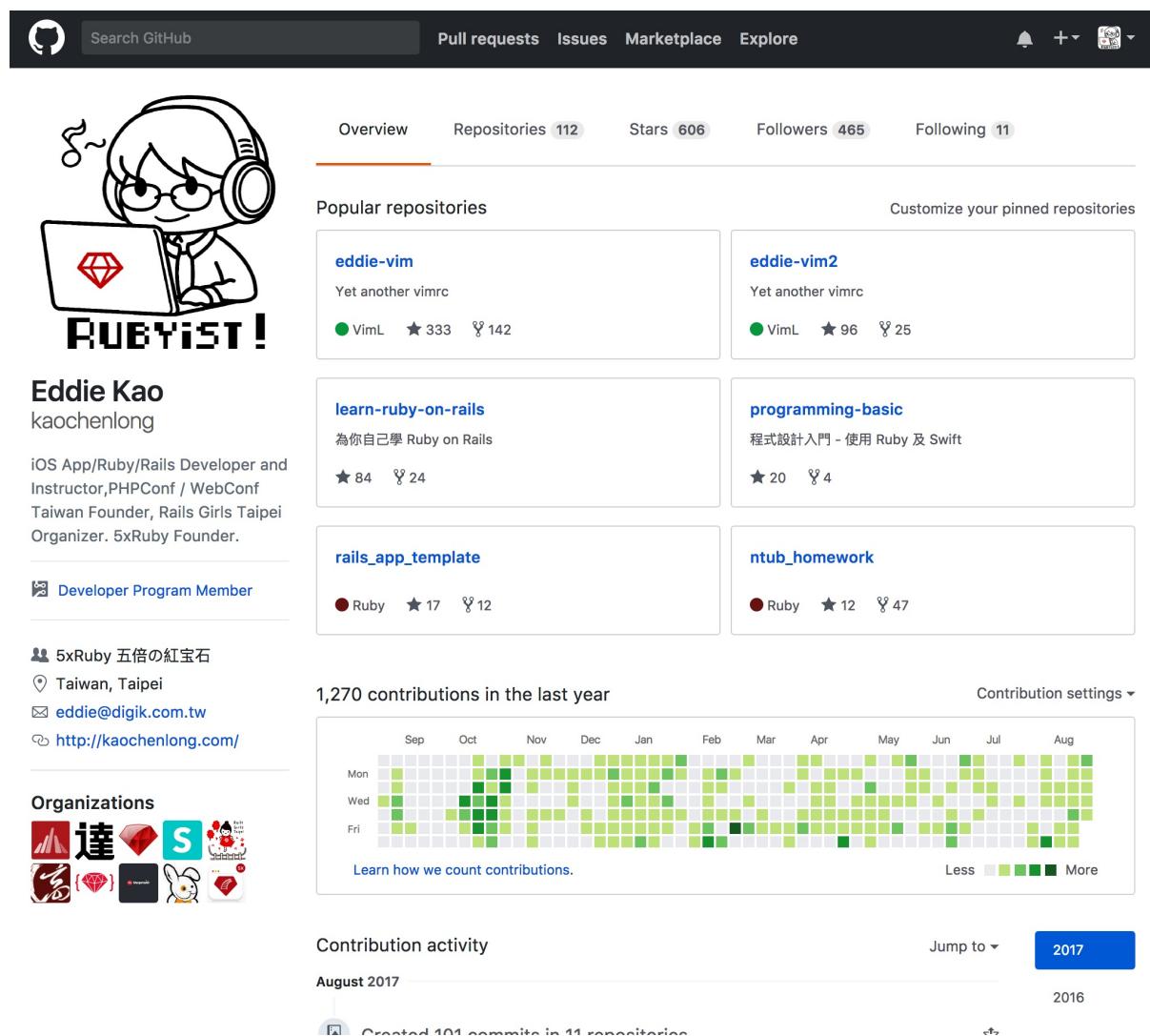
怎麼離開 detached HEAD 狀態？

既然已經知道所謂的 detached HEAD 狀態只是 HEAD 沒有指向任何分支造成的，要脫離這個狀態，只要讓 HEAD 有任何分支可以指就行了，例如讓它回到 master 分支：

```
$ git checkout master  
Switched to branch 'master'
```

這樣一來 HEAD 就解除 detached 狀態了。

GitHub 是什麼？



The screenshot shows Eddie Kao's GitHub profile. At the top, there's a search bar, navigation links for Pull requests, Issues, Marketplace, and Explore, and a notifications icon. Below the header, there's a cartoon illustration of Eddie wearing headphones and holding a laptop with a diamond icon. The profile summary includes 'Overview', 'Repositories 112', 'Stars 606', 'Followers 465', and 'Following 11'. A section titled 'Popular repositories' lists several projects like 'eddie-vim', 'learn-ruby-on-rails', and 'rails_app_template'. Another section titled 'Customize your pinned repositories' shows 'eddie-vim2' and 'programming-basic'. On the left, there's a sidebar with information about Eddie being a Developer Program Member, his location in Taiwan, Taipei, and his email (eddie@digik.com.tw). It also lists his organizations, including '達人' and '5xRuby'. The main area features a heatmap showing contributions over the last year, with a legend indicating contribution levels from 'Less' to 'More'. Below the heatmap, there's a 'Contribution activity' section for August 2017, showing he created 101 commits in 11 repositories.

在開始介紹 GitHub 之前，請記得它的 `G` 跟 `H` 是大寫，其它字母小寫。它是一個商業網站，是目前全球最大的 Git Server。在這邊，你可以跟其它厲害的開發者們交朋友，你可以幫忙貢獻其它人的專案，其它人也可以回饋到你的專案，建立良性循環。

同時，它也是開發者最好的履歷。你曾經做過哪些專案、做過哪些貢獻、寫過哪些 Code，一目了然，想要假造非常花工夫。

如果你是上傳 Open Source 專案的話，可以完全免費使用。如果想要在上面開私人專案的話則需要收費，費用是每個月 7 塊錢美金。

補充：Microsoft 在 2018 年 10 月 [併購了 GitHub](#)，以往需要付費才能開設私人專案，在 2019 年 1 月 [宣佈](#)即使是免費帳號也可無限制的開設私人專案。

學生優惠方案

如果你還有學生身份(好吧，事實上只要還有 .edu 的信箱就行了) ，GitHub 提供了許多優惠方案給還在學的學生，其中一項，就是只要你是學生的期間，都可以享有每個月 7 元美金的方案。

The screenshot shows the GitHub Education website with a yellow header. The main title is "Student Developer Pack" with the subtitle "The best developer tools, free for students". Below the title is a yellow backpack icon with the GitHub logo on it. To the right of the icon is a button labeled "Get your pack". Above the backpack icon, there is a call-to-action text: "Learn to ship software like a pro". To the right of this text are social sharing buttons for "Like 42K" and "Tweet". Below the backpack icon, there is a section titled "THE TOOLS" which is currently empty.

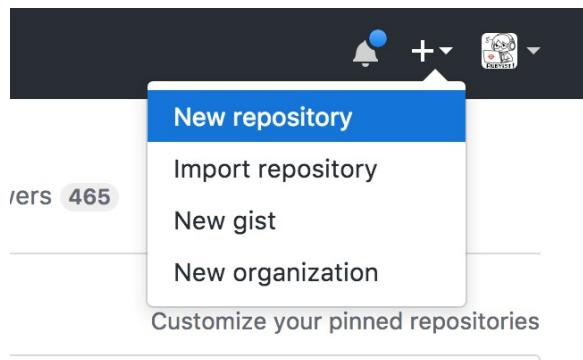
網址：<https://education.github.com/pack>

Git 跟 GitHub 的差別

一個是工具，一個是網站，GitHub 的本體是一個 Git Server，網站則是使用 Ruby on Rails 開發，別把這兩個名詞搞混囉。

在 GitHub 上開新專案

要上傳檔案到 GitHub，需要先在上面開一個新的專案。請先在 GitHub 網站的右上角點選「+」號，並選擇「New repository」：



接著填寫專案名稱：

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner	Repository name <div style="border: 1px solid #ccc; padding: 2px; display: flex; align-items: center;"> kaochenlong / practice-git ✓ </div> <p>Great repository names are short and memorable. Need inspiration? How about supreme-broccoli.</p> <p>Description (optional)</p> <div style="border: 1px solid #ccc; height: 40px; margin-top: 5px;"></div>
<p><input checked="" type="radio"/> Public Anyone can see this repository. You choose who can commit.</p> <p><input type="radio"/> Private You choose who can see and commit to this repository.</p> <p><input type="checkbox"/> Initialize this repository with a README This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.</p>	
<div style="display: flex; justify-content: space-around;"> Add .gitignore: None Add a license: None (i) </div> <input style="background-color: #00aaff; color: white; font-weight: bold; padding: 5px 20px; border: none; border-radius: 5px; width: fit-content;" type="button" value="Create repository"/>	

1. Repository name 可填寫任意名稱，只要不重複即可。

2. 存取權限選擇 Public 可免費使用，選擇 Private 則需收每個月 7 元美金的月費。

補充：在 GitHub 被 Microsoft 併購後，現在即使是免費帳號也都可以開立 Private 專案喔。

按下「Create repository」即可新增一新的 Repository。

接下來，會看到的這個引導畫面：

The screenshot shows a GitHub repository creation interface. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the bar, the repository name 'kaochenlong / practice-git' is displayed, along with 'Unwatch' (1), 'Star' (0), and 'Fork' (0) buttons. A menu bar below the repository name includes 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Settings', and 'Insights'. The main content area is titled 'Quick setup — if you've done this kind of thing before'. It provides instructions for setting up the repository on desktop or via SSH, with the SSH URL 'git@github.com:kaochenlong/practice-git.git'. It also recommends including a README, LICENSE, and .gitignore file. Below this, there's a section for '...or create a new repository on the command line' with a code block containing the following commands:

```
echo "# practice-git" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:kaochenlong/practice-git.git
git push -u origin master
```

Finally, there's a section for '...or push an existing repository from the command line' with a code block containing:

```
git remote add origin git@github.com:kaochenlong/practice-git.git
git push -u origin master
```

說明：

- 如果是全新開始，請依「create a new repository on the command line」的內容指示進行；如果是要上傳現存專案，則依照「push an existing repository from the command line」選項進行。
- 畫面中間上面，有兩個按鈕可以切換，分別是「HTTPS」以及「SSH」。至於要選擇哪一個可看個人喜好，但選擇 SSH 的話需要設定 SSH Key，SSH Key 的設定可參閱「GitHub 是什麼？」章節介紹。因為我已經有設定好 SSH Key，所以這裡我選擇「SSH」。

如果仔細觀察，便會發現選擇全新開始跟上傳現有專案這兩個選項的最後兩個步驟是一樣的。

假設我們現在什麼都沒有，要重新開始一個專案，找一個空的目錄，就照著它的說明進行。首先，先建立一個 README.md 檔案：

```
$ echo "# Practicing Git" > README.md
```

這個 README.md 是 GitHub 專案的預設說明頁面，附檔名 .md 表示它是一個 Markdown 格式，Markdown 語法可以很輕鬆的把純文字格式轉換成 HTML 的網頁格式，如果這是你第一次接觸這個語法，非常推薦花一點時間學習它。

接下來這段，就是我們熟悉的 Git 了：

```
$ git init
Initialized empty Git repository in /private/tmp/practice-git/.git/

$ git add README.md

$ git commit -m "first commit"
[master (root-commit) adc1a5a] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

到這裡，都還是 Git 的基本招式。如果忘記怎麼操作可參閱「[把檔案交給 Git 控管](#)」章節。

在這之前，我們所有的操作都是在自己電腦上，接下來就是要準備把東西推上遠端的 Git 同伺服器了。首先，需要設定一個端節的節點，例如：

```
$ git remote add origin git@github.com:kaochenlong/practice-git.git
```

說明：

1. `git remote` 指令，顧名思義，主要是跟遠端有關的操作。
2. `add` 指令是指要加入一個遠端的節點。
3. 在這裡的 `origin` 是一個「代名詞」，指的是後面那串 GitHub 同伺服器的位置。

在慣例上，遠端的節點預設會使用 `origin` 這個名字。如果是從 Server 上 clone 下來的話，它的預設的遠端節點就會叫 `origin`。（關於 Clone 的使用，請參閱「[從同伺服器上取得 Repository](#)」章節說明）

不過別擔心，這只是個慣例，不用這名字或是之後想要改也都可以，如果想改叫七龍珠

`dragonball`：

```
$ git remote add dragonball git@github.com:kaochenlong/practice-git.git
```

總之，它就是只是指向某個位置的代名詞罷了。設定好遠端節點後，接下來，就是要把東西推上去了：

```
$ git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 228 bytes | 228.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:kaochenlong/practice-git.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

這個簡單的 Push 指令其實做了幾件事：

1. 把 `master` 這個分支的內容，推向 `origin` 這個位置。
2. 在 `origin` 那個遠端 Server 上，如果 `master` 不存在，就建立一個叫做 `master` 的同名分支。
3. 但如果本來 Server 上就存在 `master` 分支，便會移動 Server 上 `master` 分支的位置，使它指到目前最新的進度上。
4. 設定 `upstream`，就是那個 `-u` 參數做的好事，這個稍候說明。

如果你能理解上面這個指令的意思，你就可以再做一些變化。例如你的遠端節點叫做 `dragonball`，而且你想把 `cat` 分支推上去：

```
$ git push dragonball cat
```

這樣就會把 `cat` 分支推上 `dragonball` 這個遠端節點所代表的位置，並且在上面建立一個名為 `cat` 同名分支（或更新進度）。

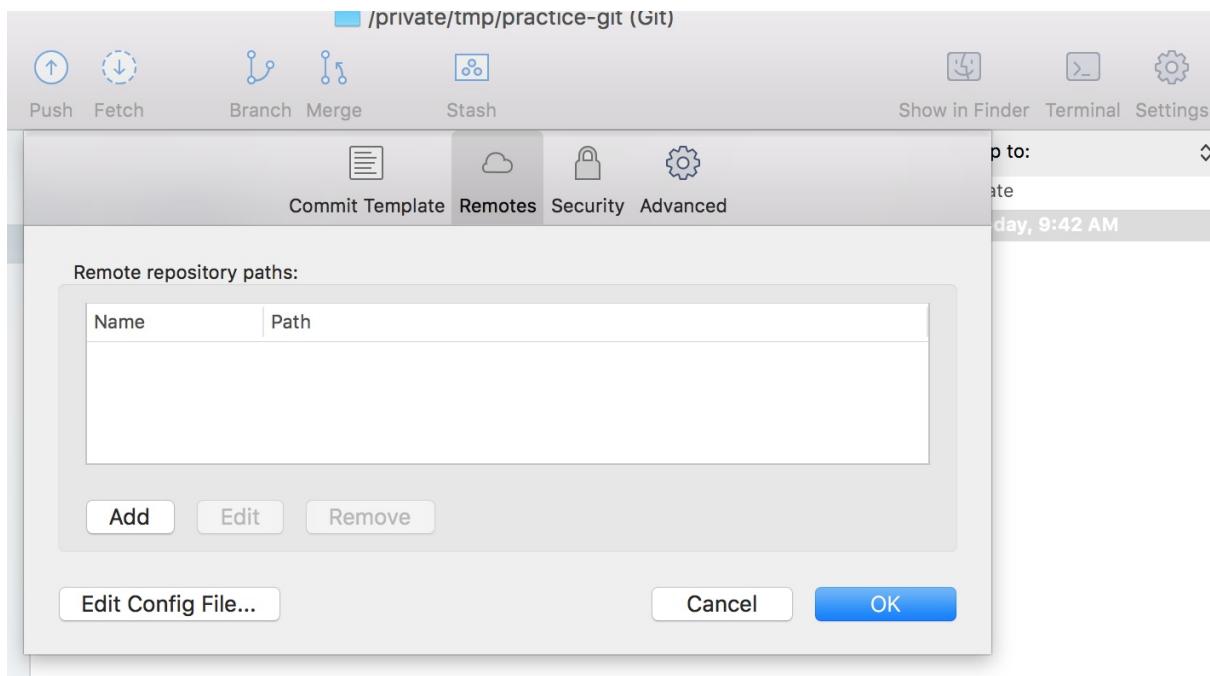
回到 GitHub 網站，重新整理一下頁面，剛才那個引導指令的畫面，現在應該會變這樣：

The screenshot shows a GitHub repository page for 'kaochenlong / practice-git'. The top navigation bar includes 'This repository', 'Search', 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the right, there are buttons for 'Unwatch' (1), 'Star' (0), 'Fork' (0), and a profile icon. Below the navigation, there are tabs for 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Settings', and 'Insights'. A note says 'No description, website, or topics provided.' with an 'Edit' button. Under the repository summary, it shows '1 commit', '1 branch', '0 releases', and '1 contributor'. A dropdown menu shows 'Branch: master' and a 'New pull request' button. Below this, a commit card for 'kaochenlong first commit' is shown, with a file list including 'README.md' and a timestamp 'Latest commit adc1a5a 2 minutes ago'. The main content area contains the file 'README.md' with the text 'Practicing Git'.

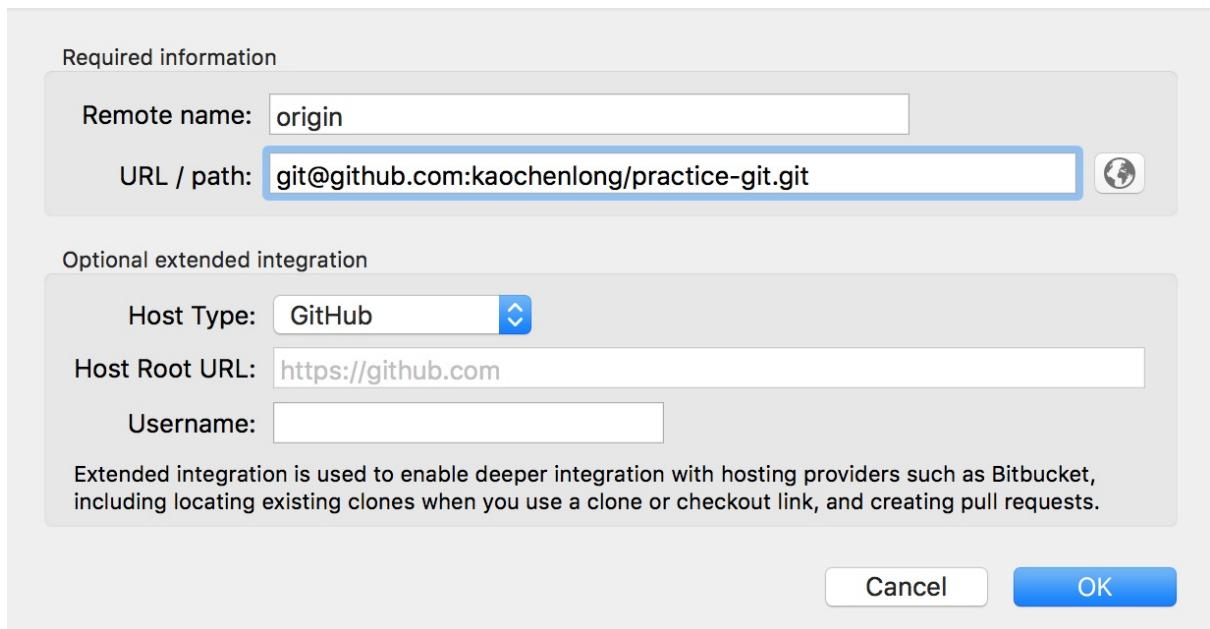
恭喜你，看到這畫面表示你已經順利把你本地專案的東西推到這個遠端的專案裡了。

Git 教學：使用 SourceTree

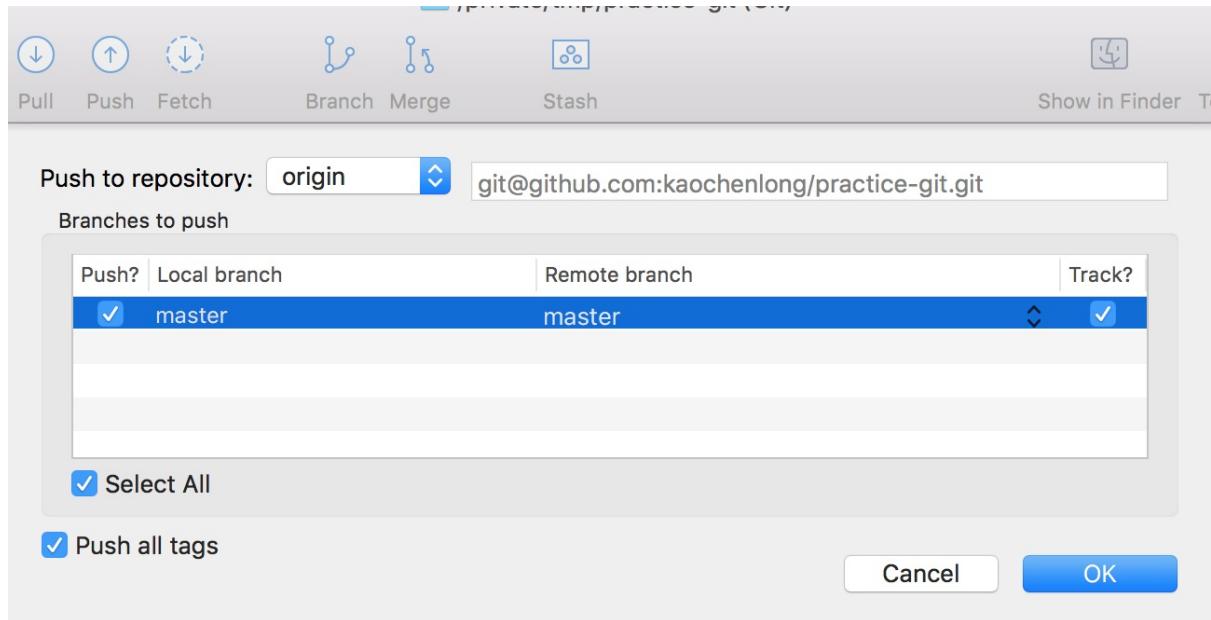
如果是使用 SourceTree，請按下右上角的「Settings」按鈕，跳出對話框後，選擇「Remote」頁籤：



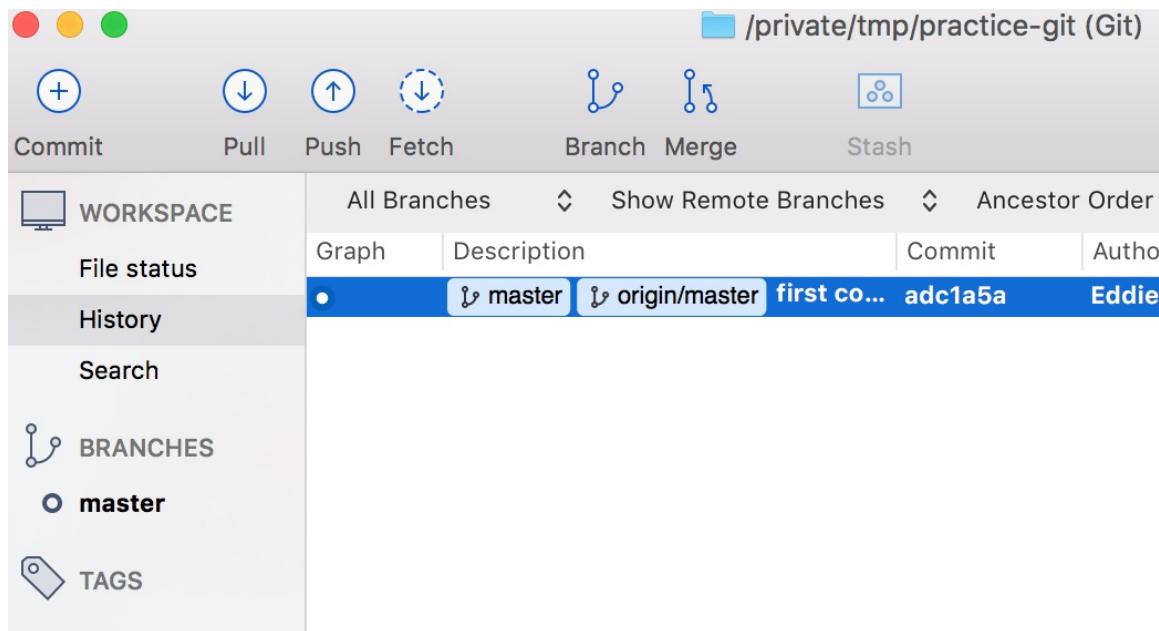
點選「Add」按鈕：



填寫「Remote name」以及「URL」欄位後，按下 OK 按鈕後，遠端節點即設定完成。接下來要把東西往上推，請點選上面工具列的「Push」按鈕：



勾選你想推的分支，按下 OK 鈕之後就會開始把指定的分支往指定的遠端節點推。成功之後再回來看，現在在 `master` 分支旁邊就多了一個 `origin/master` 的分支了：



