Show

第 22 天：修正 commit 過的版本歷史紀錄 Part 4 (Rebase)

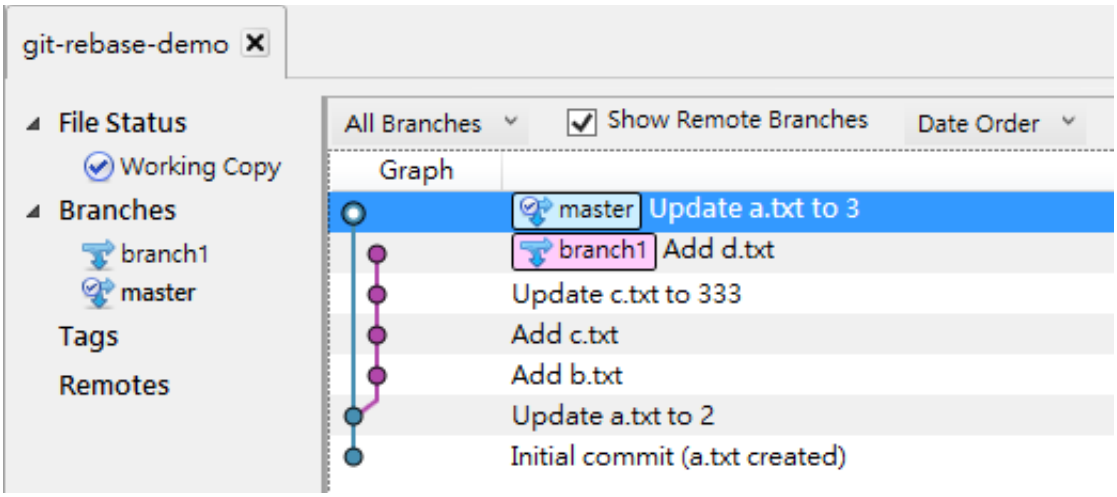
我們之前已經講了三種不同的修正版本的方法，嚴格上來說 `git revert` 與 `git cherry-pick` 並不算「修正版本歷史紀錄」，而是套用先前曾經 commit 過的版本，看是「重新套用」或「反向套用」的差別而已。本篇文章將要來說明 Git 中的 Rebase 機制，這個所謂的 Rebase 機制就是真的用來修改 commit 紀錄的功能了，其功能重要而且強大。

準備本日練習用的版本庫

我們一樣先用以下指令建立一個練習用的工作目錄與本地儲存庫 (一樣先切換到 `C:\` 然後複製貼上就會自動建立完成)：

```
mkdir git-rebase-demo

cd git-rebase-demo
git init

echo 1 > a.txt
git add .
git commit -m "Initial commit (a.txt created)"

ping 127.0.0.1 -n 2 >nul

echo 2 > a.txt
git add .
git commit -m "Update a.txt to 2"

ping 127.0.0.1 -n 2 >nul

:: 建立並切換到 branch1 分支
git checkout -b branch1

echo b > b.txt
git add .
git commit -m "Add b.txt"

echo c > c.txt
git add .
git commit -m "Add c.txt"

echo 333 > c.txt
git add .
git commit -m "Update c.txt to 333"

echo d > d.txt
git add .
git commit -m "Add d.txt"

ping 127.0.0.1 -n 2 >nul

:: 切換到 master 分支
git checkout master

echo 3 > a.txt
git add .
git commit -m "Update a.txt to 3"
```

我們用 SourceTree 查看儲存庫的 commit graph (版本線圖) 如下：



使用 `git rebase` 命令的注意事項

首先，你的「工作目錄」必須是乾淨，工作目錄下的「索引」不能有任何準備要 commit 的檔案 (staged files) 在裡面，否則將會無法執行。

```
C:\git-rebase-demo>git rebase e663e52 -i
Cannot rebase: Your index contains uncommitted changes.
Please commit or stash them.
```

