目錄

Git

這份教學適用於任何想自學 Git，或是想要更了解 Git 的人

教學內容會由淺入深，逐步把 Git 的概念全部帶出來

內容三大部分：

1. 本機篇
    - 檔案管理
    - Patch 管理
    - 分支管理
2. 遠端篇
3. 進階篇

（本教學中會使用 **patch** 一詞替代 **"名詞"** 的 **commit**，詳細內容請看 這裡 最下面）

講解時，全都是在終端機以 `command line` 的方式進行操作

- 使用 Windows 的人建議安裝 Git 官方所提供的 `Git Bash`

爲何要學 Command Line

可能很多人都已經很習慣使用視覺操作介面 (GUI) 的軟體，但爲何要學 Command Line 操作呢？

Git 原本就被是設計成 command line，因此用它可以更貼近原作者 (Linus Torvalds) 的設計概念

任何第三方所開發的 Git 視覺操作介面，其實最終都是透過 command line 去呼叫 git 的指令

雖然視覺操作介面很方便，但是每個軟體的操作都不盡相同（尤其是一些比較複雜的 git 指令）

搞懂這些 git 指令後，不論要使用哪個 GUI 都會非常容易上手，而切換時也不用從頭再學一次

若未來要在 server 上透過 git 來部署應用程式，也只能透過 command line 來操作呀！

爲了要學習會最原汁原味的 Git，建議使用 command line 會是最有投資報酬率的方式！

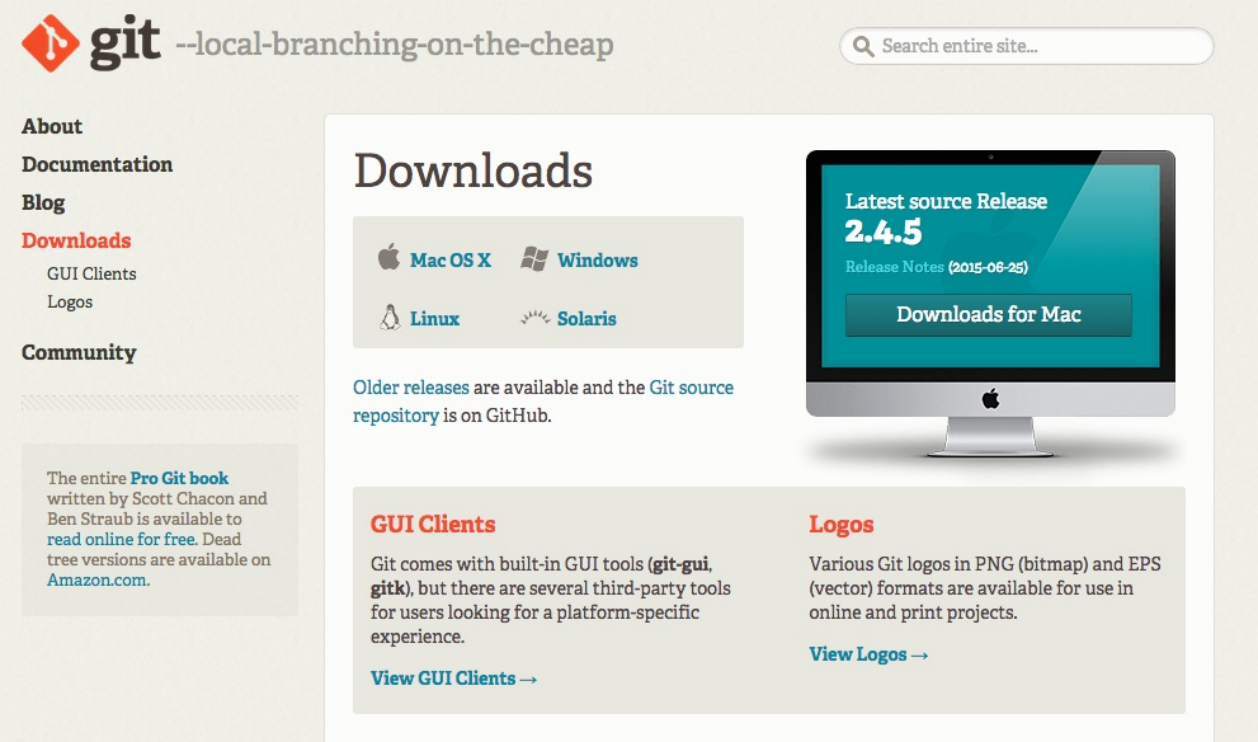
作者

@zlargon

https://github.com/zlargon

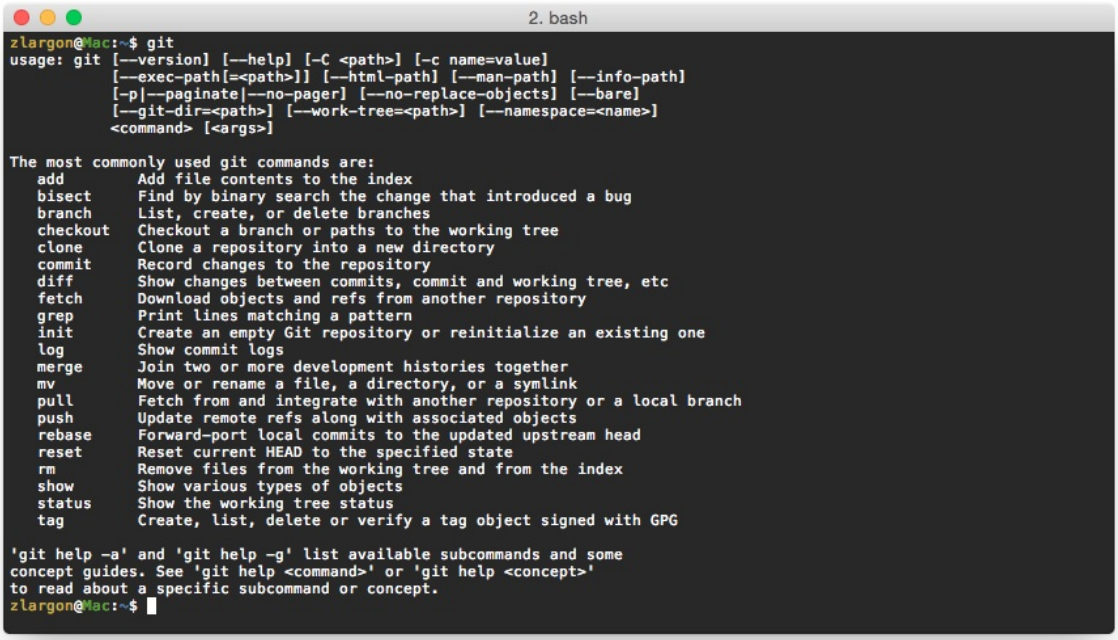
安裝

https://git-scm.com/downloads



Linux / Mac OSX

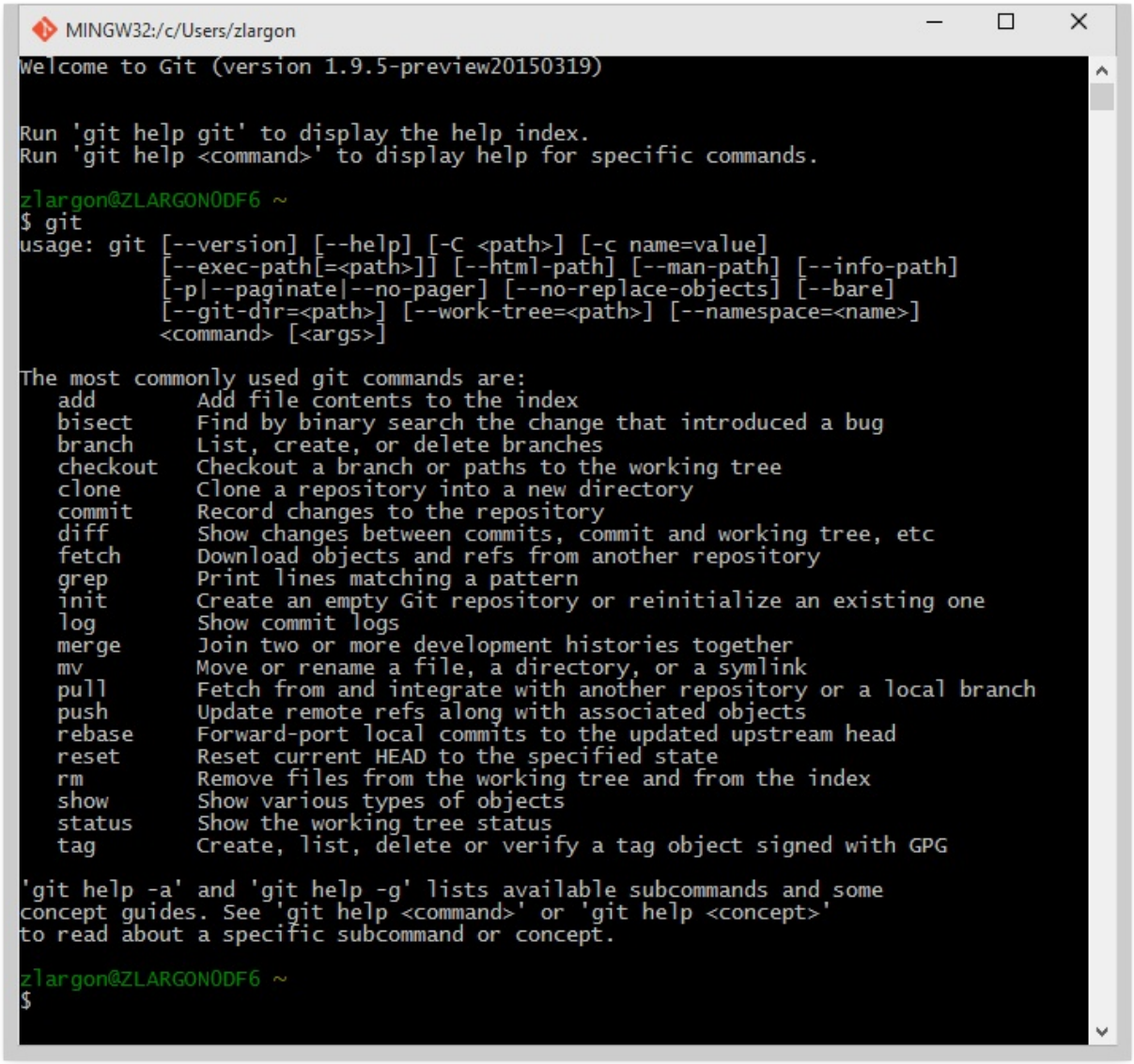
安裝好之後，就可以立即在終端機使用



Windows

`Git Bash = MinGW + Git`