【冷知識】設定 upstream 是什麼意思

在前面進行 Push 的時候，有加入了一個 `-u` 參數，表示要設定 upstream 的意思。嗯...問題是，什麼是「upstream」？

`upstream`，中文翻譯成「上游」。不要被名詞嚇到了，這 `upstream` 的概念其實就只是另一個分支的名字而已。在 Git 裡，每個分支可以設定一個「上游」（但每個分支最多只能設定一個 `upstream`），它會指向並追蹤（track）某個分支。通常 `upstream` 會是遠端 Server 上的某個分支，但其實要設定在本地端的其它分支也可以。

如果有設定，當下次執行 `git push` 指令的時候，它就會用來當預設值。舉例來說：

```
$ git push -u origin master
```

就會把 `origin/master` 設定為本地 `master` 分支的 `upstream`，當下回執行 `git push` 指令而不加任何參數的時候，它就會猜你是要推往 `origin` 這個遠端節點，並且把 `master` 這個分支推上去。

反之，沒設定 `upstream` 的話，就必須在每次 Push 的時候都跟 Git 講清楚、說明白：

```
$ git push origin master
```

否則，光就執行 `git push` 指令而不帶其它參數的話，Git 會跟你埋怨說不知道該 Push 什麼分支以及要 Push 去哪裡：

```
$ git push
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master
```

【冷知識】如果不想要同名的分支名字該如何更改？

前面提到 Push 的指令是這樣：

```
$ git push origin master
```

其實上面這個指令跟下面這個指令是一樣的效果：

```
$ git push origin master:master
```

意思是「把本地的 `master` 分支推上去後，在 Server 上更新 `master` 分支的進度，或是如果不存在該分支的話，就建立一個 `master` 分支」。但如果你想推上去之後不要叫這個名字的話，可以把後面的那個名字改掉：

```
$ git push origin master:cat
```

這樣當把本地端的 `master` 分支推上去之後，就不會在線上建立 `master` 分支，而是建立（或更新進度）一個叫做 `cat` 的分支了。

Pull 下載更新

上個章節我們介紹了如何把東西[推上 GitHub](#)，接下來我們看看怎麼把東西拉回來更新。

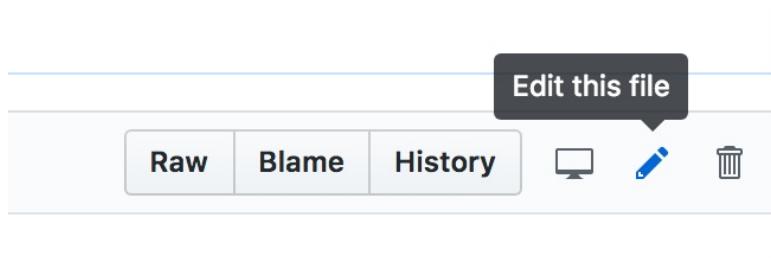
跟 Push 指令相反，Pull 指令是拉回本機更新。但要介紹 Pull 之前，需要先介紹一下 Fetch 這個指令。

Fetch 指令才是把東西拉回來的主角

以上個章節的例子來說（網址：<https://github.com/kaochenlong/practice-git>），我們試著執行這個指令：

```
$ git fetch
```

你會發現沒有任何訊息，那是因為現在我們的進度跟線上版本是一樣的（廢話，因為就只有我自己一個人啊）。為了營造「有不同進度」的效果，我們可以到 GitHub 網站上，直接在線上編輯某個檔案。例如我點進 `README.md` 檔案，在右上角會有個編輯的按鈕：



編輯內容如下：

The screenshot shows a GitHub repository named 'kaochenlong / practice-git'. The 'Code' tab is selected. Below it, a modal window is open for the file 'README.md'. The modal has tabs for 'Edit file' and 'Preview changes'. The 'Preview changes' tab is active, showing the following content:

```

1 # Practicing Git
2
3 這是我的 Git 練習本|

```

按下下方的「Commit changes」即可進行存檔並新增一次 Commit，這樣一來線上版本的 Commit 數就領先本機一次了。此時再次執行 Fetch 指令：

```
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:kaochenlong/practice-git
  85e848b..8c3a0a5  master      -> origin/master
```

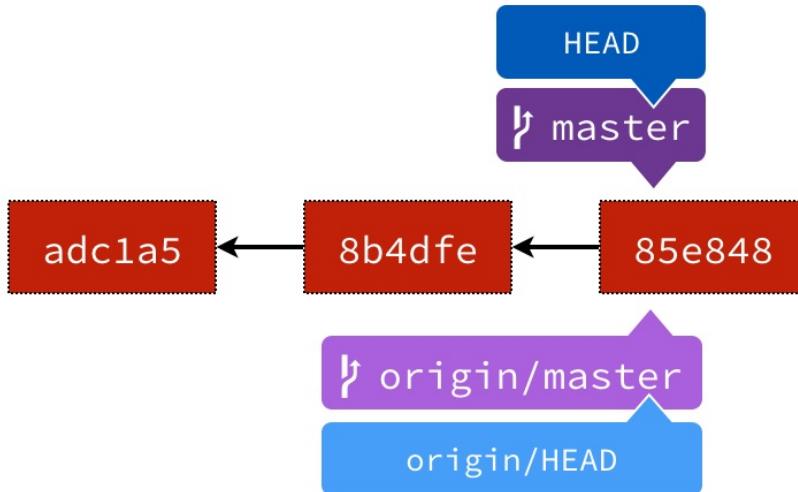
就可以看得出來有拉東西回來了。這時先看一下狀態：

All Branches	Show Remote Branches	Ancestor Order	Commit	
Graph	Description			
	origin/master	origin/HEAD	Update README.md	8c3a0a5
	master	update cat1		85e848b
		add cat 1		8b4dfec
		first commit		adc1a5a

會發現在 `master` 分支前面，有兩個奇怪的分支分別是 `origin/master` 跟 `origin/HEAD` .. Fetch 的過程到底發生了什麼事呢？

Fetch 過程發生了什麼事？

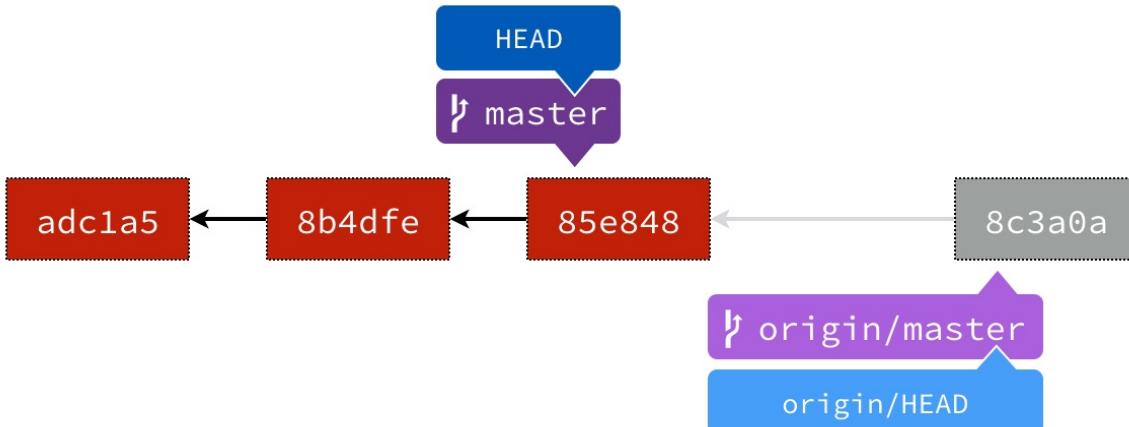
讓我用畫面來說明一下，這是在 Fetch 前的樣子，HEAD 跟 master 分支都不意外的乖乖待在它們該在的位置：



因為現在這個專案之前有推送東西到 Server 上，所以遠端分支也會記錄一份在本機上，一樣也是有 HEAD 跟 master 分支，但會在前面加註遠端節點 origin，變成 origin/HEAD 跟 origin/master。

如果你還記得，我們在第一次推的時候有使用了 `-u` 參數設定 upstream，所以目前這個 origin/master 分支其實就是本地 master 分支的 upstream 呀。

接下來，當執行 Fetch 指令，Git 看了一下線上版本的內容後，把你目前線上有但你這邊沒有的內容抓了一份下來，同時移動 origin 相關的分支：



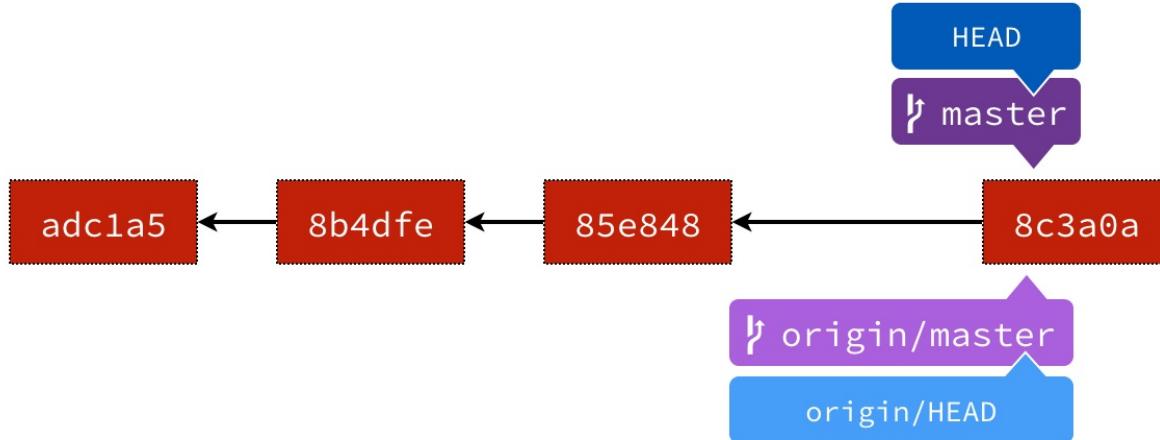
先不管 origin/master 這分支名字有點奇怪，也不用管它是本地還是遠端的分支，對 Git 來說，它就是一個從原本 master 分支分出去的分支而已。

既然這個分支是從 master 分支分出去而且進度還比 master 分支還要新，如果 master 分支想要跟上它，這個情境對大家來說不知道會不會有點熟悉？是的，就是合併 (merge) 啦：

```
$ git merge origin/master
```

```
Updating 85e848b..8c3a0a5
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

因為 `origin/master` 分支跟 `master` 分支本是同根生，所以在上面合併的過程可以看到是使用快轉模式（Fast Forward）方式進行。關於合併的說明，可參考「[合併分支](#)」章節介紹。現在的狀態就會變這樣：



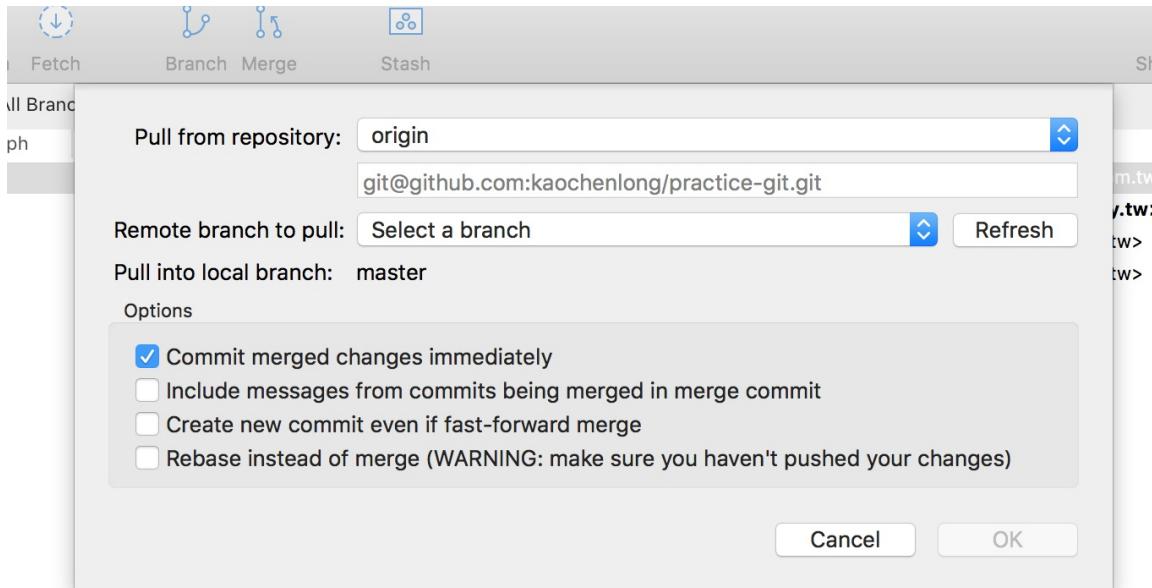
Pull 指令

如果你能理解上面 Fetch 指令在做什麼，那 Pull 指令對你來說就沒什麼問題了，因為：

```
git pull = git fetch + git merge
```

就這樣而已，Pull 指令其實就是去上線抓東西下來（Fetch），並且更新本機的進度（Merge）而已。

如果使用 SourceTree，在上方的工具箱裡就直接有一顆 Pull 跟 Fetch 按鈕，按下 Pull 按鈕後：



在「Remote branch to pull」下拉選單可選擇想要拉的遠端分支。

另外，在下方有一些選項，其中第一個「Commit merged changes immediately」這個選項，如果你知道 Pull 指令其實還有外帶 Merge 效果，現在你應該知道這個選項代表的意思了。

而第三個選項「Create new commit even if fast-forward merge」意思就是「請不要幫我使用快轉模式（Fast Forward）方式合併」，也就是使用 `--no-ff` 參數的意思。

最後一個選項「Rebase instead of merge」嗎？這個可看個人喜好而決定要不要勾，稍後會再說明。

按下 OK 鈕便可完成 Pull 指令。

Pull + Rebase

我們現在知道 Pull 實際等於 Fetch 加上 Merge，而在「[另一種合併方式（使用 rebase）](#)」章節也曾介紹過使用 Rebase 方式來合併，在執行 `git pull` 指令的時候，也可以再加上 `--rebase` 參數，它在 Fetch 完成之後，便會使用 Rebase 方式進行合併：

```
$ git pull --rebase
```

這有什麼好處？在多人共同開發的時候，大家都各自在自己的分支進行 Commit，所以拉回來用一般的方式合併的時候常會產生為了合併而產生的額外 Commit（詳情請參閱「[合併分支](#)」章節）。為了合併而產生的這個 Commit 本身並沒有什麼問題，但如果你不想要這個額外的 Commit，可考慮使用 Rebase 方式來進行合併。

而在 SourceTree 的 Pull 對話框中，勾選「Rebase instead of merge」選項就會有一樣的效果。

【狀況題】怎麼有時候推不上去...

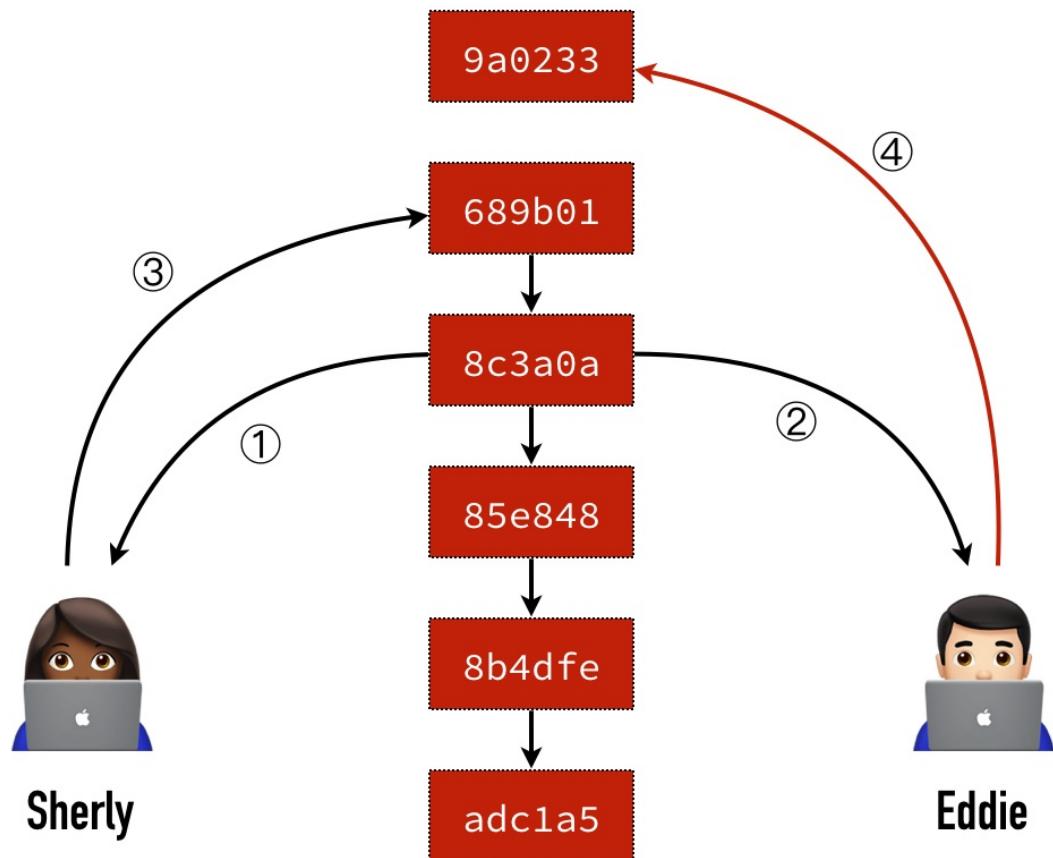
偶爾在執行 Push 指令的時候會出現這個錯誤訊息：

```
$ git push
To https://github.com/eddiekao/dummy-git.git
 ! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'https://github.com/eddiekao/dummy-git.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

這段訊息的意思是線上版本的內容比你電腦裡這份還要新，所以 Git 不讓你推上去。

怎麼造成的？

通常這個狀況會發生在多人一起開發的時候，想像一下這個情境：



1. Sherly 跟 Eddie 兩個人在差不多的時間都從 Git Server 上拉了一份資料下來準備進行開發。
2. Sherly 手腳比較快，先完成了，於是先把做好的成果推一份上去。
3. Eddie 不久後也完成了，但當他要推上去的時候發現推不上去了…

怎麼解決？

解決方法算是有兩招

第一招：先拉再推

因為你電腦裡的內容是比較舊的，所以你應該先拉一份線上版本的回來更新，然後再推一次：

```
$ git pull --rebase
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/eddiekao/dummy-git
 37aaef6..bab4d89  master      -> origin/master
First, rewinding head to replay your work on top of it...
Applying: update index
```

這裡加了 `--rebase` 參數是表示「內容抓下來之後請使用 Rebase 方式合併」，當然你想用一般的合併方式也沒問題。合併如果沒發生衝突，接著應該就可以順利往上推了。

第二招：無視規則，總之就是聽我的！（誤）

凡事總有先來後到，在上面的例子中，Sherly 先推上去的內容，後推的人就是應該拉一份下來更新，不然照規定是推不上去的。不過這規則也是有例外，只要加上了 `--force` 或是 `-f` 參數，它就會強迫硬推上去，把 Sherly 之前的内容蓋掉：

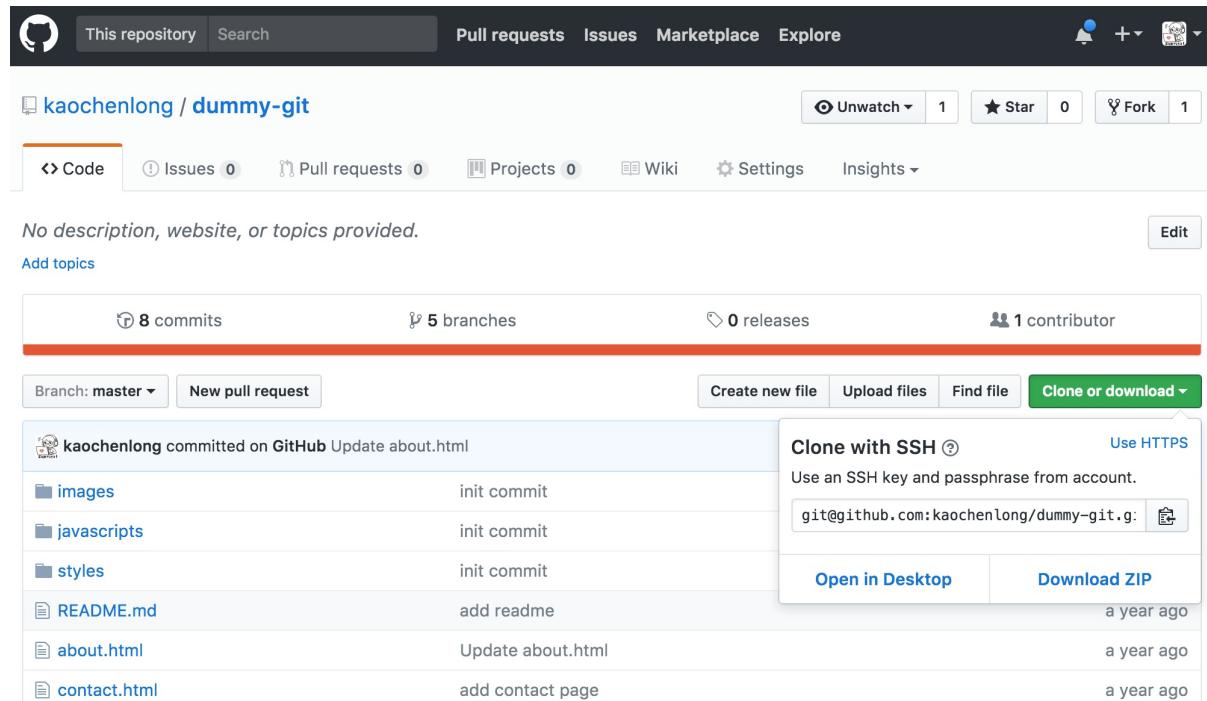
```
$ git push -f
Counting objects: 19, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (17/17), done.
Writing objects: 100% (19/19), 2.16 KiB | 738.00 KiB/s, done.
Total 19 (delta 6), reused 0 (delta 0)
remote: Resolving deltas: 100% (6/6), done.
To https://github.com/eddiekao/dummy-git.git
 + 6bf3967...c4ea775 master -> master (forced update)
```

雖然這樣一定會成功，但接下來你就要去面對 Sherly，跟她解釋為什麼你把她的進度蓋掉了。更多關於 Force Push 的說明，可參考「[【狀況題】聽說 git push -f 這個指令很可怕，什麼情況可以用它呢？](#)」章節介紹。

從伺服器上取得 Repository

在前面介紹了怎麼推（Push）以及怎麼拉（Pull），但這個的前提都是我們已經有這個專案了。

如果你在 GitHub 上看到某個專案很有趣，想要下載回來看看，只要使用 Clone 指令就可以把整個專案複製一份回來了。在 GitHub 專案的頁面有一個「Clone or download」的按鈕：