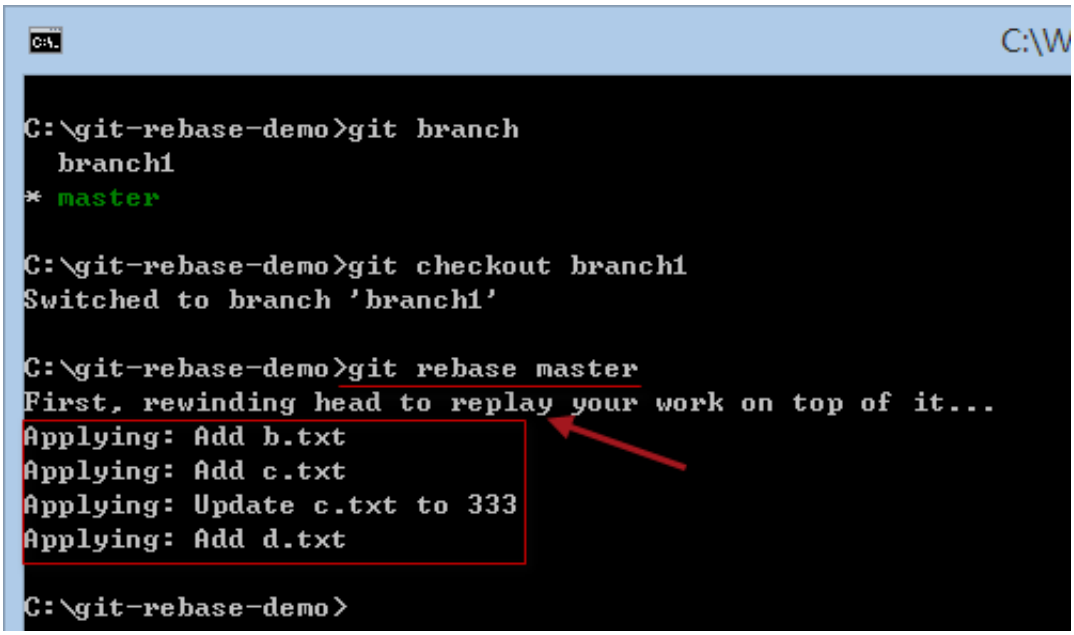
再來，也是最重要的，如果你的分支是從遠端儲存庫下載回來的，請千萬不要透過 Rebase 修改版本歷史紀錄，否則你將會無法將修改過後的版本送到遠端儲存庫！

Rebase 是什麼？

Rebase 是 "Re-" 與 "Base" 的複合字，這裡的 "Base" 代表「基礎版本」的意思，表示你想要重新修改特定分支的「基礎版本」，把另外一個分支的變更，當成我這個分支的基礎。

我們現在就來做一個簡單的 Rebase 示範，我們大概做幾件事：

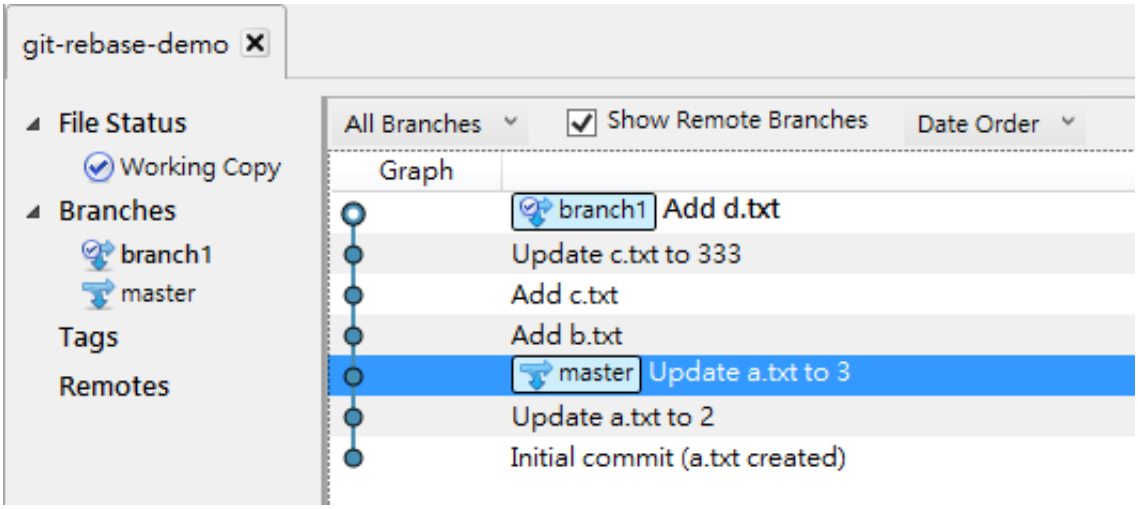
1. 切換至 `branch1` 分支： `git checkout branch1`
2. 然後執行 Rebase 動作，把 `master` 當成我們的基礎版本： `git rebase master`



請注意執行完 `git rebase master` 之後的顯示訊息，他說先將我們 `branch1` 分支的最新版本 (head)倒帶(rewind)到跟 `master` 一樣的分支起點(rewinding head)，然後再重新套用(replay)指定的 `master` 分支中所有版本。英文的 **on top of it** 代表的是讓 `branch1` 分支原本的變更套用在

`master` 上面，所謂的「上面」代表的是先套用 `master` 的版本，然後才套用 `branch1` 的版本(請見上圖的 `Applying:` 那幾行)。

我們看看套用完之後從 SourceTree 看到的版本線圖(commit graph)，你看看這是不是很神奇，版本線圖變成一直線了：



各位看官，看到上面的版本線圖，你會不會覺得「分支」的感覺不見了呢？事實上，分支並沒有改變，而是這幾個版本的「套用順序」被修改了。目前這張圖所代表的意思，就如同以下指令的執行順序：

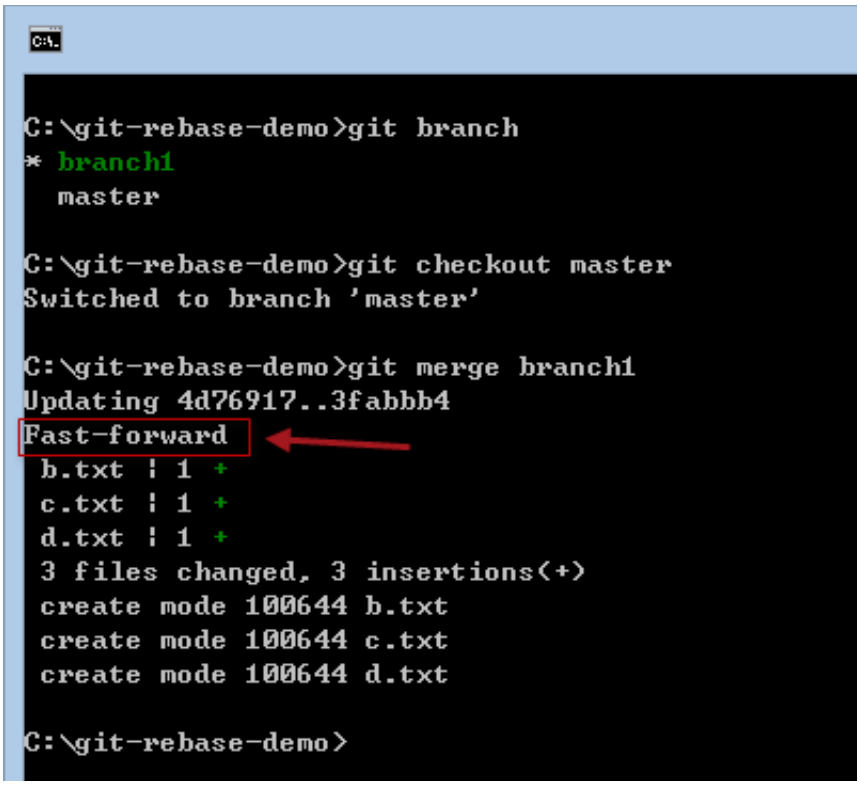
1. 建立 Initial commit (a.tx created)，同時預設建立 `master` 分支
2. 建立 Update a.txt to 2
3. 建立 Update a.txt to 3
4. 建立並切換至 `branch1` 分支
5. 然後不斷 commit 到 Add d.txt 這個版本