同樣可以選擇 HTTPS 或是 SSH，這裡我選擇 SSH。複製連結之後，接著便可使用 Clone 指令把它複製下來：

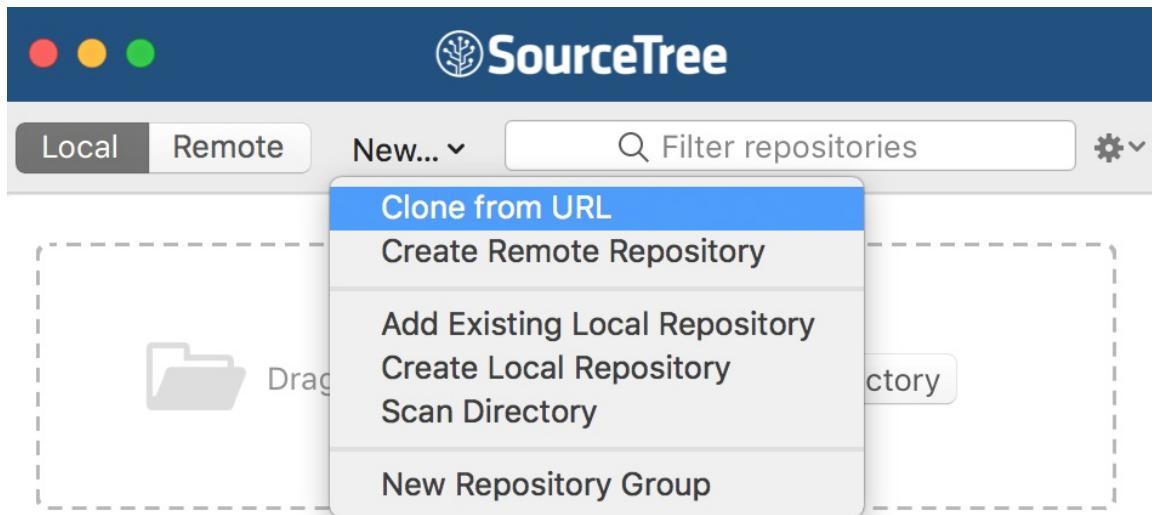
```
$ git clone git@github.com:kaochenlong/dummy-git.git
Cloning into 'dummy-git'...
remote: Counting objects: 47, done.
remote: Total 47 (delta 0), reused 0 (delta 0), pack-reused 47
Receiving objects: 100% (47/47), 28.78 KiB | 5.76 MiB/s, done.
Resolving deltas: 100% (16/16), done.
```

這個指令會把整個專案複製一份下來並存在同名的目錄裡。如果想要 Clone 下來之後存成不同的目錄名稱的話，只要在後面加上目錄的名字即可：

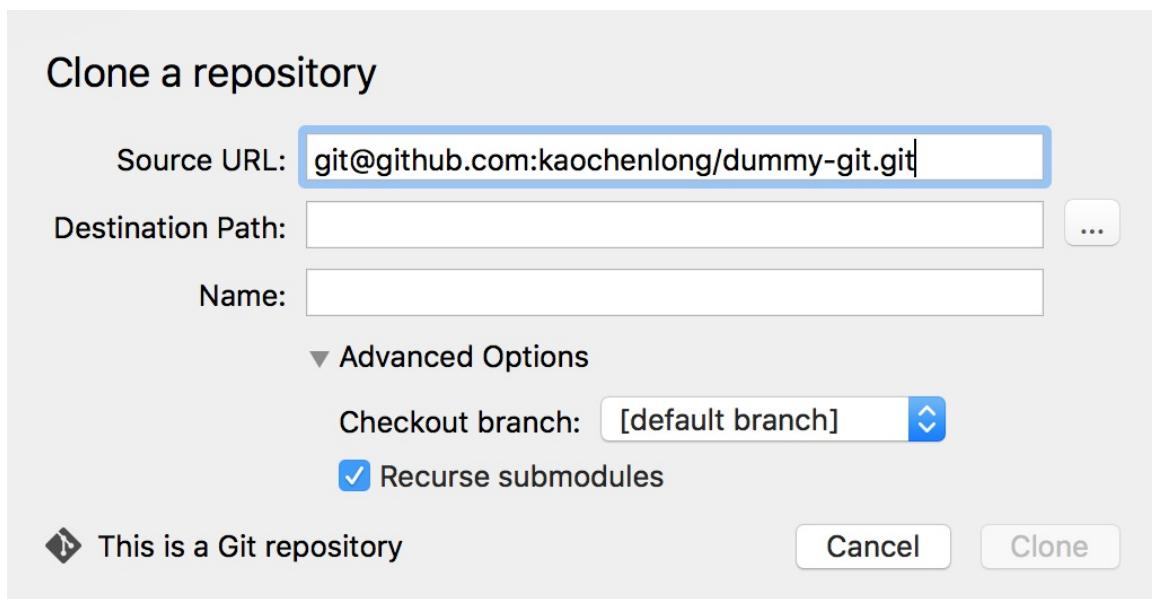
```
$ git clone git@github.com:kaochenlong/dummy-git.git hello_kitty
```

Clone 指令會把整個專案的內容複製一份到你的電腦裡，這裡指的「內容」不是只有檔案，而是指所有整個專案的歷史紀錄、分支、標籤等內容都會複製一份下來。

如果使用 SourceTree，請回到最一開始的起始畫面，或是選擇上面的功能選單，選擇「File」→「New...」：



選擇「Clone from URL」：



填寫 URL 以及要存放的目的地資料後，按下 Clone 按鈕即可完成 Clone 指令。

【常見問題】Clone 跟 Pull 指令有什麼不一樣？

對已經使用 Git 一陣子的老手來說，可能會覺得這個問題有點蠢，但對剛接觸 Git 的新手來說，Clone 跟 Pull 這兩個指令就以字面上來說感覺都有「把檔案下載到我的電腦」的感覺，不知道什麼時候該用 Clone、什麼時候該用 Pull...

Clone 跟 Pull 這兩個指令的應用場景是不同的。如果這個專案你是第一次看到，想要下載到你的電腦裡，請使用 Clone 指令；如果你已經下載回來了，你只是想要更新最新的線上版內容，請使用 Pull（或 Fetch）指令。

簡單的說，Clone 指令通常只會使用第一次，Clone 之後的更新，就是 Pull/Fetch 的事了。

與其它開發者的互動 - 使用 Pull Request (PR)

在 GitHub 上，有非常多的開源專案，有些專案你很有興趣，也很想幫忙、貢獻一己之力，於是
你可能寫信或傳訊息給專案的原作說「我覺得你的專案很有趣，開個權限給我吧，我來幫你加
一些功能」。想想看，如果你是原作者，有個不知道哪裡來的人叫你開權限給他，你願意嗎？

在 GitHub 上有個有趣的機制：

1. 先複製 (Fork) 一份原作的專案到你自己的 GitHub 帳號底下。
2. 因為這個複製回來的專案已經在你自己的 GitHub 帳號下，所以你就有完整的權限，想怎麼
改就怎麼改。
3. 改完後，先推回 (Push) 你自己帳號的專案。
4. 然後發個通知，讓原作者知道你有幫忙做了一些事情，請他看一下。
5. 原作者看完後說「我覺得可以」，然後就決定把你做的這些修改合併 (Merge) 到他的專
案裡。

其中，第 4 步的那個「通知」，就是發一個請原作來拉回去 (Pull) 的請求 (Request)，稱之
Pull Request，簡稱 PR。

小提示

Fork 這個字在這邊翻譯成「複製」並不是這個字的原意，Fork 的中文翻譯是「分叉」、
「叉子」，在技術圈來說這個詞使用的情境是「原作的做的不夠好，其它人覺得可以做得
更好或是想加入一些個人喜好的功能而修改出另外的版本」，大概就是漫畫與同人誌差不
多的概念吧。不過為了讓大家比較容易理解，這裡還是會使用「複製」做為 Fork 的翻
譯。

實際演練

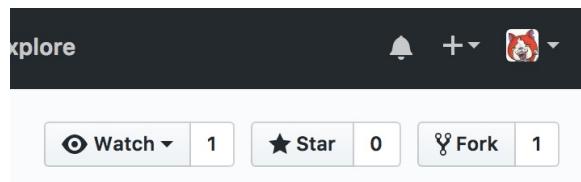
前情提要

來實際演練一下，不過因為這個演練自己一個人來，所以得要開另一隻分身來模擬一下：

- 專案網址：<https://github.com/kaochenlong/dummy-git>
- 角色 A，也就是專案的原作者：<https://github.com/kaochenlong>
- 角色 B，就是想要幫忙的路人：<https://github.com/eddiekao>

第一步：複製 (Fork) 專案

角色 B，先到專案網址的頁面，看一下右上角的那排按鈕：



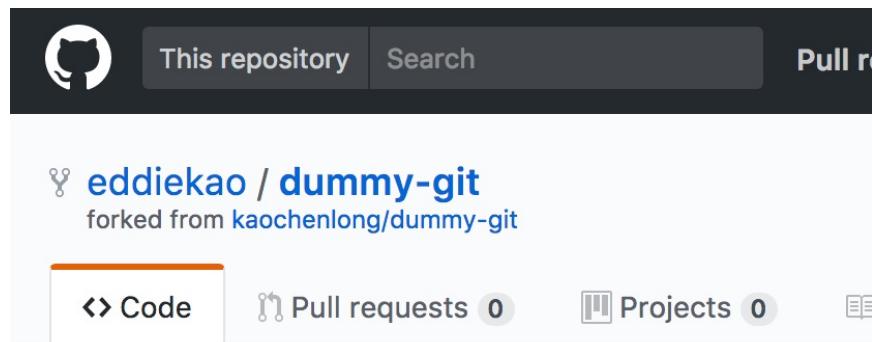
這三個按鈕的功能分別是「Watch（關注）」、「Star（給星星）」以及「Fork（複製一份到你的帳號）」，請按下 Fork 按鈕，接著會看到一個像影印機的畫面：

Forking kaochenlong/dummy-git

It should only take a few seconds.



表示正在把原作的專案，複製一份到你的帳號底下。完成後，注意看一下畫面的左上角：



現在這個專案的確是已經放到角色 B 的帳號下，而且標註「Forked from...」是來自角色 A。這也表示，你對放在自己帳號底下的這個專案有完整的存取權限了。

第二步：Clone 回來修改

如果忘記 Clone 指令，請見「[從伺服器上取得 Repository](#)」章節介紹。

```
$ git clone https://github.com/eddiekao/dummy-git.git
```

```
Cloning into 'dummy-git'...
remote: Counting objects: 47, done.
remote: Total 47 (delta 0), reused 0 (delta 0), pack-reused 47
Unpacking objects: 100% (47/47), done.
```

要確認 Clone 的網址是角色 B 的專案，不要 Clone 錯了。

接著就是開始進行修改，例如我在 `index.html` 檔案加上了一行「Git is a good tool for every developer :)」。接著就是一般的 Git Commit 動作了：

```
$ git add index.html

$ git commit -m "update index"
[master ac341ae] update index
 1 file changed, 1 insertion(+)
```

忘記怎麼使用 `git add` 或是 `git commit` 的話，可參閱「[把檔案交給 Git 控管](#)」章節說明。

第三步：Push 回你自己的專案

因為剛剛我們是用 Clone 下來的，Clone 會自動幫我們設定好 upstream，所以接下來只要執行 `git push` 指令而不需另外指定參數：

```
$ git push
Username for 'https://github.com': eddiekao
Password for 'https://eddiekao@github.com':
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 385 bytes | 385.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/eddiekao/dummy-git.git
 fd7cd38..ac341ae master -> master
```

因為我是選用 HTTPS 的方式，所以需要輸入 GitHub 的使用者帳號及密碼。忘記怎麼 Push 的話，可參考「[Push 上傳到 GitHub](#)」章節。

第四步：發 PR 紿原作者

回到自己的專案頁面，可以找到一個「New pull request」的按鈕：

No description, website, or topics provided.

Add topics

Branch: master ▾ New pull request

This branch is 1 commit ahead of kaochenlong:master.

按下「Create pull request」後，可開始填寫 PR 的相關資訊，讓作者知道你這個 PR 大概做了什麼事：

Pull requests Issues Marketplace Explore

kaochenlong / dummy-git Watch ▾ 1

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights ▾

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across 1

base fork: kaochenlong/dummy-git ▾ base: master ▾ ... head fork: eddiekao/dummy-git ▾ compare: master ▾

✓ Able to merge. These branches can be automatically merged.

Create pull request Discuss and review the changes in this comparison with others.

在這裡可以選擇要發 PR 到原作的哪個分支：

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across f](#)

The screenshot shows the GitHub interface for creating a pull request. At the top, there are dropdown menus for 'base fork' (kaochenlong/dummy-git), 'base' (master), 'head fork' (eddiekao/dummy-git), and 'compare' (master). A green checkmark indicates that the branches can be automatically merged. Below this, a message says 'Able to merge. These branches can be automatically merged.' On the left, there's a cartoon cat icon. The main area has tabs for 'Write' (selected) and 'Preview'. The preview window shows a message: '這超厲害的，快收下吧，不用謝了!' (This is super cool, just take it, no need to thank!). There are rich text editing tools above the message. Below the message, there's a placeholder for attachments: 'Attach files by dragging & dropping, selecting them, or pasting from the clipboard.' At the bottom, there's a checkbox for 'Allow edits from maintainers' and a green 'Create pull request' button.

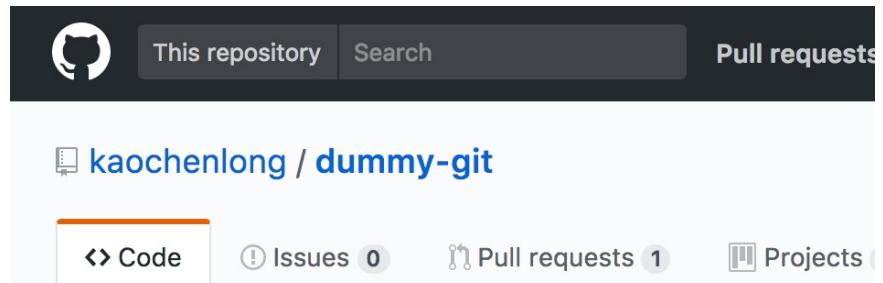
填寫完畢後，按下「Create pull request」按鈕後，即算完成送出 PR：

update index #1

The screenshot shows the details of a pull request titled 'update index #1'. It was opened by 'eddiekao' and merged into 'kaochenlong:master' from 'eddiekao:master'. The pull request has 0 conversations, 1 commit, and 1 file changed. A comment from 'eddiekao' is visible: '這超厲害的，快收下吧，不用謝了!' (This is super cool, just take it, no need to thank!). Below the commit, there's a link to 'update index'. At the bottom, there's a note: 'Add more commits by pushing to the master branch on eddiekao/dummy-git.' A green box highlights a message: 'This branch has no conflicts with the base branch. Only those with write access to this repository can merge pull requests.' with a checkmark icon.

第五步：原作收下 PR

切換回角色 A，也就是原作者，便可以在專案的頁面看到 Pull requests 的數量增加了：



The screenshot shows a GitHub repository page for 'kaochenlong / dummy-git'. The top navigation bar has tabs for 'This repository', 'Search', and 'Pull requests'. The 'Pull requests' tab is selected, showing a count of 1. Below the navigation, there's a section for the repository details: 'No description, website, or topics provided.' and a 'Add topics' button. At the bottom, there's a code diff view showing a single commit:

```
5      <title>Home</title>
6      </head>
7      <body>
8 +    Git is a good tool for every developer :)
9      </body>
10     </html>
```

如果覺得可以收，就只要按下 Merge pull request 按鈕後，就會合併這次的 Commit：

update index #1

Merged kaochenlong merged 1 commit into kaochenlong:master from eddiekao:m

Conversation 0 Commits 1 Files changed 1

 **eddiekao** commented 2 minutes ago First-timer

這超厲害的，快收下吧，不用謝了！

 update index

 kaochenlong merged commit **b5a956d** into kaochenlong:master just now

發送 PR，原作確認沒問題就收下來，這種發 PR 的方式，在開源界是很常見的貢獻程式碼的方法。

應用情境

不只在開源專案，即使在企業內部的專案也適合使用發 PR 的方式來進行開發。在開發產品的時候，通常會挑選固定一個分支做為可以上線的正式版本分支，慣例常會使用 `master` 或是 `production` 分支做為正式分支。當越多人參與同一個專案，讓每個人都可以 Commit 到專案正式上線的分支不是個好的做法，這時候便可使用 PR 方式來進行。

每位開發者都先從公司的專案 Fork 一份到自己的帳號下，待功能完成後再發 PR 回公司的專案。負責管理這個專案的人收到 PR 後，進行 Code Review 並確認無誤後便可進行合併，這樣一來可讓這個產品分支處於隨時都是可上線的狀態。

也許一開始會覺得這樣很麻煩，但隨著越多人一起協同開發，就越需要訂定規則，在後面 Git Flow 章節所要介紹的也是其中一款開發流程。

【狀況題】怎麼跟上當初 fork 專案的進度？

在上個章節「[與其它開發者的互動 - 使用 Pull Request \(PR\)](#)」介紹了如何發送 Pull Request (PR)，但其它人也可能會發送 PR，當原作也收了別人的 PR 之後，該專案的進度就比在你自己帳號底下的進度還要前面了。如果你想要讓你帳號底下這個 Fork 過來的專案跟上原作專案目前的進度，GitHub 網站上目前並沒有提供方便的按鈕功能可以做這件事，但你可以透過以下兩種做法來達成這個目的：

第一招：砍掉重練

這招很簡單，就是把 Fork 過來的專案砍掉，再重新 Fork 一次，這樣保證就會是最新版本。不要笑，雖然這招感覺技術力很低，但這招很好用，完全不需要打任何程式碼或指令就可以完成。我也知道其實不少人會用這個方式在進行同步…

第二招：跟上游同步

比較有技術力一點的做法（其實也是比較正統的做法），就是把原作的專案設定成上游專案，Fetch 回來之後再自己手動合併。

第一步：設定原作的遠端節點

舉例來說，這是 Fork 過來的專案：

```
$ git remote -v
origin  https://github.com/eddiekao/dummy-git.git (fetch)
origin  https://github.com/eddiekao/dummy-git.git (push)
```

`git remote` 指令加上 `-v` 參數後可以看到更完整的資訊。從這裡可以看得出來目前這個專案只有設定一個遠端節點 `origin`。接下來我要幫它加上另一個遠端節點，這個遠端節點指的位置就是原作的專案：

```
$ git remote add dummy-kao https://github.com/kaochenlong/dummy-git.git
```

其實大部份的資料都會教你使用 `upstream` 做為原作遠端節點的名字，但為避免大家跟之前在「[Push 上傳到 GitHub](#)」章節介紹的 `upstream` 搞混，所以這裡我故意使用 `dummy-kao` 做為指向原作的遠端節點。這時候在這個專案應該就有 2 個遠端節點了，一個是原來的 `origin`，一個是原作的 `dummy-kao`：

```
$ git remote -v
dummy-kao    https://github.com/kaochenlong/dummy-git.git (fetch)
dummy-kao    https://github.com/kaochenlong/dummy-git.git (push)
origin      https://github.com/eddiekao/dummy-git.git (fetch)
origin      https://github.com/eddiekao/dummy-git.git (push)
```

第二步：抓取原作專案的內容

接下來，就是使用 Fetch 指令來取得原作專案最新版的內容：

```
$ git fetch dummy-kao
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 1), reused 3 (delta 1), pack-reused 1
Unpacking objects: 100% (4/4), done.
From https://github.com/kaochenlong/dummy-git
 * [new branch]      features/mailer      -> dummy-kao/features/mailер
 * [new branch]      features/mailer-plus -> dummy-kao/features/mailер-plus
 * [new branch]      features/mailer_pro   -> dummy-kao/features/mailер_pro
 * [new branch]      features/member     -> dummy-kao/features/member
 * [new branch]      master            -> dummy-kao/master
```

如果忘了 Fetch 指令是做什麼的，請參閱「[Pull 下載更新](#)」章節介紹。還記得 Fetch 下來之後，在本地的遠端分支會往前移動嗎？如果想要跟上剛抓下來的這些進度的話，便是使用 Merge 指令（或是要用 Rebase 也可）：

```
$ git merge dummy-kao/master
Updating ac341ae..689b015
Fast-forward
 contact.html | 2 ++
 1 file changed, 2 insertions(+)
```

這樣一來，你本機的進度就跟原作的一樣了。

第三步：推回自己的專案

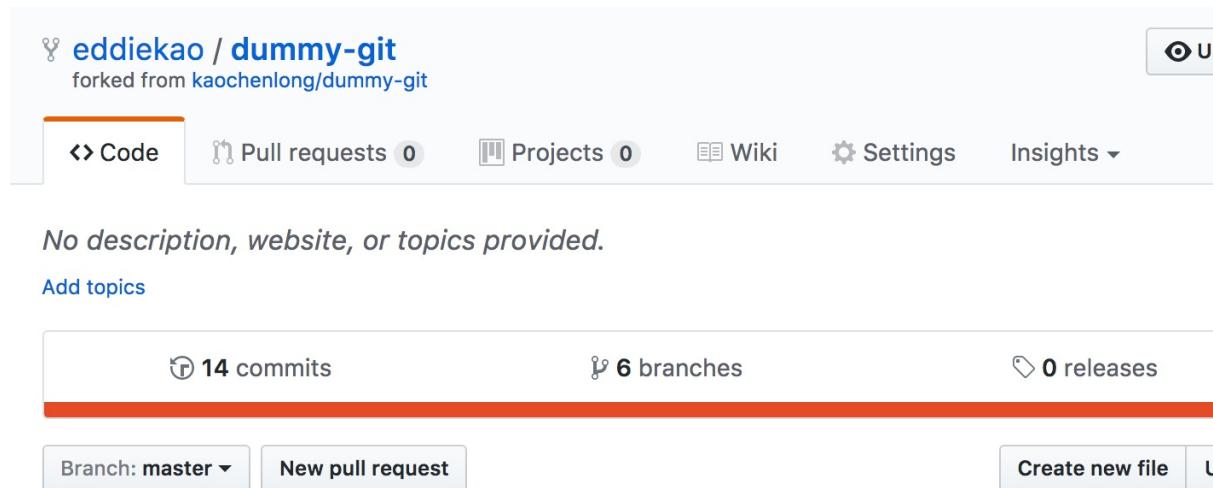
這個步驟要不要做就看你自己決定了，畢竟在你電腦上已經是最新版本了，如果你希望你在 GitHub 上那個 Fork 的專案也跟到最新版，只要推上去就行了：

```
$ git push origin master
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 596 bytes | 596.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/eddiekao/dummy-git.git
 ac341ae..689b015  master -> master
```

這樣一來，你電腦裡的專案，以及在 GitHub 上從原作那邊 Fork 過來的專案都會是最新進度了。

【狀況題】怎麼刪除遠端的分支？

老實說，這是個有趣的題目，有趣的點於它的刪除指令，待會來看看是哪裡有趣，先看看怎麼樣從 GitHub 網站來做這件事。打開 GitHub 網站的專案頁面：



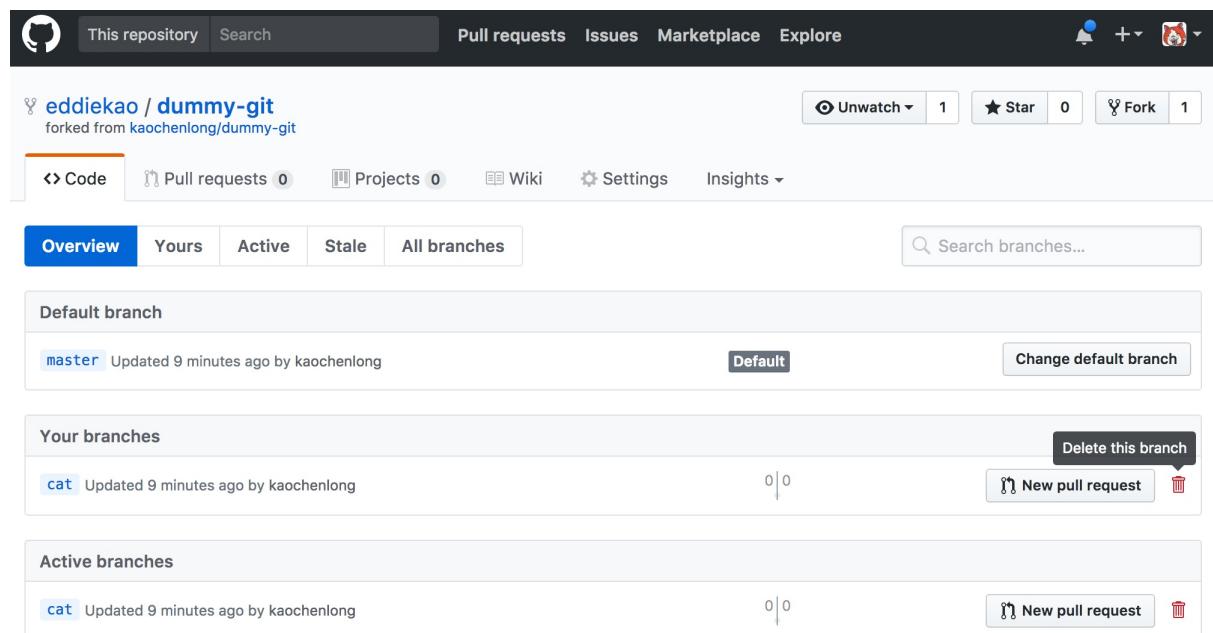
No description, website, or topics provided.