所以，這其實還是「兩個分支」喔，並沒有被合併成一個！千萬別認為這張圖只有一條線，所以只有一個分支。

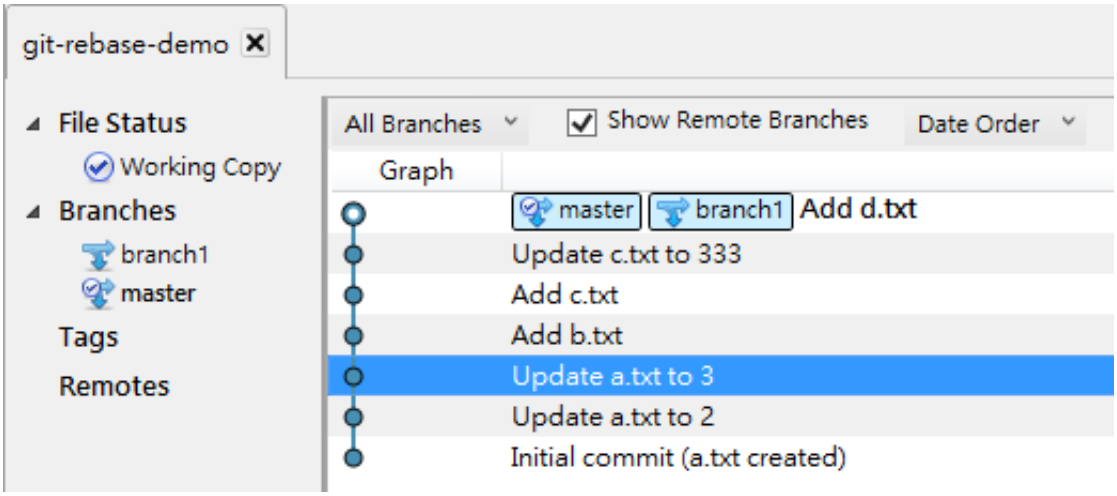
有分支，就有合併，現在的你，如果想要把 `branch1` 的變更，套用到 `master` 分支上，在使用過 Rebase 之後，你會有兩種合併的方式：

1. 透過一般合併指令，並觸發 Git 的快轉機制 (Fast-forward)

先切換到 `master` 分支，然後直接執行 `git merge branch1`，這時會引發 Git 的快轉機制(Fast-forward)。所謂的「快轉機制」，就是 Git 得知這個合併的過程，其實會依序套用 `branch1` 原本就有的變更，所以在合併的時候會直接修改 `master` 分支的 `HEAD` 參照絕對名稱，直接移動到 `branch1` 的 `HEAD` 那個版本。