Add topics

14 commits 6 branches 0 releases

Branch: master ▾ New pull request Create new file

點擊畫面中間的分支列表可以看到目前所有的分支，在分支旁邊有一顆紅色垃圾筒的圖示：



This repository Search Pull requests Issues Marketplace Explore

eddiekao / dummy-git forked from kaochenlong/dummy-git

Unwatch 1 Star 0 Fork 1

Code Pull requests 0 Projects 0 Wiki Settings Insights

Overview Yours Active Stale All branches Search branches...

Default branch

master Updated 9 minutes ago by kaochenlong Default Change default branch

Your branches

cat Updated 9 minutes ago by kaochenlong 0|0 Delete this branch New pull request

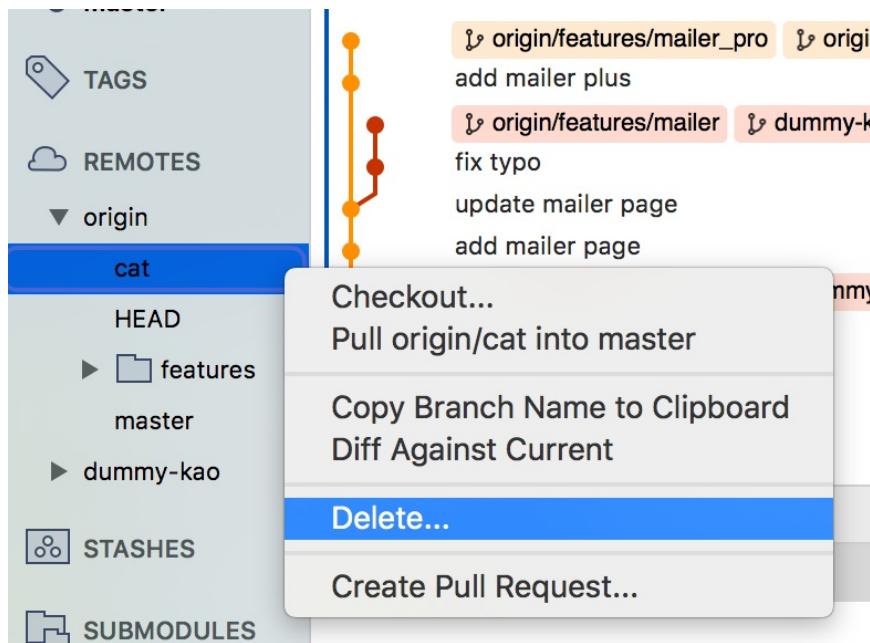
Active branches

cat Updated 9 minutes ago by kaochenlong 0|0 New pull request

只要按下去就可以刪掉這個分支了，相當容易。

如果你對於合併過的分支是否要留著有所疑慮，請參閱「[【常見問題】合併過的分支要留著嗎？](#)」章節說明。

若要使用 SourceTree 來刪除遠端分支，請在左邊的選單找到「REMOTES」，在你想要刪除的分支上按滑鼠右鍵：



選擇「Delete...」功能後會跳出一對話框，確認無誤按下 OK 鈕之後便可刪除遠端分支。

如果是使用指令：

```
$ git push origin :cat
To https://github.com/eddiekao/dummy-git.git
 - [deleted]          cat
```

是的，你沒看錯，就是在分支前面加上冒號，而且是用 Push 指令來刪除遠端分支，這就是我前面提到有趣的點。

但仔細想想好像也不是那麼不合理，還記得我們在「[Push 上傳到 GitHub](#)」章節提到這樣的指令：

```
$ git push origin master:cat
```

意思就是要把本地的 `master` 分支推上去之後，在 Server 上建立 `cat` 分支。如果把這個指令前面的 `master` 拿掉：

```
$ git push origin :cat
```

就像是推了空的內容去更新線上的 `cat` 分支的內容，也算是變相的把該分支刪除。只是使用 Push 指令刪分支，有一點不直覺而已。

【狀況題】聽說 git push -f 這個指令很可怕，什麼情況可以使用它呢？

git push -f 這個指令真的令人又愛又恨。愛的是它可以完全無視先來後到的規則，一切以你為主，你推什麼就是什麼，別人之前推的內容都會被無視，就像《權力遊戲》裡 Daenerys Targaryen 騎著她的飛龍大喊「Dracarys！」，怒火燒盡九重天的爽快感！

但恨的就是如果團隊裡有人沒有先知會大家就突然使用這樣的大絕招，你就是被 Dracarys 燒到的那個人，身為隊友的你應該會想把放火的人吊起來打。

所以，這個指令使用請小心，請確定你清楚你在做什麼（建議使用指令前先睡飽八小時以免發生憾事）。

用槍時機

整理歷史紀錄

有時候專案 Commit 的歷史紀錄真的太亂了，想要大刀闊斧的來整頓一下，於是你就用了 Rebase 指令（關於如何使用 Rebase 指令可參閱「修改歷史紀錄」相關章節）。因為 Rebase 等於是修改已經發生的事實，所以正常來說是推不上去的。

這時候就可使用 Force Push 來解決這個問題，但請使用前知會一下跟你同一個專案的隊友，請他們到時候以你這份進度為主。

只用在自己身上

我自己在工作的時候，通常會開一個分支去做，但做完發現 Commit 太過瑣碎，便會想使用 Rebase 來整理一下這個分支。雖然 Rebase 是修改歷史沒錯，但因為這個歷史影響的範圍只有我自己這個分支，所以並不會影響其它人正常使用：

```
$ git push -f origin features/my_branch
```

這樣只會強制更新 `features/my_branch` 這個分支的內容，不會影響其它分支。

啟動保護機制

但總是有人可能不小心使用了 `-f` 參數來 Push，GitHub 網站有提供保護機制，可以避免某個分支被 Force Push。請到專案的「Settings」頁籤，左邊選擇「Branches」，接著可以選擇想要保護的分支：

The screenshot shows the GitHub repository settings for 'kaochenlong / dummy-git'. The 'Branches' tab is selected. Under 'Default branch', the 'master' branch is chosen. Under 'Protected branches', a dropdown menu is open, showing a list of branches: 'features/mailer-plus', 'features/mailer_pro', 'features/mailer', 'features/member', and 'master'. The 'master' branch is highlighted with a blue background. At the bottom right of the dropdown, there are links for 'Contact GitHub', 'API', 'Training', 'Shop', 'Blog', and 'Help'.

勾選你想要保護的選項：

Branch protection for master

Protect this branch

Disables force-pushes to this branch and prevents it from being deleted.

Require pull request reviews before merging

When enabled, all commits must be made to a non-protected branch and submitted via a pull request with at least one approved review and no changes requested before it can be merged into **master**.

Require status checks to pass before merging

Choose which **status checks** must pass before branches can be merged into **master**. When enabled, commits must first be pushed to another branch, then merged or pushed directly to **master** after status checks have passed.

Include administrators

Enforce all configured restrictions for administrators.

Save changes

完成之後，這個分支就不能被 Force Push 了，甚至連刪除也得先來這裡解除封印才行。

萬一被蓋掉了，救得回來嗎

救回來其實還滿簡單的，就是換你或是其它有之前進度的隊友，再次進行 `git push -f` 指令一次，把正確的內容強迫推上去，蓋掉前一次的 `git push -f` 所造成的災難。

使用 GitHub 免費製作個人網站

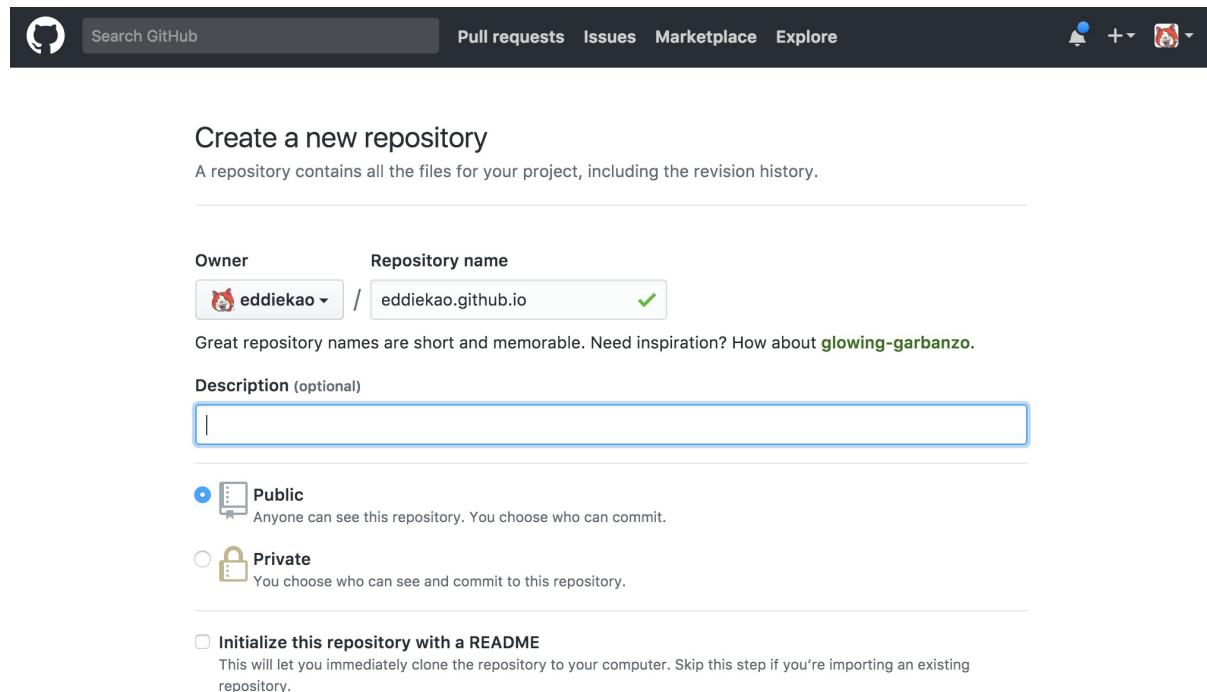
GitHub 除了提供免費的 Git Server 外，如果推上去的分支的名字剛好叫做 `gh-pages`，你可以用 GitHub 當做靜態檔案的伺服器，它比一般外面虛擬主機要便宜得多，也安全得多。不過也有一些限制：

1. 僅呈現靜態頁面內容，如果是什麼 PHP 或是 ASP 的它不會理你。
2. 不支援 `.htaccess` 之類的設定檔，所以無法設定使用者密碼。
3. 僅能使用 Git 上傳，沒有什麼 FTP 之類的東西。
4. 不像 Repository 有 Private 的設定，所有的 GitHub Pages 都是公開的，甚至放在 Private 專案裡，頁面也是公開的。

整體上來說，GitHub Pages 的優點還是多於缺點，至少它穩定、安全又沒收你錢。接著就試試看如何把頁面放上去吧。

新增專案

首先在 GitHub 上開一個全新的專案：



The screenshot shows the GitHub interface for creating a new repository. At the top, there's a dark header bar with the GitHub logo, a search bar, and navigation links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. On the right side of the header, there are user profile icons and dropdown menus. Below the header, the main form has a title 'Create a new repository' and a subtitle explaining that a repository contains all files for a project, including revision history. The 'Owner' field is set to 'eddiekao' and the 'Repository name' field is filled with 'eddiekao.github.io', which is highlighted with a green checkmark. A note below suggests using short and memorable names like 'glowing-garbanzo'. The 'Description (optional)' field is empty. Under 'Visibility', the 'Public' option is selected, indicated by a blue radio button and the text 'Anyone can see this repository. You choose who can commit.' The 'Private' option is also available. At the bottom of the form, there's a checkbox for 'Initialize this repository with a README', with a note explaining it allows cloning the repository to a computer. The entire process is set against a white background with blue and grey UI elements.

在專案名稱的地方，填寫「`username.github.io`」，這個 `username` 指的是自己的 GitHub 帳號，所以我這邊是填 `eddiekao.github.io`。

接下來，我先找一個空的目錄，建立 `index.html`，內容如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>你好，GitHub</title>
  </head>
  <body>
    <h1>覺得厲害</h1>
  </body>
</html>
```

完成之後，就是 `git add` 跟 `git commit` 的基礎指令了（可參考「[把檔案交給 Git 控管](#)」章節）：

```
$ git add index.html

$ git commit -m "add index"
[master (root-commit) 80450b2] add index
 1 file changed, 10 insertions(+)
 create mode 100644 index.html
```

接下來就是一般的 Push 了，如果忘記怎麼操作可參閱「[Push 上傳到 GitHub](#)」章節。

```
$ git remote add origin https://github.com/eddiekao/eddiekao.github.io.git

$ git push -u origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 327.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/eddiekao/eddiekao.github.io.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

順利推上去之後，回到專案的頁面看看應該可以看到東西已經推上去了：

No description, website, or topics provided.

Add topics

1 commit 1 branch 0 releases 1 contributor

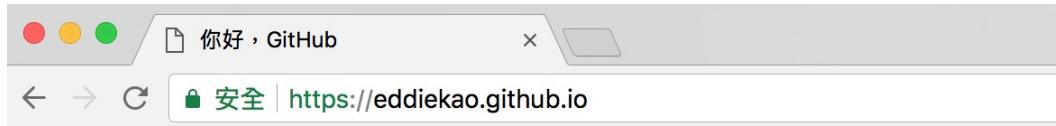
Branch: master New pull request Create new file Upload files Find file Clone or download

kaochenlong add index Latest commit 80450b2 3 minutes ago

index.html add index 3 minutes ago

Add a README

這時候，輸入網址 <https://eddiekao.github.io/> 便可連上頁面：



覺得厲害

另外，坊間有其它像是 [Jekyll](#) 或是 [Octopress](#) 之類的好用套件，可以使用 Markdown 語法編寫，並幫你轉成 HTML 格式或是產生整個 Blog，甚至有方便的指令可一行指令自動上傳到 GitHub Pages 上。

本書所有內容(<https://gitbook.tw/>)以及我個人的 Blog(<https://kaochenlong.com/>)就是透過這些套件完成的，同時也是放在 GitHub Pages 上，又安全又穩定。詳情請見這些套件的官方網站介紹。

- Jekyll: <https://jekyllrb.com/>
- Octopress: <http://octopress.org/>

客制化網址

覺得原來的網址不好記嗎？沒關係，GitHub Pages 有支援客制化網址，只要簡單兩個步驟就可完成：

1. 在該專案的根目錄放一個檔名為 `CNAME` 的檔案，內容只要放你想要客制化的那個網址。
2. 請幫你管理網域的人，幫你設定一組 `CNAME` 指到 `eddiekao.github.io.` 即可。（你的網域管理人應該知道什麼是 `CNAME`）

參考網址：<https://pages.github.com/>

【冷知識】一定要有 GitHub 才能得到別人更新的檔案嗎？

大部份人的觀念，檔案如果沒有上傳到 GitHub，其它人是要怎麼更新？難道是要用 Email 寄來寄去嗎？

嗯，你猜對了，雖然現在有 GitHub 很方便沒錯，只要 Push、Pull 幾個簡單的指令檔案就同步，但在以前的時候的確就是用 Email 來寄送更新檔案。

製作更新檔

來試一下怎麼製作所謂的「更新檔（Patch）」。假設目前的歷史紀錄是這樣：

```
$ git log --oneline
fd7cd38 (HEAD -> master, origin/master, origin/HEAD) Update about.html
2eb8fea add readme
953cbd9 update info page
15202a1 add info page
b8ac91f add contact page
9a0233e add about page
1bbf412 update index.html
59dbbc9 init commit
```

然後我先很快的做了 2 次 Commit，所以現在歷史變這樣：

```
$ git log --oneline
6e6ed76 (HEAD -> master) add product page
6aba968 update info.html
fd7cd38 (origin/master, origin/HEAD) Update about.html
2eb8fea add readme
953cbd9 update info page
15202a1 add info page
b8ac91f add contact page
9a0233e add about page
1bbf412 update index.html
59dbbc9 init commit
```

接下來，我使用 `git format-patch` 指令來產生幾個更新檔

```
$ git format-patch fd7cd38..6e6ed76
0001-update-info.html.patch
0002-add-product-page.patch
```

後面的參數 `fd7cd38..6e6ed76` 表示會產生從 `fd7cd38` 這個 Commit 之後（不包括本身），直到 `6e6ed76` 這個 Commit 為止的更新檔。如果是這樣：

```
$ git format-patch -2  
0001-update-info.html.patch  
0002-add-product-page.patch
```

後面那個 `-2` 表示「請幫我產生最新的兩次 Commit」的更新檔。因為這個指令會直接在當下目錄產生 Patch 檔，如果加上 `-o` 參數可以指定產生的更新檔的位置：

```
$ git format-patch -2 -o /tmp/patches  
/tmp/patches/0001-update-info.html.patch  
/tmp/patches/0002-add-product-page.patch
```

使用更新檔

要使用 `format-patch` 指令產出的修正檔，使用的是 `git am` 指令：

```
$ git am /tmp/patches/*  
Applying: update info.html  
Applying: add product page
```

你可以一次使用一個更新檔，或是像這樣一口氣把剛剛產出在 `/tmp/patches` 目錄的更新檔全部用上，Git 會依據檔案的名字依序一個一個套在現有的專案上了。

Git Flow 是什麼？為什麼需要這種東西？

當在同一個專案一起開發的人數越來越多，如果沒有訂好規矩，每個人的 Commit 習慣可能都不同，放任大家隨便 Commit 的話遲早會造成災難。

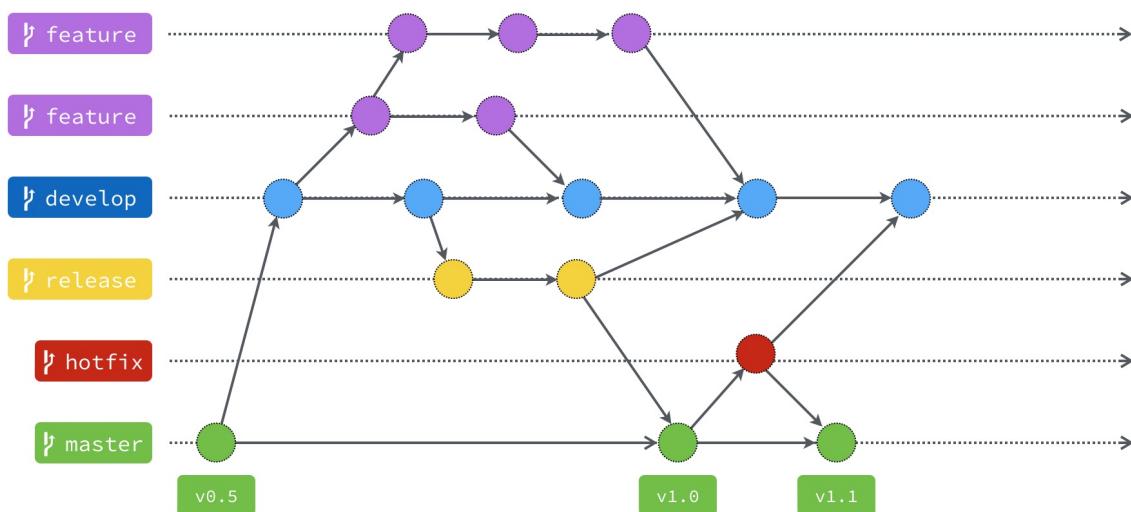
在 2010 年的時候，就有人提出了一套流程，或說是訂了一套規矩讓大家可以遵守：

網址：<http://nvie.com/posts/a-successful-git-branching-model/>

不過因為這套流程是 2010 年提出的，到現在也已經九年了，這幾年來也陸續有其它優秀的開發流程，例如 GitHub Flow、Gitlab Flow 等流程，我們這邊僅以 Git Flow 做為介紹。

分支應用情境

根據 Git Flow 的建議，主要的分支有 master 、 develop 、 hotfix 、 release 以及 feature 這五種分支，各種分支負責不同的功能。其中 Master 以及 Develop 這兩個分支又被稱做長期分支，因為他們會一直存活在整個 Git Flow 裡，而其它的分支大多會因任務結束而被刪除。



Master 分支

主要是用來放穩定、隨時可上線的版本。這個分支的來源只能從別的分支合併過來，開發者不會直接 Commit 到這個分支。因為是穩定版本，所以通常也會在這個分支上的 Commit 上打上版本號標籤。

Develop 分支

這個分支主要是所有開發的基礎分支，當要新增功能的時候，所有的 Feature 分支都是從這個分支切出去的。而 Feature 分支的功能完成後，也都會合併回來這個分支。

Hotfix 分支

當線上產品發生緊急問題的時候，會從 Master 分支開一個 Hotfix 分支出來進行修復，Hotfix 分支修復完成之後，會合併回 Master 分支，也同時會合併一份到 Develop 分支。

為什麼要合併回 Develop 分支？如果不這麼做，等到時候 Develop 分支完成並且合併回 Master 分支的時候，那個問題就又再次出現了。

那為什麼一開始不從 Develop 分支切出來修？因為 Develop 分支的功能可能尚在開發中，這時候硬是要從這裡切出去修再合併回 Master 分支，只會造成更大的災難。

Release 分支

當認為 Develop 分支夠成熟了，就可以把 Develop 分支合併到 Release 分支，在這邊進行算是上線前的最後測試。測試完成後，Release 分支將會同時合併到 Master 以及 Develop 這兩個分支上。Master 分支是上線版本，而合併回 Develop 分支的目的，是因為可能在 Release 分支上還會測到並修正一些問題，所以需要跟 Develop 分支同步，免得之後的版本又再度出現同樣的問題。

Feature 分支

當要開始新增功能的時候，就是使用 Feature 分支的時候了。Feature 分支都是從 Develop 分支來的，完成之後會再併回 Develop 分支。

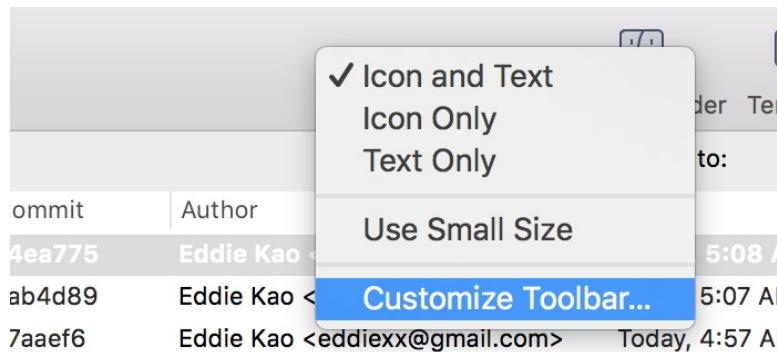
下一章，就來看看怎麼使用 Git Flow 吧。

使用 Git Flow

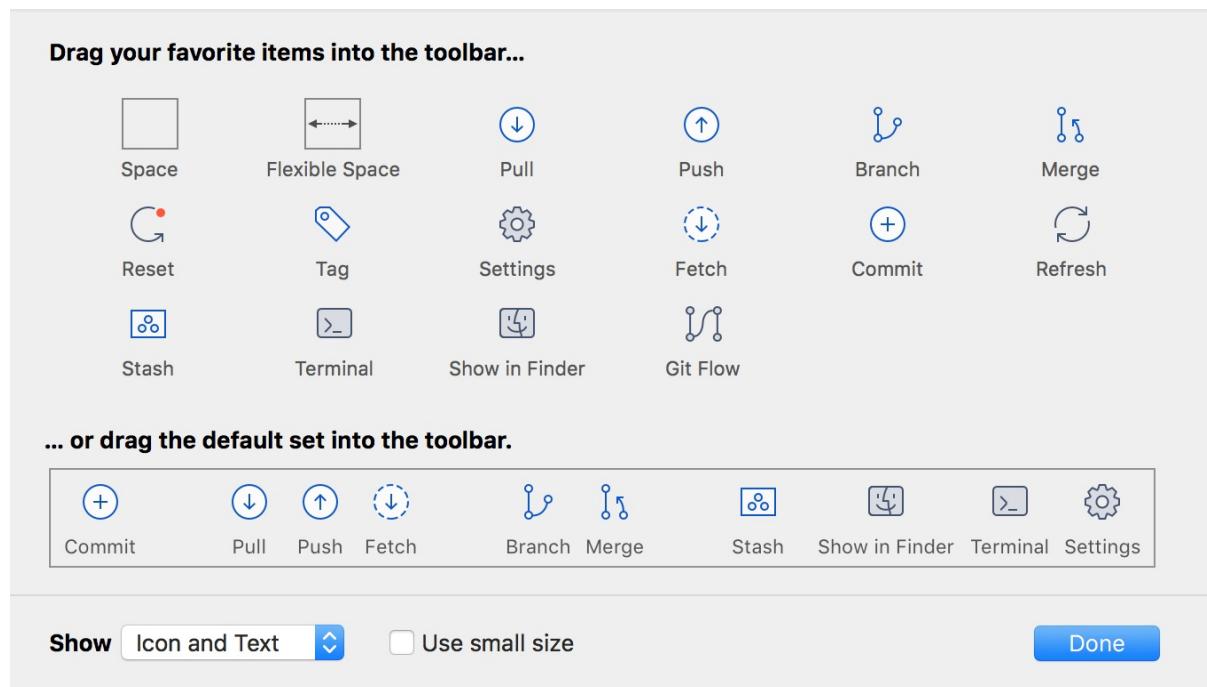
在 Git Flow 的 [GitHub 專案上](#)，有介紹如何安裝以及使用 Git Flow，但因 SourceTree 本身有內建支援 Git Flow，所以本章節我將直接使用 SourceTree 介紹。