最後我們得到的線圖還是一直線，但你可以看到 `master` 的分支已經移動到跟 `branch1` 一樣了。如下圖示：

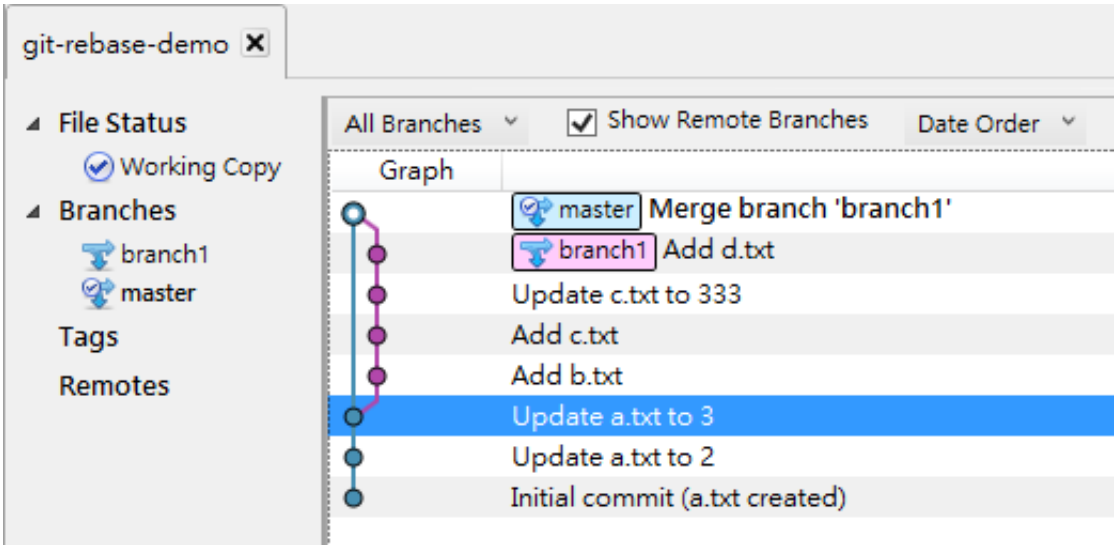


2. 透過 `--no-ff` 參數，停用 Git 的快轉機制

先切換到 `master` 分支，然後直接執行 `git merge branch1 --no-ff` 即可。

```
C:\git-rebase-demo>git reset --hard HEAD@{1}
HEAD is now at 4d76917 Update a.txt to 3

C:\git-rebase-demo>git branch
  branch1
* master

C:\git-rebase-demo>git merge branch1 --no-ff
Merge made by the 'recursive' strategy.
 b.txt | 1 +
 c.txt | 1 +
 d.txt | 1 +
 3 files changed, 3 insertions(+)
 create mode 100644 b.txt
 create mode 100644 c.txt
 create mode 100644 d.txt
```

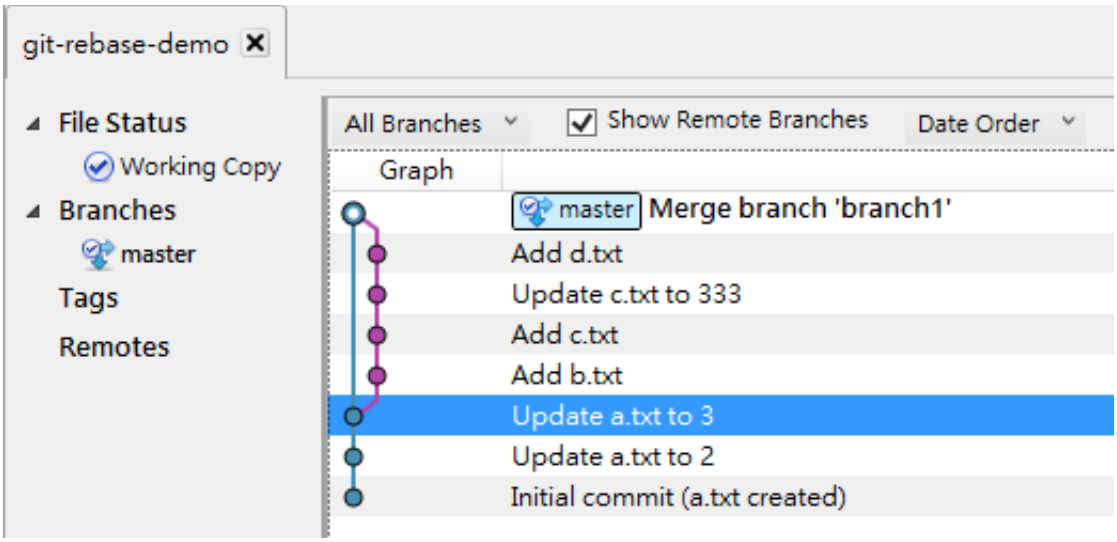
當你合併時指定停用 Git 的快轉機制，那就代表「不允許快轉」的意思。也代表著，他會強迫你打算合併的那個 `branch1` 先建立一個分支，然後最後再合併回 `master`，也代表著我們在次修變更了 `branch1` 的版本線圖。最終，你看到的版本線圖應該會長成以下這個樣子，不是比剛剛一直線的版本線圖還漂亮呢！ :-)