設定 Git Flow 按鈕

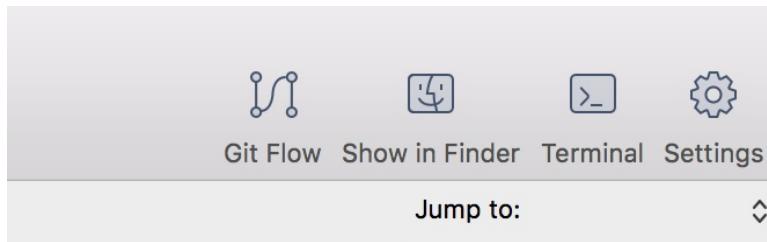
如果你在 SourceTree 上方看不到 Git Flow 的按鈕，請在工具箱按滑鼠右鍵：



選擇「Customize Toolbar...」

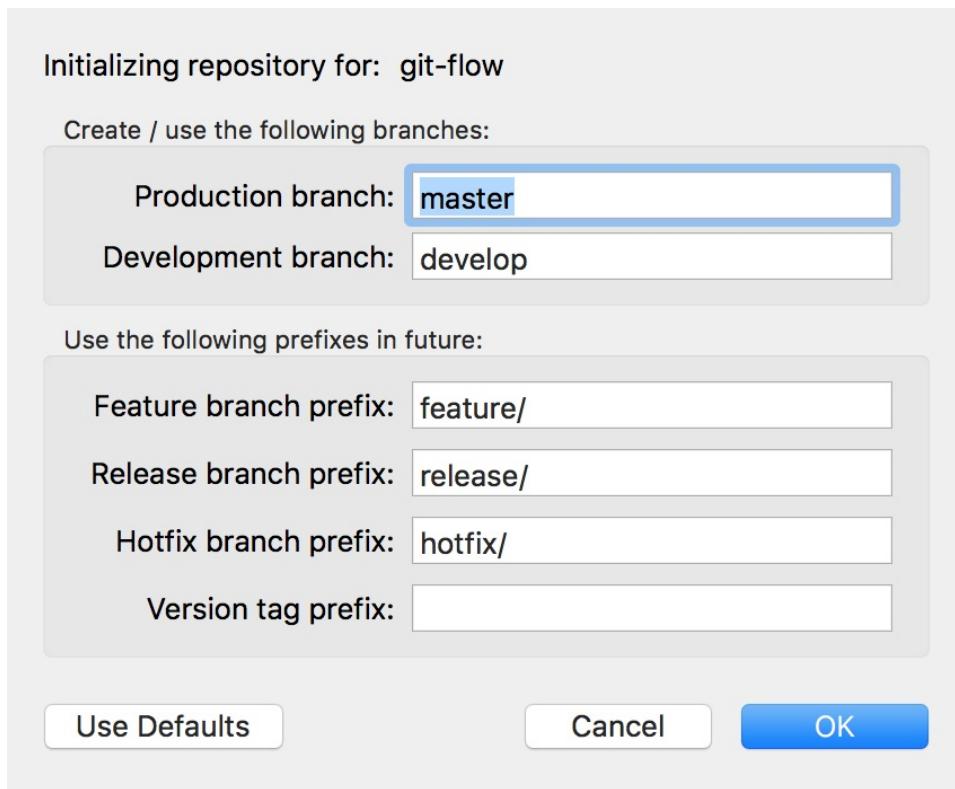


把「Git Flow」的按鈕拖拉到工具箱，像這樣：



Git Flow 初始化

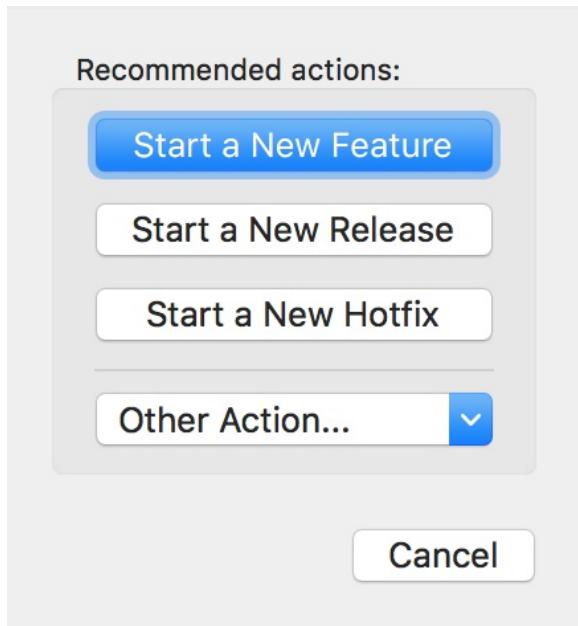
這邊的初始化不是使用 Git 一開始的那個 `git init` 喔，這個只是單純為了讓原本的專案認得 Git Flow 的指令而已。請按下上方工具箱的「Git Flow」按鈕，如果是第一次按的話，會進入初始化設定：



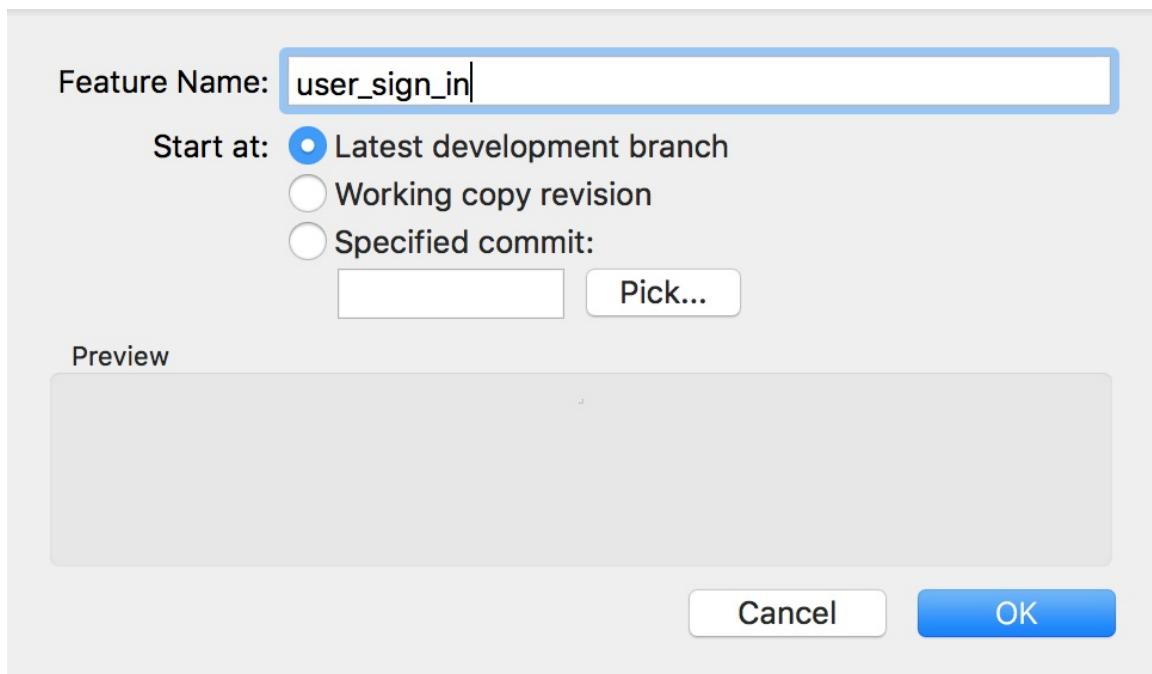
這個地方就是設定在上個章節提到的五種分支，基本上使用預設值就行了，按下 OK 鈕即可完成 Git Flow 初始化。

開始加功能

要開始加功能，請一樣按下上方工具箱那顆「Git Flow」按鈕後會跳出這個畫面：



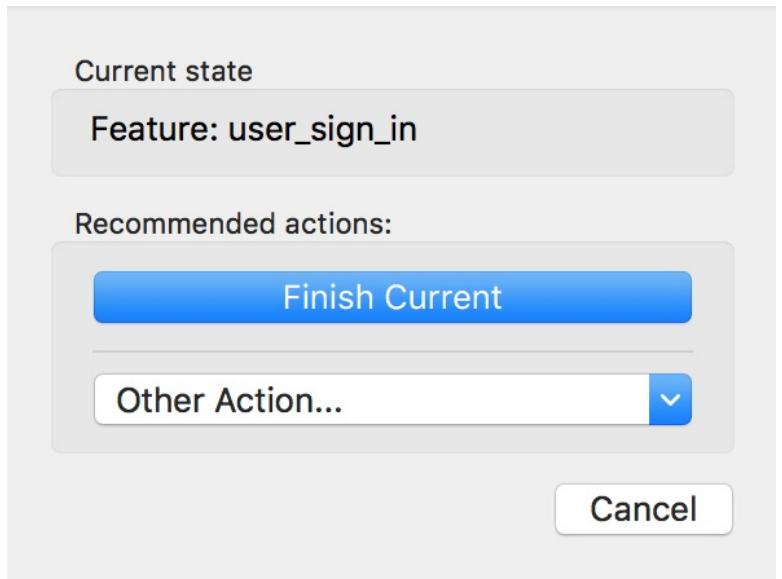
有一些選項可以選，因為要加功能，所以這裡選擇「Start a New Feature」，接著就是填寫這個 Feature 分支的名字：



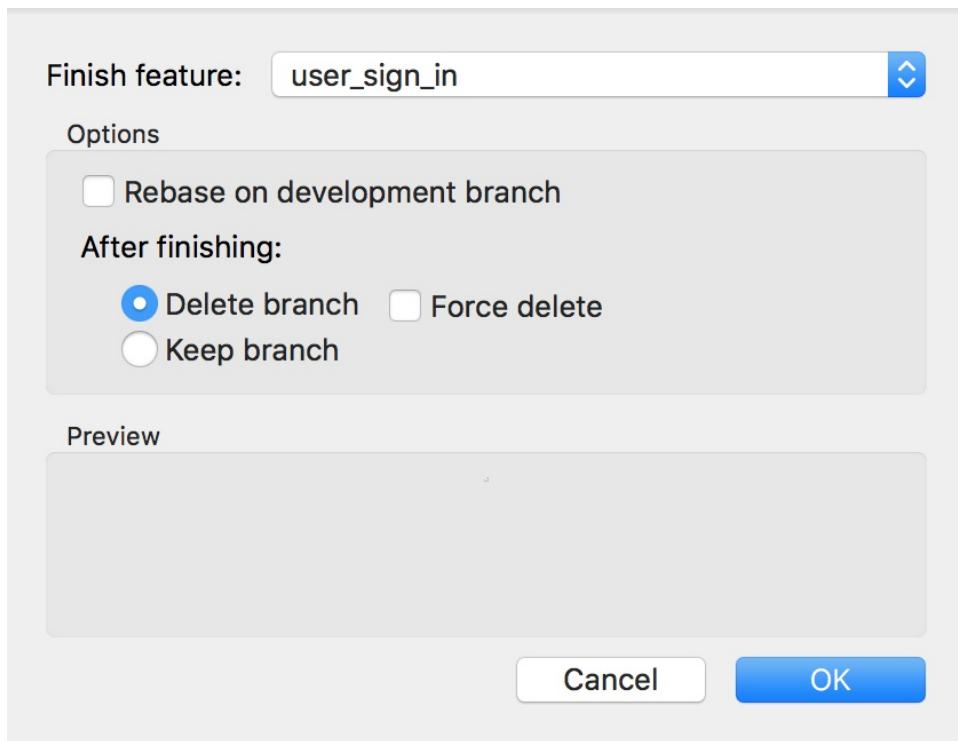
接下來，就是開始努力的工作...工作...工作...

完成功能

好不容易做完了，接下來又是按下那個「Git Flow」按鈕：



選擇「Finish Current」，表示要完成這一個 Feature 分支的進度。按下後會再出現這個對話框：



這些選項應該都不難懂，看是要用一般的合併還是使用 Rebase 方式合併，或是合併過的分支要不要留著...我想到這裡，這些問題的答案你應該都很清楚了才是。按下 OK 鈕，就算完成一次的 Feature 分支了。

小結

不管是 Feature、Release 還是 Hotfix 分支，都是一樣的模式進行，所以你必須先知道這幾種分支有什麼不同，在結束的時候會怎麼進行合併...雖然 Git Flow 已經是七年前提出來的流程，越來越多其它優秀的 Flow 也都提出來了，但不管是什麼 Flow，重點是 Flow 是要讓整個開發團隊看的，要讓所有人都遵守同一套流程，開發起來才會順手。

【狀況題】啊，我還沒開分支就 Commit 下去了！

使用 Git 的時候，分支是個非常方便的設計，在工作上不管是功能開發或是使用 Flow 團隊共同協作的時候，可能都有被長官交待要先開分支再進行開發。但偶爾還是會遇到「啊，我忘了開分支...」的情況，這時候你會怎麼做？

1. 把剛才新增或變更的檔案先 copy 一份出來放在別的目錄。
2. 從網路上重新下載（clone）一份專案。
3. 進入剛剛重新下載的專案，建立分支並切換到新的分支。
4. 把剛才 copy 的檔案再放回剛下載的專案。
5. 繼續進行 commit，搞定！

這樣做基本上沒什麼問題，算是「逃避雖可恥但有用」的解決方法，但如果每次有問題就要重新下載一份也太辛苦了。事實上，如果你對分支的觀念理解正確的話，以下這幾個「忘記開分支」的狀況是很容易解決的。

檔案下載：<https://ubin.io/git-branch1>

狀況一：開了分支，但忘記切換分支...

長官有交待，專案要進行開發之前要先開分支，所以就先開一個 `cat` 分支：

```
$ git branch cat
```

開好了分支，準備要開始工作了，Go Go Go！

接著，我新增一個 `cat.html` 檔案，然後編輯了 `hello.html` 檔案的內容，這時候專案的狀態應該會變成這樣：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   hello.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    cat.html
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

一個 Untracked，一個 modified，看起來跟預期的一樣，沒什麼問題。但，在辛苦工作了一、二個小時之後，才驚覺好像一開始的時候就忘記切換分支，這時候有可能有以下幾種情況：

情境 1. 還沒有 Commit...

在上面這個範例裡，只有做了檔案的修改但還沒有進行 Commit，所以不管是 master 分支或是 cat 分支都還是停留在原地（也就是 e12d8ef 這個 Commit）。而且 cat 分支是從 master 分支建立的，所以這兩個分支現在是指向同一個 Commit，也就是說這兩個分支的進度是一樣的。若對分支還不熟悉的話，可再參閱「[對分支的誤解](#)」章節內容。

先跟大家講兩個關於分支的重點：

第一個重點，就是在 Git 的世界，Commit 才是本體，分支或是標籤什麼的都只是浮雲，就算把分支的名字從 master 改成 slave，或是把分支全部刪光光也不會影響到 Commit。

第二個重點，就是在 Git 裡如果有超過一個以上的分支存在的時候，在進行 Commit 或是 Reset 的時候，怎麼決定哪個分支會移動位置呢？很簡單，其實就是被 HEAD 指到的那個分支會移動。

綜合以上這兩個重點，如果還沒有 Commit，不管檔案是不是已經被加至暫存區（Staging Area）都沒關係，這時只要切換分支：

```
$ git checkout cat
M     hello.html
Switched to branch 'cat'
```

先把 HEAD 移到對的分支上，然後進行 Commit 就行了：

```
$ git add --all

$ git commit -m "add cat"
[cat 9ee944a] add cat
2 files changed, 1 insertion(+)
create mode 100644 cat.html
```

這時候的樣子就會變成這樣：

Graph	Commit	Description
	9ee944a	↳ cat add cat
	e12d8ef	↳ master add database.yml in config folder
	85e7e30	add hello
	657fce7	add container
	abb4f43	update index page
	cef6e40	create index page
	cc797cd	init commit

這樣看起來就不像忘記切換分支的樣子了。

情境 2. 已經 Commit 了！

同樣以下載的專案為範例，我先在專案裡做了以下的操作：

```
# 新增檔案 cat1.html
$ touch cat1.html

# 把檔案加至暫存區
$ git add cat1.html

# 進行 Commit !
$ git commit -m "add cat 1"
[master 15b4d3f] add cat 1
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 cat1.html

# 新增檔案 cat2.html
$ touch cat2.html

# 把檔案加至暫存區
$ git add cat2.html

# 進行 Commit !
$ git commit -m "add cat 2"
[master 56fb360] add cat 2
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 cat2.html
```

我在上面的操作中做了 2 次 Commit，然後這時才發現忘了切換分支，這時的歷史紀錄應該是這個樣子：

Graph	Commit	Description
○	56fb360	↳ master add cat 2
●	15b4d3f	add cat 1
●	e12d8ef	↳ cat add database.yml in config folder
●	85e7e30	add hello
●	657fce7	add container
●	abb4f43	update index page
●	cef6e40	create index page
●	cc797cd	init commit

因為沒有切換分支，`HEAD` 指向 `master` 分支，所以在 Commit 之後 `master` 分支往前移動，目前領先了 `cat` 分支 2 次 Commit 的進度。接下來可以有幾種解法：

方法 A. Reset 之後再重新進行 Commit

先取消這兩次 Commit：

```
$ git reset HEAD~2

$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    cat1.html
    cat2.html

nothing added to commit but untracked files present (use "git add" to track)
```

我在進行 Reset 的時候沒有加參數，所以 Reset 指令會以預設的 `--mixed` 模式進行，在這兩次 Commit 所新增的 `cat1.html` 跟 `cat2.html` 檔案都會在暫存區。如果忘記 Reset 指令怎麼用的話，可參閱「[【狀況題】剛才的 Commit 後悔了，想要拆掉重做...](#)」章節介紹。

做到這裡，其實就跟前面的「還沒有 Commit...」的狀況一樣，接下來就可以切換到 `cat` 分支然後接著進行 Commit 了。

方法 B. 移動分支的位置

再次強調，因為分支只是個貼紙的概念，所以在我們這個狀況來說，只要能把貼紙移到對的位置就行了。我們在「[【狀況題】我可以從過去的某個 Commit 再長一個新的分支出來嗎？](#)」章節曾經介紹過這個指令：

```
$ git branch bird 657fce7
```

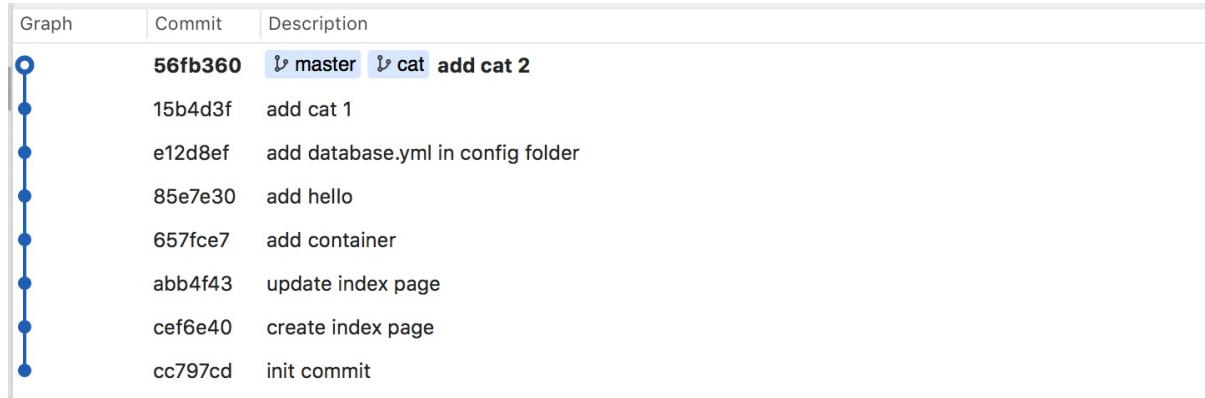
這個指令的意思是「我要在 `657fce7` 這個 Commit 建立一個名為 `bird` 的分支」，同樣的道理，下面這個指令：

```
$ git branch cat master
```

就是「我要在 `master` 分支所指的 Commit 的位置建立一個名為 `cat` 的分支」的意思。不過，在我們這個範例中，`cat` 分支已經存在，所以這時需要額外加上 `-f` 參數來強制完成這件事：

```
$ git branch -f cat master
```

這樣，原本的 `cat` 分支就會被刪除，並且在 `master` 分支的位置重新建立一個新的 `cat` 分支，看起來就會有「把 `cat` 分支移到 `master` 分支的位置」的效果。現在歷史紀錄大概是這個樣子：



把 `cat` 分支移到 `master` 分支的位置後，下一步繼續用同樣的手法，把 `master` 分支移到原本 `cat` 分支的位置。但由於現在 `HEAD` 正在 `master` 分支上，所以即使加上了 `-f` 參數也無法順利成功，必須先使用 `git checkout` 指令把 `HEAD` 換個位置：

```
$ git checkout cat
Switched to branch 'cat'
```

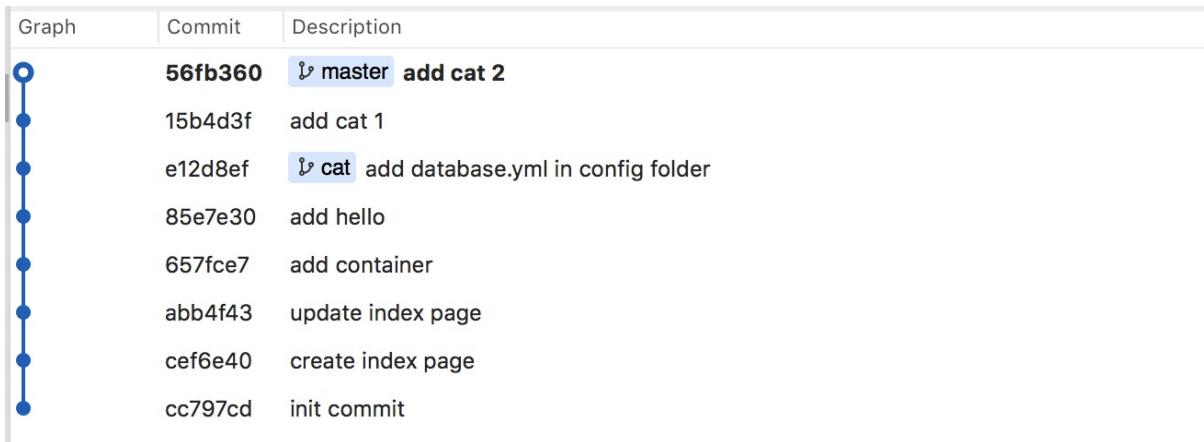
把 `HEAD` 至到 `cat` 分支後，就可以把 `master` 分支移到原本 `cat` 分支的位置：

```
$ git branch -f master e12d8ef
```

因為 Git 的分支就像貼紙一樣，所以透過移動分支貼紙的手法就可以解決這個情況。

方法 C. 直接改分支名字

再次強調，Git 的分支就像貼紙一樣，即然剛才我們可以使用移動分支的手法，當然也可透過修改分支名字的手法來達到目的。繼續使用上面的例子：



要修改分支名字，使用的是 `git branch -m` 指令。不過在改分支名字的時候，如果 `HEAD` 剛好在要修改的那個分支上的話會無法修改，所以我先把 `HEAD` 移到別的地方，例如移到 `56fb360` 上：

```
$ git checkout 56fb360
Note: checking out '56fb360'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 56fb360 add cat 2
```

因為 `HEAD` 移到 `56fb360` 之後，沒有指向任何一個本機的分支，所以呈現了「斷頭 (detached HEAD)」的狀態。放心，這不是錯誤訊息，這只是跟你說現在 `HEAD` 沒有指向任何一個分支而已。更詳細內容可參閱「[【冷知識】斷頭\(detached HEAD\) 是怎麼一回事？](#)」章節介紹。既然 `HEAD` 已經移開了，接下來就可以安心的來修改分支的名字：

```
# 把 cat 先改成 tmp
$ git branch -m cat tmp

# 把 master 改成 cat
$ git branch -m master cat

# 把 tmp 改回 master
$ git branch -m tmp master
```

這概念有點像要把 `x` 跟 `y` 兩個變數的值對調一樣，先把 `cat` 改成 `tmp`，然後把 `master` 改成 `cat`，最後再把 `tmp` 改成 `master`，這樣就達到目的了。最後，再把 `HEAD` 移回 `master` 分支：

```
$ git checkout master  
Switched to branch 'master'
```

搞定！

狀況二：忘了開分支，直接在 Commit 在 master 分支

前面的情況都是有開分支但忘記切換，接下來這個情況是分支都忘了開就直接在 `master` 分支進行了 2 次 Commit，原本 `master` 的位置應該是在 `e12d8ef`，現在的歷史紀錄如下圖：

Graph	Commit	Description
	<code>cfb1946</code>	<code>↳ master add cat 2</code>
	<code>4f1a6a7</code>	<code>add cat 1</code>
	<code>e12d8ef</code>	<code>add database.yml in config folder</code>
	<code>85e7e30</code>	<code>add hello</code>
	<code>657fce7</code>	<code>add container</code>
	<code>abb4f43</code>	<code>update index page</code>
	<code>cef6e40</code>	<code>create index page</code>
	<code>cc797cd</code>	<code>init commit</code>

沒開分支，就補開一個吧：

```
$ git checkout -b cat
```

`git checkout` 指令可用來切換分支，`-b` 參數則是「當分支不存在的時候，會直接建立分支」，這個指令執行後，現在的狀態應該會像這樣：

Graph	Commit	Description
	<code>cfb1946</code>	<code>↳ cat ↳ master add cat 2</code>
	<code>4f1a6a7</code>	<code>add cat 1</code>
	<code>e12d8ef</code>	<code>add database.yml in config folder</code>
	<code>85e7e30</code>	<code>add hello</code>
	<code>657fce7</code>	<code>add container</code>
	<code>abb4f43</code>	<code>update index page</code>
	<code>cef6e40</code>	<code>create index page</code>
	<code>cc797cd</code>	<code>init commit</code>

因為現在 `HEAD` 在 `cat` 分支，所以這時可以再用剛才介紹的「移動分支的位置」的技巧，把 `master` 分支「搬到」它原本的位置（就是 `e12d8ef`）：

```
$ git branch -f master e12d8ef
```

指令執行後的狀態應該會像這樣：

Graph	Commit	Description
	<code>cfb1946</code>	<code>! cat add cat 2</code>
	<code>4f1a6a7</code>	<code>add cat 1</code>
	<code>e12d8ef</code>	<code>! master add database.yml in config folder</code>
	<code>85e7e30</code>	<code>add hello</code>
	<code>657fce7</code>	<code>add container</code>
	<code>abb4f43</code>	<code>update index page</code>
	<code>cef6e40</code>	<code>create index page</code>
	<code>cc797cd</code>	<code>init commit</code>

這樣就搞定了！

小結

分支在 Git 使用的機會非常多，像這種「啊，我還沒開分支就 Commit 下去了！」的狀況是很常見的失誤。遇到這樣的問題，感覺上應該使用 `git reset` 之後再重新 Commit 比較正確，而像移動分支的位置或是修改分支的名字這樣的手法雖然感覺不太正經，但還是可以達到一樣的效果，甚至如果你很清楚自己在做什麼的話，要直接進到 `.git` 目錄修改分支名稱或是內容也都沒問題。

介紹到這裡大家應該不難發現，如果對 Git 的運作原理以及分支基本觀念熟悉的話，只要知道所謂的「開分支」本質上是怎麼一回事，以上這些狀況都有機會補救回來的。基本技巧練熟了，再多的變化都不怕喔！

【狀況題】嘆？這個問題是從什麼時候就有的？

「嘆？為什麼系統的訂單金額一直都是顯示成 0 元？」「從什麼時候開始有這問題的？」

主管看到訂單系統的顯示結果不正常，想問到底是怎麼一回事。但這個專案的開發團隊有二十多人，大家每天都有許多的 Commit，也許某位同事無意間改了計算稅務的功能，結果不小心跟著改到訂單顯示金額了。

如果很明確的知道問題是出在哪一個檔案的話，我們在前面章節「[【狀況題】等等，這行程式誰寫的？](#)」曾經介紹過 `git blame` 指令可以找出指定檔案的第幾行是誰在什麼時候寫的，一下子就可以把犯人抓出來。

但是，有時候一個問題的造成，背後可能是多個程式執行之後的結果，或像是開發 iOS App 需要進行編譯才能看到顯示結果為什麼不正確，所以可能不是單純光用 `git blame` 就能輕鬆找出問題來。更何況，這個問題可能早在幾週前就發生了，只是當時沒人發現，直到現在才曝光。在這麼多的 Commit 裡，要怎麼知道從哪個 Commit 開始是有這個問題的？特別是如果大家在 Commit 訊息都沒有特別註明自己改了什麼內容的話，面對這幾百甚至是幾千個 Commit，難道只能 Checkout 到每一個 Commit，然後檢查訂單的顯示結果是不是正確嗎？

使用「二分法」

不知道大家有沒有玩過「猜數字」遊戲：

「請你從 1 到 100 之間猜一個數字，如果沒猜中，我會跟你說你猜的數字比答案大還是比答案小，然後繼續猜，直到猜到為止」

「50！」

「比 50 大」

「75！」

「比 75 小」

「65！」

「恭喜你，答對了」

通常一開始我們會使用「二分法」從中間的 50 開始猜，為什麼？因為這樣一次就能刪掉一半的數字，沒猜中的話，就再對剩下的另一半再切一半繼續猜，這樣應該沒幾次就能猜到答案了。

在 Git 裡有個叫做 `bisect` 的指令，也是用類似的「猜數字」的方式，可以很快把有問題的點找出來。

檔案下載：<https://ubin.io/git-bisect>

在這個範例裡，目前總共有 23 次 Commit：

```
$ git log --oneline
6e593bc (HEAD -> master) update CSS
e67e3cf add footer
da11948 update CSS
84ee212 add links
8b59e89 add Tiger page
711bc24 update title of Fish page
1ee10be update page
6a8fb89 update title
6e3fab8 update title
b031a2c update page
73860fa update page
f53c5ea add Fish page
afa4537 add cover image
2d15868 remove js code
8f744f8 add Dog page
b51f5e8 add Cat page
da3d4b9 update CSS
7a51f8e add Sign in form
7e64d30 update script
1086842 link style and script file in index
f6b72af add stylesheet and script file
3de629a add HTML content
f9f14ff init commit
```

不知道什麼時候開始，在 `index.html` 頁面上的 Sign In 按鈕沒辦法按了（被 `disabled` 了）。看看上面這些 Commit 訊息，寫得實在很糟糕...根本看不出來 update 了什麼內容。

所以，除了一個一個去看到底每個 Commit 改了什麼之外，我們可以試著使用 Git 裡的 `bisect` 指令來抓問題：

```
$ git bisect start 起始點 結束點
```

因為我確定在 `f6b72af` 的時候功能還是正常的，所以可以這樣做：

```
$ git bisect start HEAD f6b72af
```

意思是針對目前的所在地（也就是 `HEAD`）到 `f6b72af` 這段範圍開始來做二分法。按下 Enter 鍵執行之後：

```
$ git bisect start HEAD f6b72af
Bisecting: 9 revisions left to test after this (roughly 3 steps)
```

```
[73860fa1ba395a7db7dd34c966644f667b0b87e3] update page
```

這時 Git 會直接跳到差不多一半的地方，也就是 `73860fa` 這個 Commit。這時候發現功能還是正常的，Sign In 按鈕是可以按的，所以就可以跟 Git 說目前的結果是好的：

```
$ git bisect good
Bisecting: 4 revisions left to test after this (roughly 2 steps)
[711bc2458ac4c9c87b41110a4f23224ff1536a13] update title of Fish page
```

這時 Git 會自動跳到差不多中間的位置，也就是 `711bc24` 這個 Commit。重新整理頁面，發現這時候功能是壞掉的...所以繼續跟 Git 說現在是壞掉的：

```
$ git bisect bad
Bisecting: 2 revisions left to test after this (roughly 1 step)
[6e3fab83aeb14d9640337357027c167194f0c628] update title
```

現在看起來是正常的，所以：

```
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[1ee10be0c8b8565a8d2e66e1707d90c3ef3fa07a] update page
```

如果結果是正常的，就是 `git bisect good`，如果結果是不正確的，就是 `git bisect bad`，在不斷的 good 跟 bad 的交錯幾次之後，最後找到答案了：

```
$ git bisect bad
6a8fb890400553b95e27996ec948fefb836cb2d6 is the first bad commit
commit 6a8fb890400553b95e27996ec948fefb836cb2d6
Author: Eddie Kao <eddie@digik.com.tw>
Date:   Wed Aug 7 16:18:23 2019 +0800

    update title

:100644 100644 bd266562ec6d1ec64e5c29f32c7190582db95958 39f7c9edc892bd2b2e916e233ad1bc3989
f7a3d4 M      index.html
```

抓到了！看看在第一行出現的訊息，Git 告訴你「`6a8fb89...cb2d6 is the first bad commit`」，就是從這個地方開始壞掉的，原來是某位同事在修改頁面的時候暫時把那個按鈕關掉，結果忘記改回來...

如果 `bisect` 做一半不想做了，或是做完了，可以使用 `reset` 參數：

```
$ git bisect reset
Previous HEAD position was 6a8fb89 update title
Switched to branch 'master'
```

然後就回到原本的 `master` 分支了。

雖然在這個範例只有少少 23 個 Commit，可能感受不出來效果，感覺自己一個一個 Commit 查也是可以查得到，但如果是 230 個或是 2,300 個就不一樣了。生命應該浪費在寶貴的事物上，而不是浪費在這種地方呀 :)

【狀況題】測試還要跑好久，但老闆叫我去修別的分支的問題...

如果開發的時候有寫測試的習慣（Test-Driven Development, TDD），隨著專案成長，測試程式碼（理論上）越來越多，要執行完整的測試可能需要花上半小時、一小時甚至更久。這時可能時不方便切換分支去做別的事情，這段時間好像也只能等，但又不想被長官覺得是在打混摸魚...

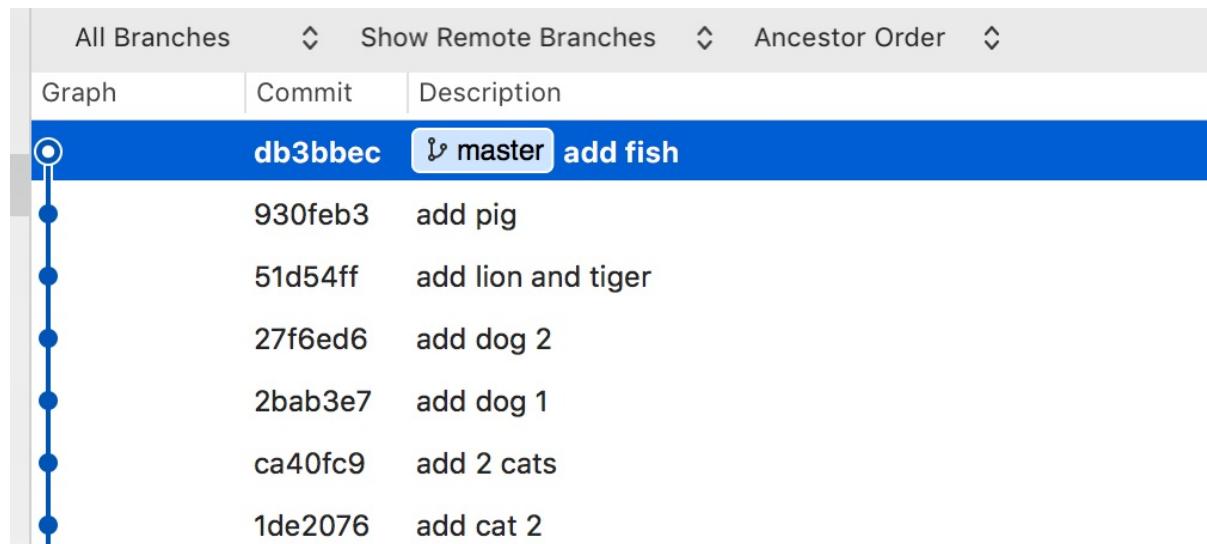
業界常見的做法，通常會另外架設一台 CI（Continuous Integration，持續整合）的機器，你把程式碼推上去或是發個 PR（Pull Request）之後，CI 就會自動幫你執行測試。也就是說，不是在你的本機執行測試，所以自然就不會有上面提到的「不方便在跑測試的時候切換分支」的事情發生。

但，不是每個單位都有架設 CI 機器，而且就算有 CI 也可能需要排隊等其它的進度跑完才行。如果一定要在自己本機跑測試，有些人可能會把原來的專案整個複製（Copy & Paste）一份出來，或是再重新 Clone 一份，然後在這個複製品進行修改，改完再想辦法放回原專案。

這樣感覺有點辛苦，事實上，Git 有個方便的指令可以做這件事...

檔案下載：<https://ubin.io/git-rebase>

目前的專案的歷史紀錄看起來像這樣：

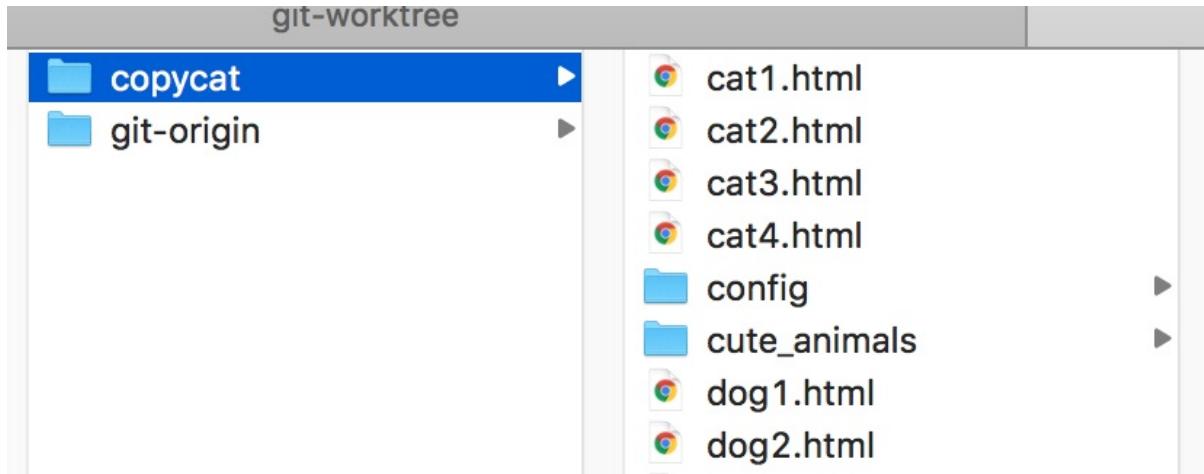


我們常用的 `git branch cat` 指令會建立一個名為 `cat` 的分支，而所謂的「分支」也不過就是一個 40 個 bytes 大小的檔案，其實分支就跟「貼紙」的概念差不多，每個分支都會指向某個 Commit 上面。（關於分支，請參閱「[對分支的誤解](#)」章節介紹）。建立一個分支也只有這樣而已，建立分支的過程並沒有實際複製任何專案裡的檔案或目錄到別的地方。也許在別的版控軟體的確是會複製一份檔案或目錄，但 Git 不是這樣運作的。

而接下來要介紹的這個 `git worktree` 指令除了建立分支之外，還真的會把檔案或目錄複製一份到別的目錄：

```
$ git worktree add -b cat ../copycat
Preparing worktree (new branch 'cat')
HEAD is now at db3bbec add fish
```

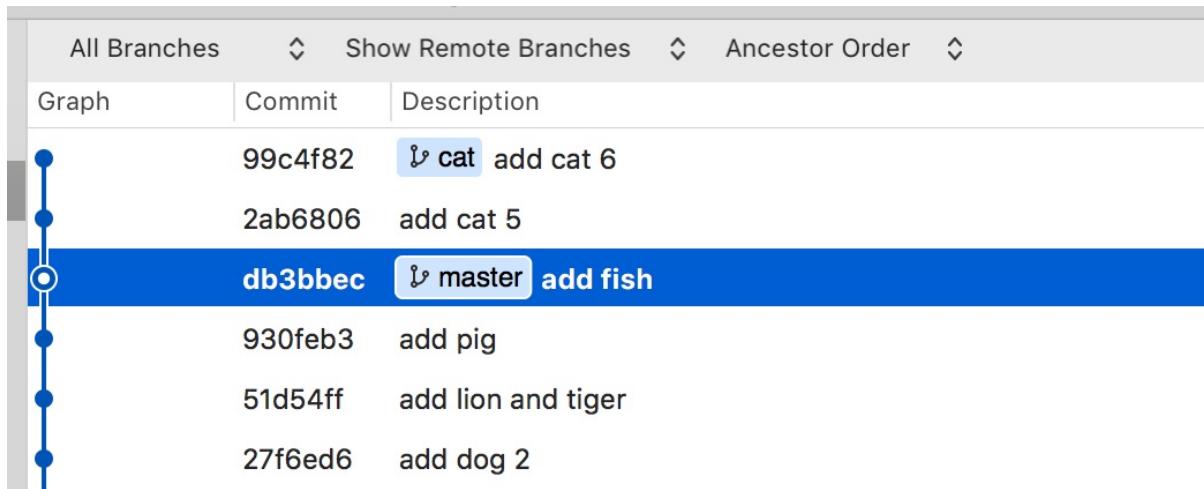
這個指令會建立一個名為 `cat` 分支（那個 `-b` 參數就是建立分支的意思），並同時在原專案的上一層目錄建立一個叫做 `copycat` 的目錄：



這時，你只要進到剛剛複製的 `copycat` 目錄，就等於進到 `cat` 分支了：

```
$ cd ../copycat
$ git branch
* cat
  master
```

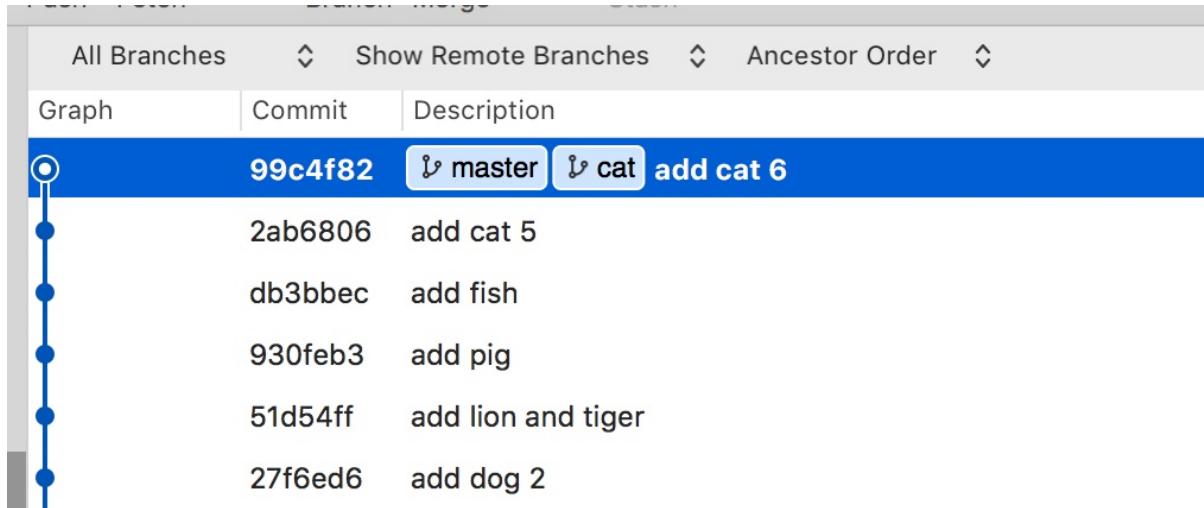
接下來的操作，就跟一般的操作沒什麼太大的差別，同樣可以進行像是 `add`、`commit` 或是 `log` 指令，差別只在於這個 `cat` 分支「不是在原本的目錄」而已。我在 `cat` 分支進行了兩次 Commit 之後，雖然這個 `cat` 分支是在別的目錄，但這兩次 Commit 在原來的專案也還是會被看到：



所以，用起來就真的跟一般的分支沒兩樣。而在 `cat` 分支的任務完成後，可回到 `master` 分支進行合併：

```
$ cd ../git-origin
$ git merge cat
Updating db3bbec..99c4f82
Fast-forward
 cat5.html | 0
 cat6.html | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cat5.html
 create mode 100644 cat6.html
```

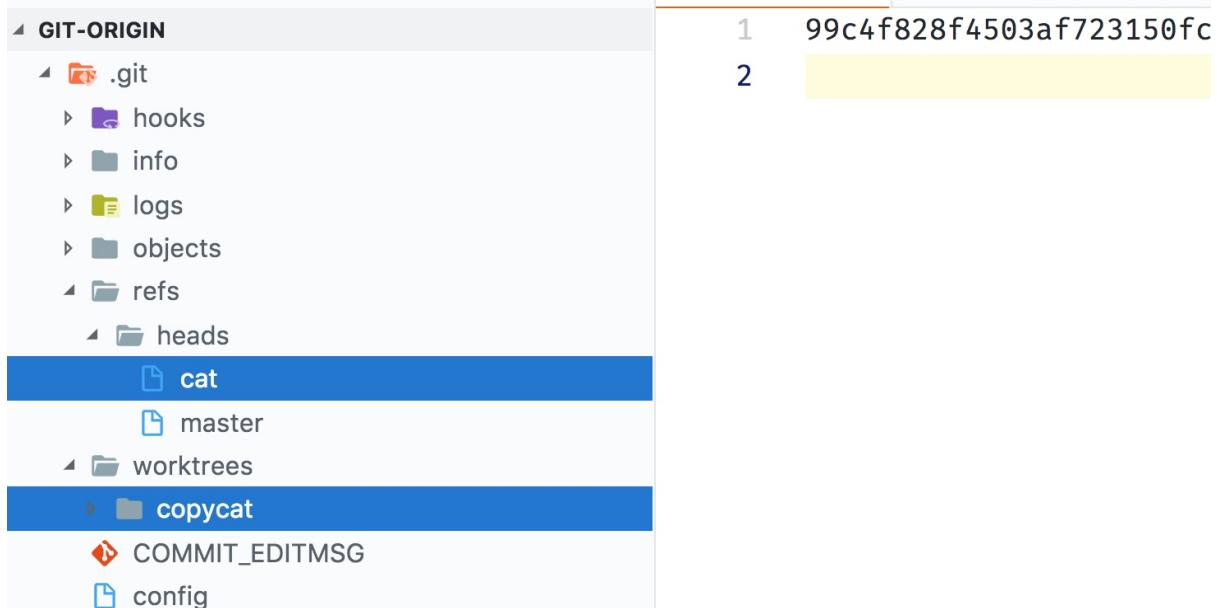
合併完成後，歷史紀錄變這樣：



看到了嗎！這用起來就跟一般的分支沒什麼兩樣。至於剛剛複製的 `copycat` 目錄在合併之後要刪除或是要留著就看你的心情了。基本上分支合併後，更新的檔案在原專案也有一份了，所以 `copycat` 目錄可以放心的刪除。

怎麼做到的？

讓我們來看看 Git 是怎麼做到這件事的。一般的 `git branch` 指令，會在 `.git/refs/heads` 目錄下新增一個檔案，但執行 `git worktree` 的時候，除了在 `.git/refs/heads` 目錄裡新增了 `cat` 檔案之外，在 `.git` 目錄裡還會多一個 `worktrees` 目錄，裡面還有我們剛剛新增的 `copycat` 目錄：



在 `worktrees/copycat` 目錄裡，有個 `gitdir` 檔案，它的內容是：

```
/tmp/copycat/.git
```

你可以把 Worktree 當做是一般的分支看待，只是 Worktree 指令其實是複製一份檔案出來（這可能跟很多人認知的錯誤分支觀念是一樣的），並且在複製出來的專案裡加上一個 `.git` 檔案。注意，是檔案，不像一般的 Git 專案是一個 `.git` 目錄。如果你進到 `copycat` 目錄看看這個 `.git` 檔案的話：

```
$ cat .git  
gitdir: /tmp/git-origin/.git/worktrees/copycat
```

你會發現這個檔案其實就只有一行，內容指向原專案的 `.git/worktrees/copycat` 目錄。根據 Git 官方文件上面寫著：

Note: Also you can have a plain text file `.git` at the root of your working tree, containing `gitdir: <path>` to point at the real directory that has the repository. This mechanism is often used for a working tree of a submodule checkout, to allow you in the containing superproject to git checkout a branch that does not have the submodule. The checkout has to remove the entire submodule working tree, without losing the submodule repository.