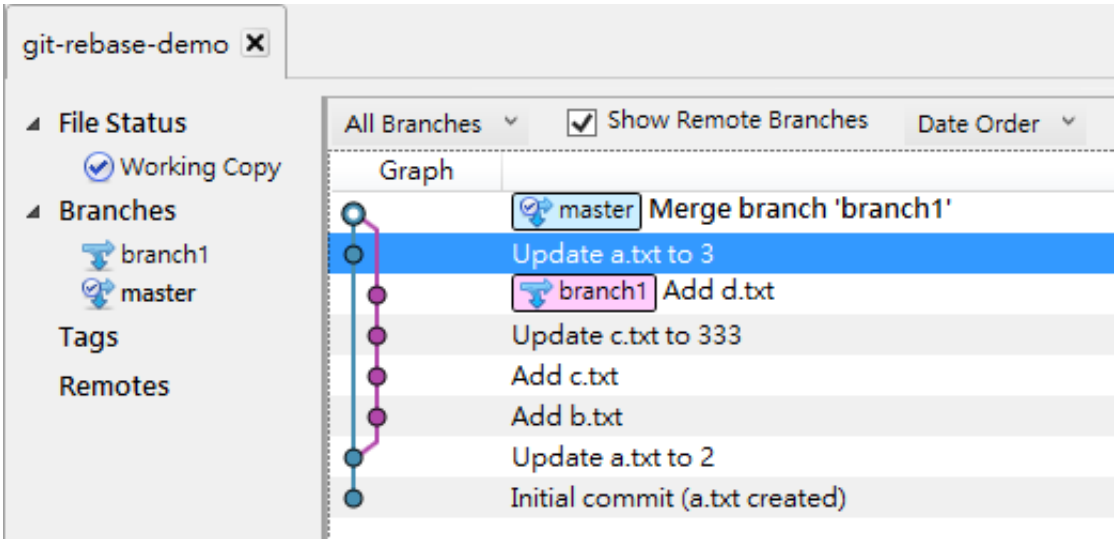
最後，如果你的 `branch1` 用不到的話，就可以把這個分支給刪除： `git branch -d branch1`

```
C:\git-rebase-demo>git branch
  branch1
* master

C:\git-rebase-demo>git branch -d branch1
Deleted branch branch1 (was 3fabbb4).

C:\git-rebase-demo>git branch
* master

C:\git-rebase-demo>_
```

最終我們的版本線圖如下：



我們來比對一下，如果用我們最剛開始的建立的初始版本進行合併的話，線圖會長得像以下這樣。各位有沒有發現，我們原本的 `branch1` 是從 **Update a.txt to 2** 這一版開始分支的，經過我們透過 Rebase 之後，分支的起點不太一樣了，而是改由 **Update a.txt to 3** 這個分支開始，是不是很有趣呢！



今日小結

第一次接觸 Rebase 的人，或許會覺得很抽象，各位必須細心品味，才能真正感受到 Rebase 帶來的強大威力。之後的文章裡，我還會更加詳細的介紹 Rebase 的進階用法。

我重新整理一下本日學到的 Git 指令與參數：

- git rebase master
- git merge branch1
- git branch -d branch1