這段說明的意思是說，如果你在專案裡放一個 `.git` 的「檔案」（注意，不是目錄喔！），檔案裡的內容像上面這樣註明正確的 `gitdir: <路徑>`，即使沒有 `git init` 所生成的 `.git` 目錄結構，也可以被當做一個有 Git 版控的目錄來看待。

為什麼只靠一個 `.git` 檔案就能做到一般 Git 專案的 `.git` 目錄結構，說穿了就是把原本應該有的 `.git` 目錄結構放在原來的專案的 `.git/worktrees` 裡，然後透過這個 `.git` 檔案指向原本專案的 `.git/worktrees` 目錄而達到版本控制的效果（感覺很像繞口令）。

不過要注意的是，`git worktree` 指令是在 Git 2.6.0 之後的版本才有的，所以如果你的 Git 版本太老舊的話就沒辦法使用這個功能了。

【冷知識】手工初始化 .git 目錄

在「[新增、初始 Repository](#)」章節曾經介紹過，要開始使用 Git，第一件要做的事就是初始化：

```
$ git init
```

這個指令，會幫你建立一個 `.git` 目錄，剩下的，就是各位的事了。`git init` 這個指令幫你在 `.git` 目錄裡建立了許多的檔案及目錄，但其實要讓 Git 可以進行版控，並不需要這麼多東西。

根據 Git 的[官方文件說明](#)，`git init` 指令會做的事：

This command creates an empty Git repository - basically a `.git` directory with subdirectories for objects, refs/heads, refs/tags, and template files. An initial HEAD file that references the HEAD of the master branch is also created.

既然知道初始化大概在做些什麼事，接下來就讓我們用純手工的方式來建立這個 `.git` 目錄吧！（本章節用到的系統指令，可參閱「[終端機及常用指令介紹](#)」章節介紹）

純手工打造

先讓我來建一個叫做 `super-fake-project` 專案：

```
$ mkdir super-fake-project
```

這個專案就是我們的實驗對象，目前裡面什麼都沒有。別忘了先使用 `cd` 指令進到這個目錄，然後這次我不打算使用 `git init` 指令，而是手工新增一個 `.git` 目錄：

```
$ cd super-fake-project
$ mkdir .git
```

不過，並不是有了 `.git` 目錄就等於可以讓 Git 版控，例如這時候執行這個指令：

```
$ git status
fatal: not a git repository (or any of the parent directories): .git
```

系統訊息告訴你，目前這個地方並不是一個有效的 Git Repository。根據文件說明，在 `.git` 目錄裡面應該要有地方可以存放「物件」。即然需要，那就做給它：

```
$ mkdir .git/objects
```

Git 世界的「物件」就是存放在 `.git/objects` 這個目錄裡。關於 Git 世界的「物件」的更多細節，請參閱「[【超冷知識】在 .git 目錄裡有什麼東西？](#)」的 Part 1 以及 Part 2 章節介紹。

除了有可以放物件的地方之外，還需要可以放分支（Branch）的地方。在 Git 的世界，分支是存放在 `.git/refs/heads` 這個目錄裡，所以繼續再手動做給它：

```
$ mkdir .git/refs  
$ mkdir .git/refs/heads
```

上面這兩個動作，其實可以濃縮成一個：

```
$ mkdir -p .git/refs/heads
```

如果要建立像是 `/aaa/bbb/ccc` 這樣的目錄結構，通常必須依序建立 `aaa` 及 `bbb` 目錄之後，才能建立 `ccc` 目錄。如果在 `mkdir` 指令後面加上 `-p` 參數，可以一口氣幫你做出中間還沒有建立的目錄。

最後，根據文件，還需要再做一個 `HEAD` 檔案，讓 Git 知道現在在什麼地方：

```
$ echo "ref: refs/heads/master" > .git/HEAD
```

`echo` 指令會把接在後面的內容印在畫面上，而後面接的 `>` 符號，是指要把這些內容輸出成一個叫做 `HEAD` 的檔案，並放在 `.git` 目錄下。至於這個 `HEAD` 是做什麼的，可參閱「[【冷知識】HEAD 是什麼東西？](#)」章節說明。

到這裡，Git 應該就可以用了：

```
$ git status  
On branch master  
  
No commits yet  
  
nothing to commit (create/copy files and use "git add" to track)
```

好像可以...那就來試用看看：

```
$ touch index.html  
$ git add index.html  
$ git commit -m "init commit"  
[master (root-commit) 684756b] init commit  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 index.html
```

看起來可正常使用沒問題！

簡單的說，即使沒有完整的 `.git` 目錄結構，只要有這三項東西，就能開始使用 Git 進行版控了：

1. `.git/objects` 目錄，用來裝物件的。
2. `.git/refs/heads` 目錄，用來裝分支的。
3. `.git/HEAD` 檔案，內容寫著 `ref: refs/heads/master`，用來告訴 Git 現在的位置。

其它像是在 `.git/objects` 裡以及在 `.git/refs` 裡的其它檔案或目錄，在開始使用 Git 的時候就會自動生出來了。

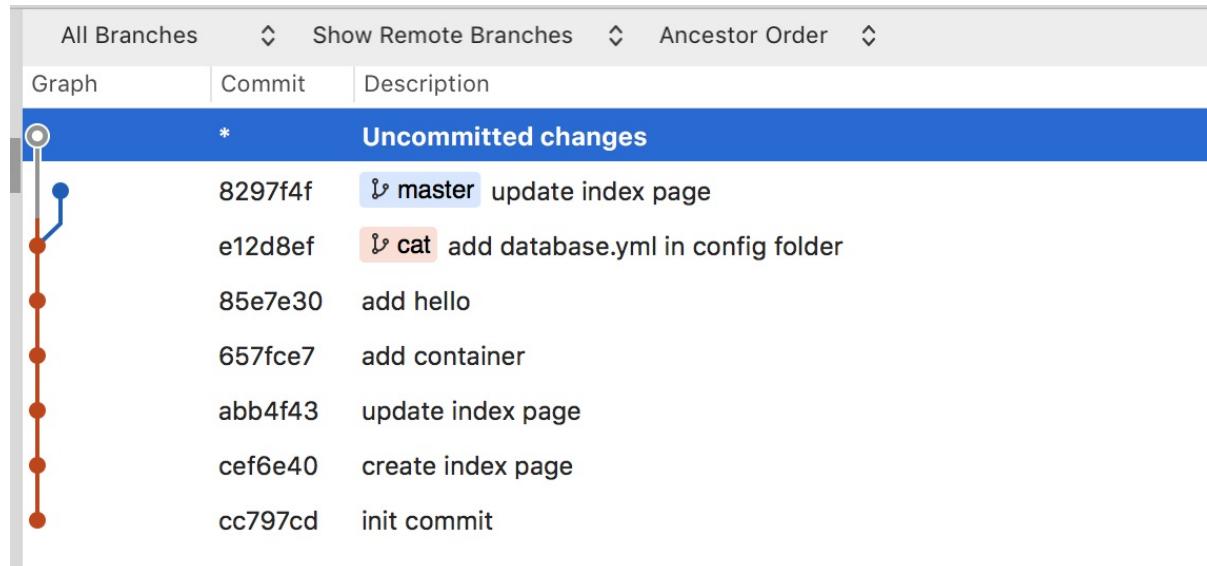
是說，手工初始化 Git 有什麼好處嗎？沒有！這個章節的目的只是讓你更了解一些關於 Git 的原理，但實務上我還沒看過人會放著 `git init` 這麼簡單的指令不用，然後純手工打造 `.git` 目錄的。

【冷知識】Stash 是什麼？

在「[【狀況題】手邊的工作做到一半，臨時要切換到別的任務](#)」章節曾經介紹過 `git stash` 這個指令，可以暫時把工作「壓」在某個地方，等需要的時候再把它撿回來繼續做。那，Stash 到底是什麼東西呢？

檔案下載：<https://ubin.io/git-stash>

我們先來看看這個專案在 SourceTree 裡的樣子：



在這個專案中，現在正在 `cat` 分支進行修改，有一個修改一半的 `index.html`，再使用 `git status` 指令確認一下現在的狀態：

```
$ git status
On branch cat
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

接著執行 Stash 指令：

```
$ git stash
Saved working directory and index state WIP on cat: e12d8ef add database.yml in config fol
der
```

這樣就把目前的進度「壓」起來了。但，到底壓到哪去了？再翻一下目前的 Stash 列表：

```
$ git stash list
stash@{0}: WIP on cat: e12d8ef add database.yml in config folder
```

好，目前的確只有一個 Stash，但這個 `stash@{0}` 的內容到底是什麼，讓我們用 `git cat-file` 指令來看看這傢伙到底是什麼：

```
$ git cat-file -t stash@{0}
commit
```

咦？這個 Stash 原來是一個 Commit 耶！再看看裡面是什麼：

```
$ git cat-file -p stash@{0}
tree 4b4bb17f62a942bdef6d060005123192635663dd
parent e12d8ef0e8b9deae8bf115c5ce51dbc2e09c8904
parent 59c60d82440dc21d7eec0b085ebfad51e8eb5936
author Eddie Kao <eddie@digik.com.tw> 1560838887 +0800
committer Eddie Kao <eddie@digik.com.tw> 1560838887 +0800

WIP on cat: e12d8ef add database.yml in config folder
```

嗯，這個 Stash 看起來跟一般的 Commit 很像，也是有 `tree` 跟 `parent` 之類的結構（詳情請見「[【超冷知識】在 .git 目錄裡有什麼東西？Part 1](#)」章節介紹）。但，它怎麼會有 2 個 Parent Commit？其中那個 `e12d8ef` 看起來是在進行 Stash 之前的 Commit，那另一個呢？繼續再往下挖看看：

```
$ git cat-file -p 6d1b7aa9
tree 3320b5871aa32d76f6d3a125f43ce27bca629502
parent e12d8ef0e8b9deae8bf115c5ce51dbc2e09c8904
author Eddie Kao <eddie@digik.com.tw> 1560645669 +0800
committer Eddie Kao <eddie@digik.com.tw> 1560645669 +0800

index on cat: e12d8ef add database.yml in config folder
```

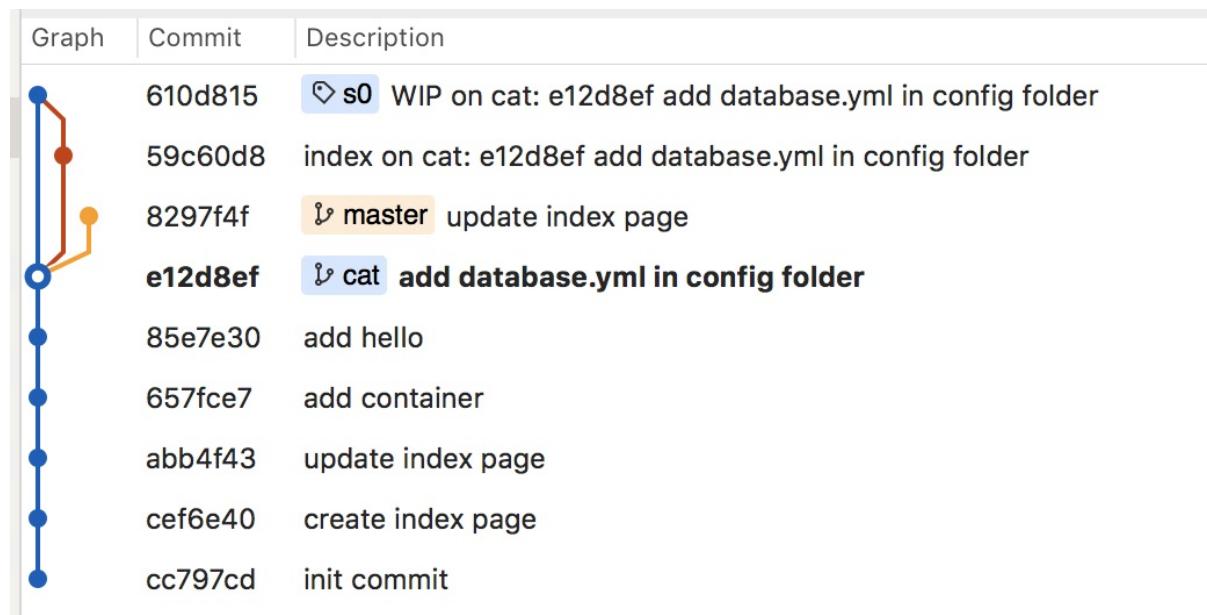
這個神秘的 Commit 的 Parent 也是 `e12d8ef`。既然 Stash 本身也是一個 Commit，所以我們可以用 `git log` 指令來看看它的歷史：

```
$ git log --oneline --graph stash@{0}
*   610d815 (refs/stash) WIP on cat: e12d8ef add database.yml in config folder
|\ 
| * 59c60d8 index on cat: e12d8ef add database.yml in config folder
|/
* e12d8ef (HEAD -> cat) add database.yml in config folder
* 85e7e30 add hello
* 657fce7 add container
* abb4f43 update index page
* cef6e40 create index page
* cc797cd init commit
```

雖然我們可以透過 `git log` 指令看到 Stash 的所在地，但為了能讓這個 Stash 被更清楚看到，我直接在 `stash@{0}` 打上一個名為 `s0` 的標籤（tag）：

```
$ git tag s0 stash@{0}
```

這樣就可以讓這個 Stash 也顯示出來了：



所以，到底在產生 Stash 的時候發生了什麼事？

其實，在執行 `git stash` 的時候，根據情境不同，會產生 2 個或 3 個 Commit。以上面的例子來說，`610d815` 這個 Commit 的內容就是 Stash 的修改內容，也就是 `index.html` 這個檔案的修改內容。而 `610d815` 的前一個 Commit `59c60d8`，其實就是在執行 Stash 指令的時候，暫存區（Staging Area）裡的東西，以我們剛剛這個例子來說，`index.html` 只是修改完並沒有加到暫存區，所以暫存區在這個時候是空的，因此在這個 Commit 裡並沒有任何內容。

那什麼時候會有 3 個 Commit？如果在執行 Stash 指令的時候也把 untracked 狀態的檔案一併壓進 Stash 的話，就會產生第 3 顆 Commit 出來。讓我們重新解壓縮範例，回到最一開始的狀態，然後再做以下三件事：

1. 新增一個名為 `somebody.html` 的檔案，檔案內容不拘。
2. 修改 `hello.html` 的檔案，修改內容不拘。
3. 執行 `git add index.html` 指令，把 `index.html` 加入暫存區。

這時候的狀態會變成這樣：

```
$ git status
On branch cat
```

```
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
    modified:   index.html  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git checkout -- <file>..." to discard changes in working directory)  
  
    modified:   hello.html  
  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  
    somebody.html
```

整理一下現在的狀態：

1. 修改的 `index.html` 已被加至暫存區（Staging Area）。
2. 修改的 `hello.html` 還在工作目錄（Working Directory）。
3. 新增的 `somebody.html` 也在工作目錄，但它是 Untracked 狀態。

如果對暫存區或工作目錄不熟悉的話，可參閱「[工作區、暫存區與儲存庫](#)」章節介紹。接著要執行 Stash 指令的時候，要特別把 Untracked 的檔案也加進來的話，需要額外加上 `-u` 或是 `-all` 參數：

```
$ git stash -u  
Saved working directory and index state WIP on cat: e12d8ef add database.yml in config fol  
der
```

同樣為了方便觀察，我在 `stash@{0}` 貼上一個名為 `s1` 的標籤：

```
$ git tag s1 stash@{0}
```

這時候的狀態變成這樣：

Graph	Commit	Description
	e7fee36	⌚ s1 WIP on cat: e12d8ef add database.yml in config folder
	88ddfdf	untracked files on cat: e12d8ef add database.yml in config folder
	43a8b9e	index on cat: e12d8ef add database.yml in config folder
	8297f4f	↵ master update index page
	e12d8ef	↵ cat add database.yml in config folder
	85e7e30	add hello
	657fce7	add container
	abb4f43	update index page
	cef6e40	create index page
	cc797cd	init commit

形狀變得更有趣了！我們一個一個來看看：

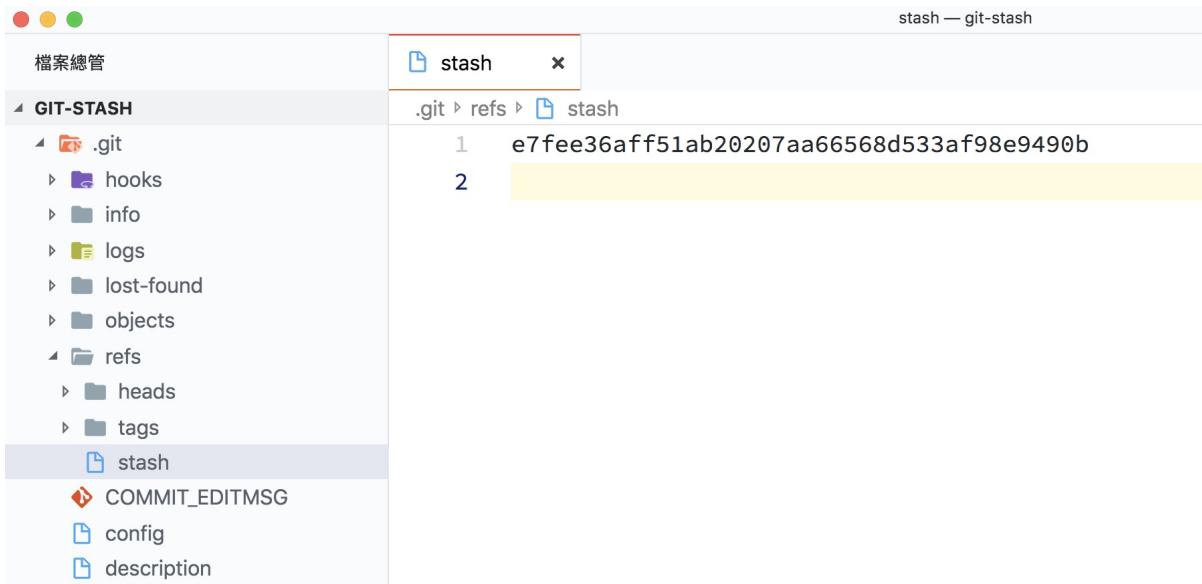
1. 貼著 `s1` 標籤的 `e7fee36` 是 Stash 本體，也是 `index.html` 跟 `hello.html` 的修改內容。
2. `e7fee36` 這個 Commit 的其中一個 Parent Commit `43a8b9e` 如同前面介紹的，就是暫存區的內容，因為我們只有把 `index.html` 加入暫存區，所以這個 Commit 的內容就是 `index.html` 的修改內容。
3. 而 `e7fee36` 的另一個 Parent Commit `88ddfdf`，就是 Untracked 但被一起被壓進來的內容，也就是 `somebody.html` 這個檔案。

基本上你可以把 Stash 看成跟 Commit 是差不多的東西，它的某個 Parent Commit 也會接在原來的 Commit 上，只是除了這個之外，Stash 指令還會分別把暫存區以及 Untracked 的內容另外開兩個 Commit 來放，所以執行 `git stash` 指令時，會根據情境不同，而可能產生二個或三個 Commit。

Stash 存在哪裡？

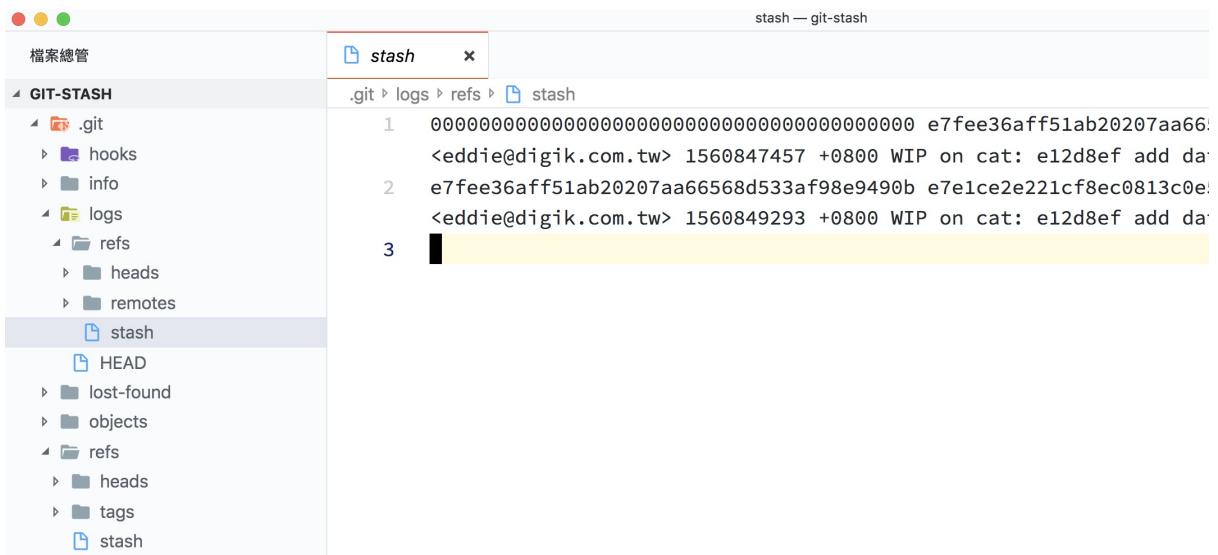
既然知道其實 Stash 只是一個 Commit，表示這些相關的物件（Commit 物件、Tree 物件、Blob 物件）都是存放在 `.git/objects` 目錄裡，規則就跟一般的 Commit 物件一樣。但 Stash 之間並不像一般的 Commit 是一個接一個，所以當如果有多個 Stash 的話，表示應該有個地方存放這些 Stash 的列表。

就跟一般的分支（branch）或是標籤（tag）一樣，Stash 也有在 `.git/refs/stash` 這個檔案裡留下紀錄：



在這個檔案裡的確紀錄了 `stash@{0}`，也就是最後一個疊上去的 Stash 的 SHA-1 值，如果再疊新的 Stash 上去，這個檔案的內容會被換成最後一個 Stash 的 SHA-1 值。那 Stash 列表呢？

原來它藏在 `.git/logs/refs/stash` 裡：



從 Stash 長一個分支出來？

Stash 本質上其實就是 Commit，所以如果有需要的話，也是可以從 Stash 再開一個分支出來，例如在上面的例子中，我想要從 `s1` 標籤的 `e7fee36` 位置來開一個新的兔子分支（rabbit），可以這麼做：

```
$ git branch rabbit e7fee36
```

但如果有這個需求的話，`git stash` 的指令集裡本身就有提供這個功能：

```
$ git stash branch rabbit
```

這個指令就會從最近的 Stash，也就是 `stash@{0}` 來建立一個名為 `rabbit` 的分支，然後同時會把 `stash@{0}` 從 Stash 列表中移除。如果想從其它 Stash 開分支的話，就在後面再加上指定的 Stash ID，像這樣：

```
$ git stash branch rabbit stash@{2}
```

這樣就會從第 3 個 Stash（因為 Stash ID 是從 0 開始算的）建立分支，並且從 Stash 列表上移除它。

git stash pop 之後...

在「[【狀況題】手邊的工作做到一半，臨時要切換到別的任務](#)」章節也曾介紹到，如果要把 Stash 的進度撿回來繼續做，可以使用 `git stash pop` 跟 `git stash apply` 這兩個指令，而 `apply` 跟 `pop` 指令的差別，就是 `pop` 在收回進度的時候會順便把 Stash 清掉。但我們也一直跟大家強調，Git 本身並沒有刪除 Commit 的指令，而 Stash 本身就是 Commit，所以即使把 Stash 給 `pop` 出來，只是把它從 Stash 列表中移除，但並不會讓那個 Stash（或是 Commit）馬上消失，這個 Commit 就是靜靜的等著之後被資源回收（Garbage Collection）給收掉。

了解 Stash 其實就是一種 Commit 之後，下次在使用 `git stash` 指令把進度存起來的時候，應該就會更清楚這個指令在做什麼事了。

【冷知識】～跟 ^ 有什麼不同？

在「[【狀況題】剛才的 Commit 後悔了，想要拆掉重做…](#)」章節曾經介紹過關於 `git reset` 這個指令，假設現在專案的 Git 狀態是這樣：



如果執行這個指令：

```
$ git reset HEAD^ --hard
```

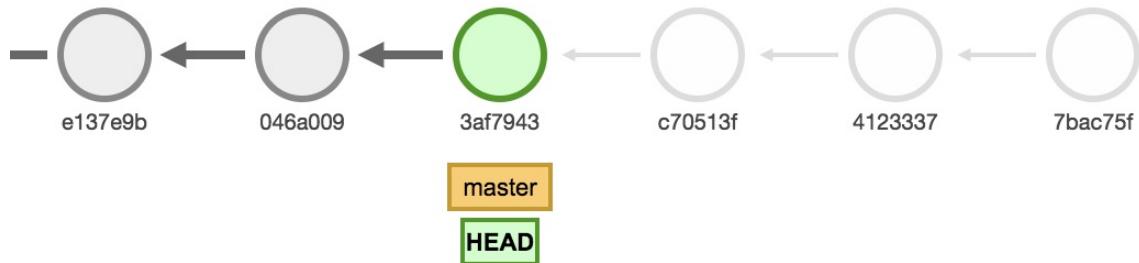
這個 `^` 符號，英文是「Caret」，但在中文尚無通用名稱，可以叫它「插入符號」。如果把這個符號接在 `HEAD` 後面，是指 `HEAD` 的「前一個位置」的意思。所以當這個指令執行之後，就會往前退一個 Commit，看起來像這樣：



每一個 `^` 就是往前一步的意思。如果有多個 `^`，像這樣：

```
$ git reset HEAD^^^
```

就是往前 3 步的意思，執行之後的樣子會變這樣：



當然，如果你覺得連續寫 3 個 ^ 很醜或覺得麻煩的話，你可以改用 ~ 符號：

```
$ git reset HEAD~3
```

這個 ~ 符號，英文是「Tilde」，你可稱它「波浪符號」，這個 ~3 跟 ^3 會指到同一個位置。這兩個符號，看起來好像可以互相替換，但其實還是有些差別的...

所以，是哪裡不一樣？

如果只是上面的例子，基本上是可以互換的沒錯，但我們先看一下這個指令：

```
$ git reset HEAD^
```

其實，它後面省略了數字 1：

```
$ git reset HEAD^1
```

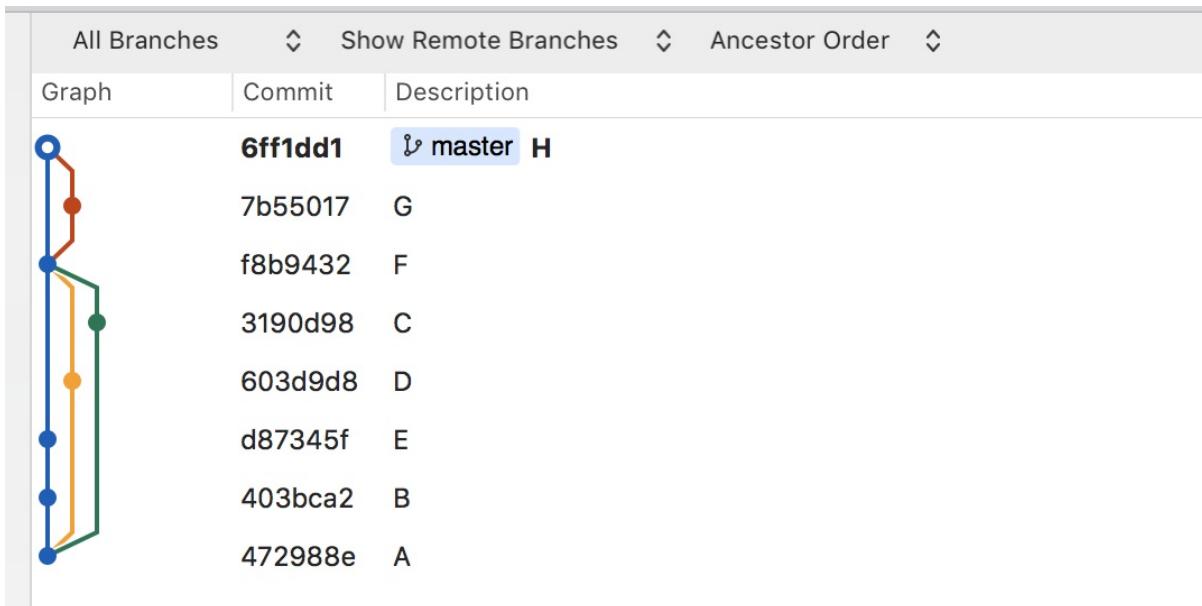
這樣意思是一樣的，但如果是 1 的話，可以省略不寫。那如果是兩個呢？

```
$ git reset HEAD^^
```

因為 1 可以省略不寫，所以上面這個指令可以改成這樣：

```
$ git reset HEAD^1^
$ git reset HEAD^^1
$ git reset HEAD^1^1
```

上面這三個指令的意思都是一樣的，不管是省略第一個 ^ 的 1 還是第二個 ^ 的 1，或是都不省略，都是往前兩個 Commit 的意思。但，如果遇到分支合併，這個 ^ 後面接的數字就會有點差別了。假設我們有個專案的分支像這個樣子：

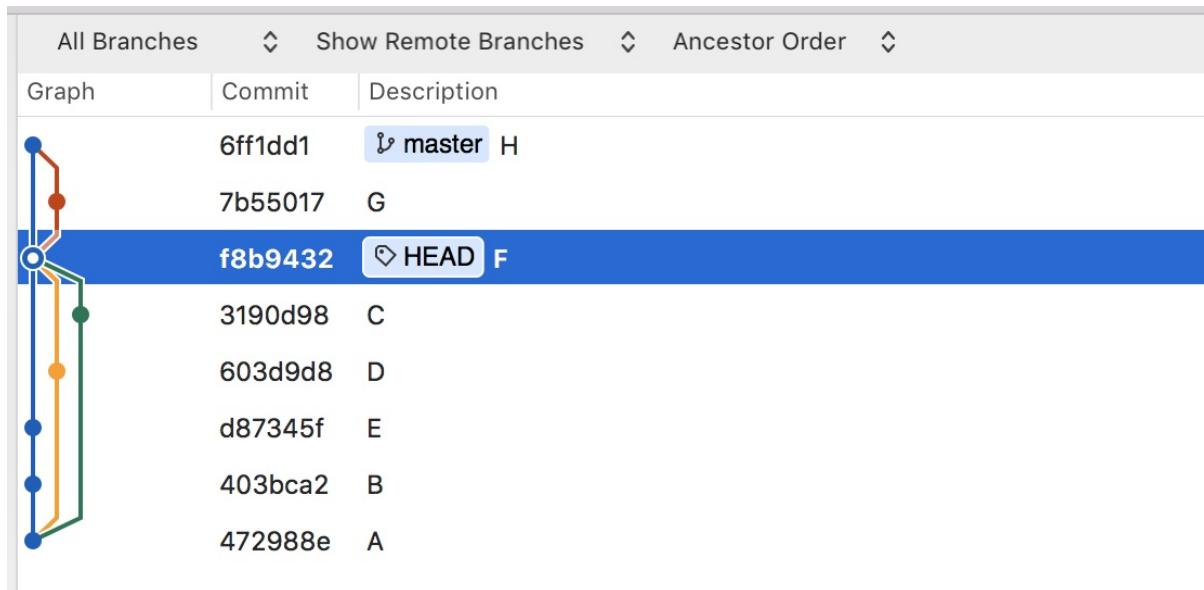


`git checkout HEAD^` 這個指令會把 `HEAD` 往前移一個 Commit。但是，等等...現在的 `HEAD` 是在 `H` (`6ff1dd1`) 的位置，這個指令執行後，`HEAD` 的位置是會跑到 `G` (`7b55017`) ? 還是那個合併節點 `F` (`f8b9432`) 呢？讓我們先用 `git log` 指令看一下目前 `HEAD` 所在的那個 Commit :

```
$ git log
commit 6ff1dd181f7b4bf257a049d98b83600b880b53b7 (HEAD -> master)
Merge: f8b9432 7b55017
Author: Eddie Kao <eddie@digik.com.tw>
Date:   Tue Feb 5 01:30:47 2019 +0800
```

H

因為 `H` 是合併後所產生的 Commit，所以它會有兩個 Parent Commit，分別是 `f8b9432` 跟 `7b55017`，而這兩個 Commit 的順序，決定了當你在「往前移動」的時候會走哪條路。如果下的指令是 `git checkout HEAD^1` 的話，Git 會選擇往第一個 Parent Commit 方向移動，也就是 `F` :

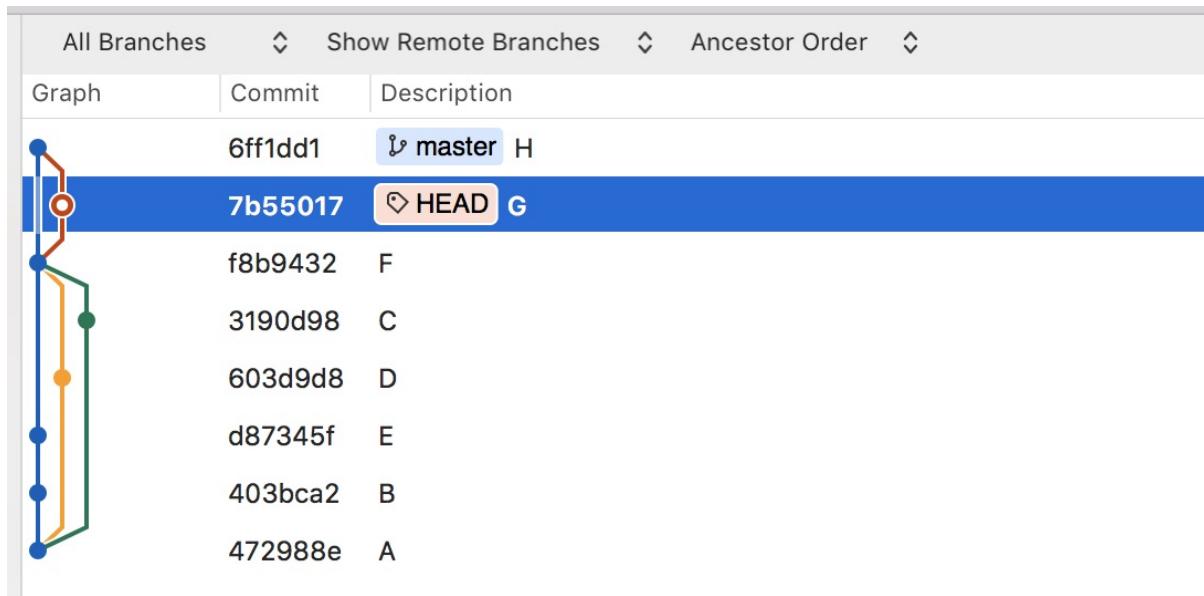


前面也提到，這個 `^1` 的 `1` 可以省略，所以當我們在執行 `git checkout HEAD^` 的時候，`HEAD` 會往回退一個 Commit，如果遇到分叉路的話便會往「第一個 Parent Commit」移動。

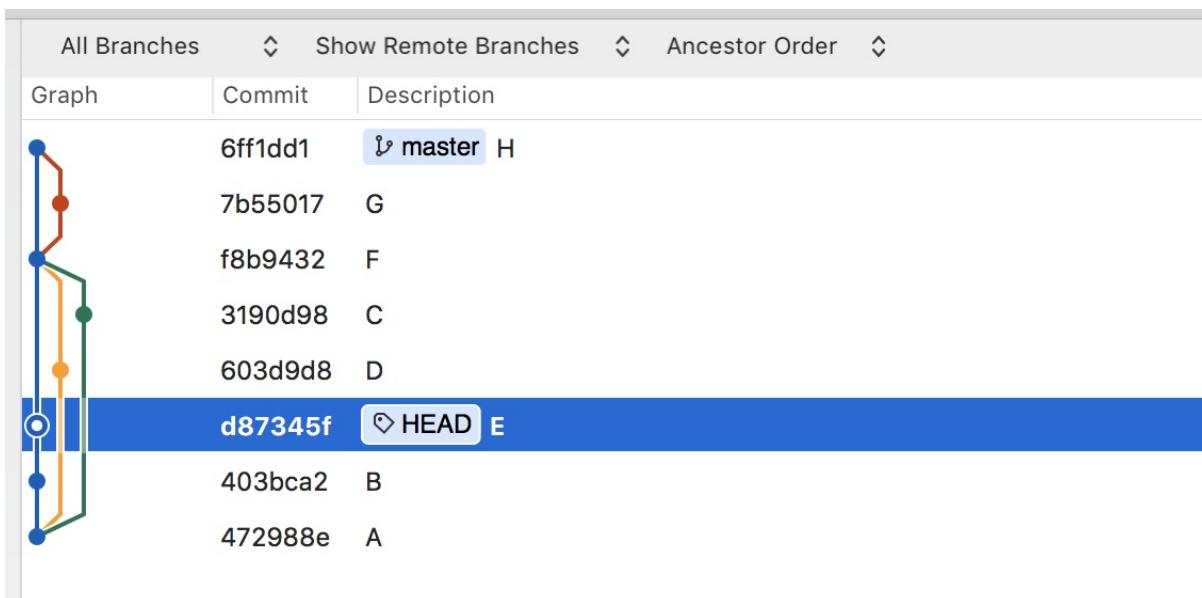
那要怎麼往另一條分支路移動？只要把 `^1` 改成 `^2`：

```
$ git checkout HEAD^2
```

這樣就會往另一個條路走了：

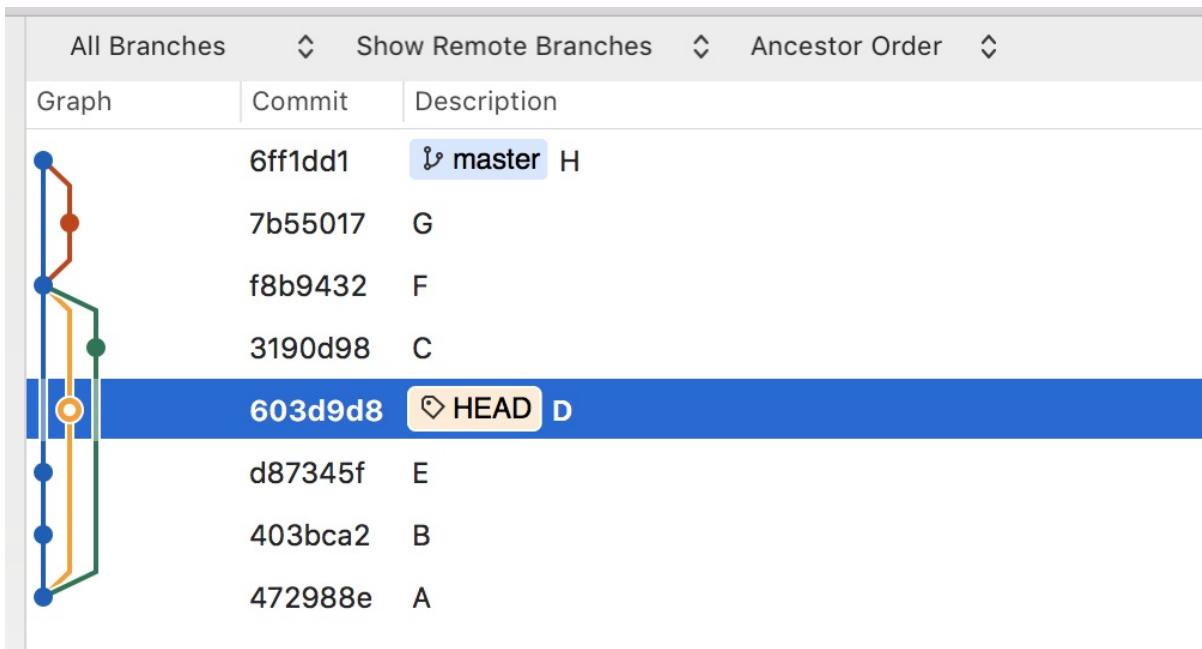


讓我們再往下看。如果從原本的 `H` (`6ff1dd1`) 執行 `git checkout HEAD^^` 指令的話，根據前面的規則，首先，它會先順著第一個 Parent Commit 退到 `F` (`f8b9432`)，再來遇到三叉路，會順著第一個 Parent Commit 退，所以它會退到 `E` (`d87345f`) 這個 Commit：



如果想把 HEAD 從 H 退到 D 的話：

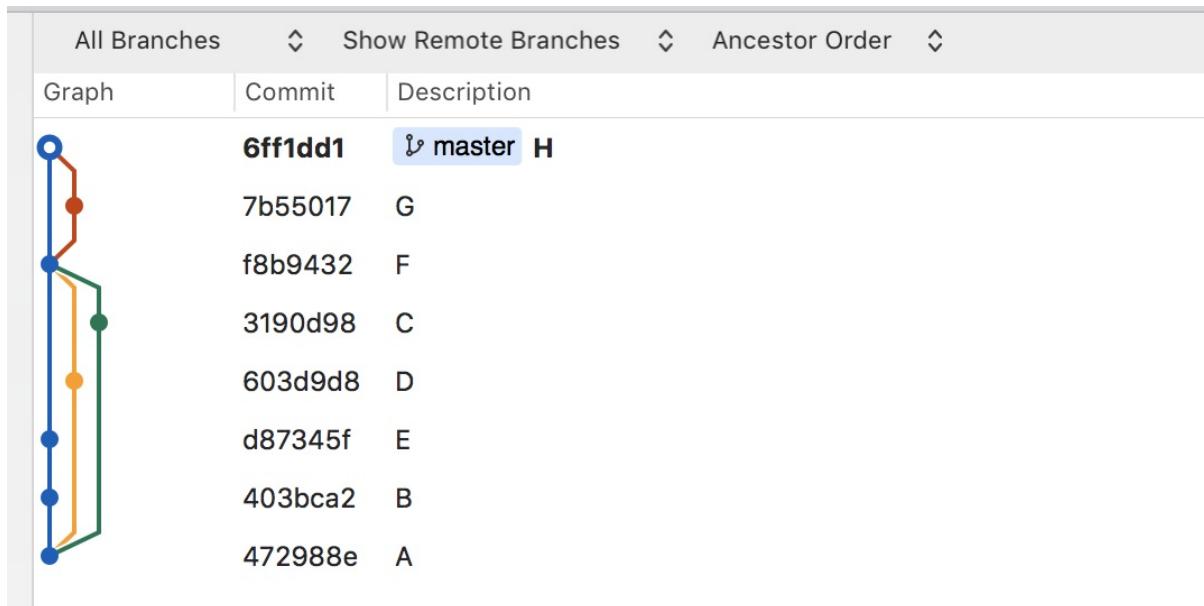
```
$ git checkout HEAD^^2
```



猜猜看，如果想要退到 c (3190d98) 的話該怎麼做？既然 1 是往第一個 Parent Commit 的方向移動， 2 是往第二個 Parent Commit，那要往第三個的話：

```
$ git checkout HEAD^^3
```

這樣就搞定了。最後我們做個小整理：



- $F = H^1 = H^\sim = H^{-1}$
- $G = H^2$
- $E = H^{1^1} = H^{1\wedge} = H^{\wedge 1} = H^{\wedge\wedge} = H^{\sim 2}$
- $B = H^{1^11^1} = H^{\wedge\wedge\wedge} = H^{1\sim 2} = H^{\sim 2} = H^{\sim 2}$
- $D = H^{1^2} = H^{\wedge\wedge 2}$
- $C = H^{1^3} = H^{\wedge\wedge 3}$

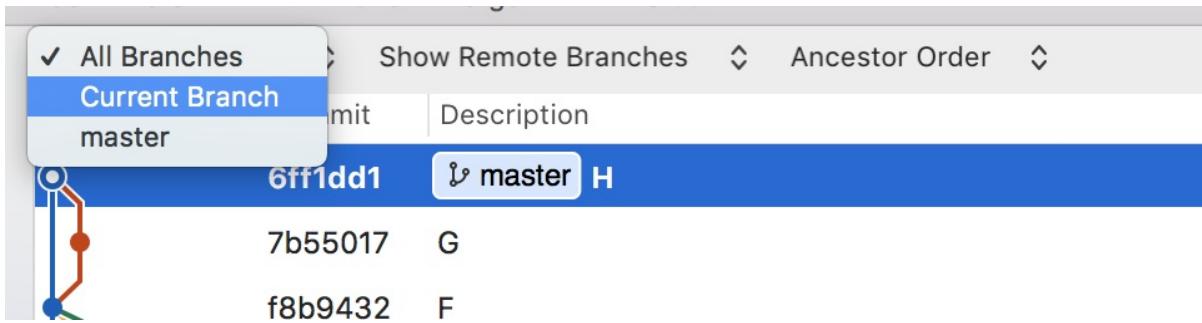
到這裡，你有發現 \wedge 跟 \sim 這兩個符號微妙的不同了嗎？ $\wedge 2$ 會往第二個 Parent Commit 移動，而 ~ 2 沒這個功能，它就是等於 $\wedge 1 \wedge 1$ 也就是 $\wedge\wedge$ 的意思。

只看第一個 Parent Commit 就好？

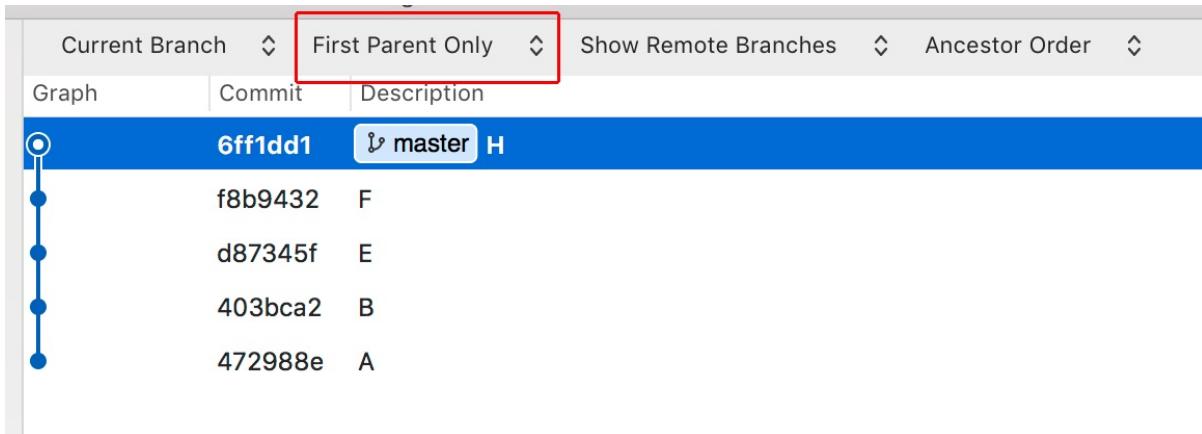
分支一多，畫面看起來會有點亂，這時候可以在 `git log` 後面加上 `--first-parent` 參數，顧名思義，就是只看第一個 Parent Commit 的那條路線的紀錄：

```
$ git log --oneline --first-parent
6ff1dd1 H
f8b9432 F
d87345f E
403bca2 B
472988e A
```

看起來清爽多了。如果是在 Sourcetree 的話，可在介面上先選擇「Current Branch」：



然後就可以再切換到「First Parent Only」模式了：



是說，知道 ^ 跟 ~ 的差異可以做什麼？好像沒什麼用，至少我工作到現在，沒有真的用過 ^2 這樣的用法（因為只要知道 Commit 的 SHA1 值就直接飛過去了，誰還在那邊一步一步慢慢退啊！），不過知道這個冷知識也是不錯啦。