`MinGW` 讓我們可以在 Windows 的環境下，使用與 UNIX-like 的方式下指令 `cd`, `ls`, `vim`, `curl`, ...


MINGW32:/c/Users/zlargon
Welcome to Git (version 1.9.5-preview20150319)
Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.
zlargon@ZLARGON0DF6 ~
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
[-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
<command> [<args>]
The most commonly used git commands are:
add Add file contents to the index
bisect Find by binary search the change that introduced a bug
branch List, create, or delete branches
checkout Checkout a branch or paths to the working tree
clone Clone a repository into a new directory
commit Record changes to the repository
diff Show changes between commits, commit and working tree, etc
fetch Download objects and refs from another repository
grep Print lines matching a pattern
init Create an empty Git repository or reinitialize an existing one
log Show commit logs
merge Join two or more development histories together
mv Move or rename a file, a directory, or a symlink
pull Fetch from and integrate with another repository or a local branch
push Update remote refs along with associated objects
rebase Forward-port local commits to the updated upstream head
reset Reset current HEAD to the specified state
rm Remove files from the working tree and from the index
show Show various types of objects
status Show the working tree status
tag Create, list, delete or verify a tag object signed with GPG
'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
zlargon@ZLARGON0DF6 ~
$

配置

設定 username, email

初次使用 git，最重要的就是設定 `username` 以及 `email`

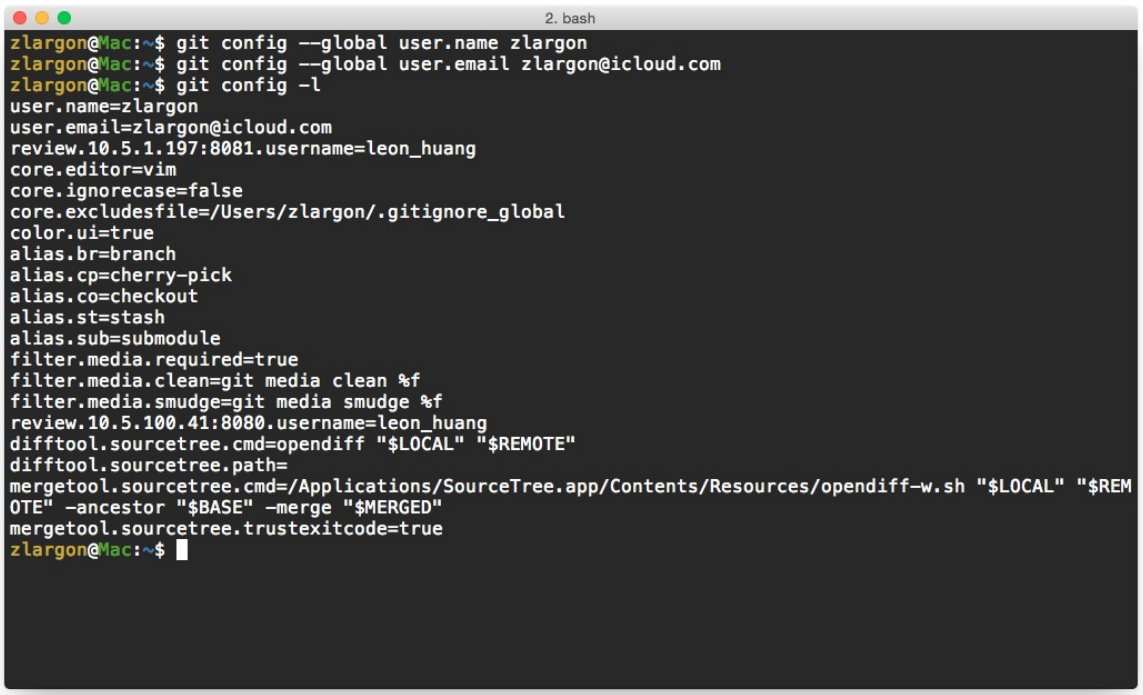
```
$ git config --global  user.name "zlargon"
$ git config --global  user.email "zlargon@icloud.com"
```

第一次使用可以先隨意設定，但是之後若要搭配 github 的服務時，就必須使用在 github 註冊時所使用的 username, email

> 使用 `--global` 的參數，表示對於所有的 ***git project*** 都會採用這組預設值

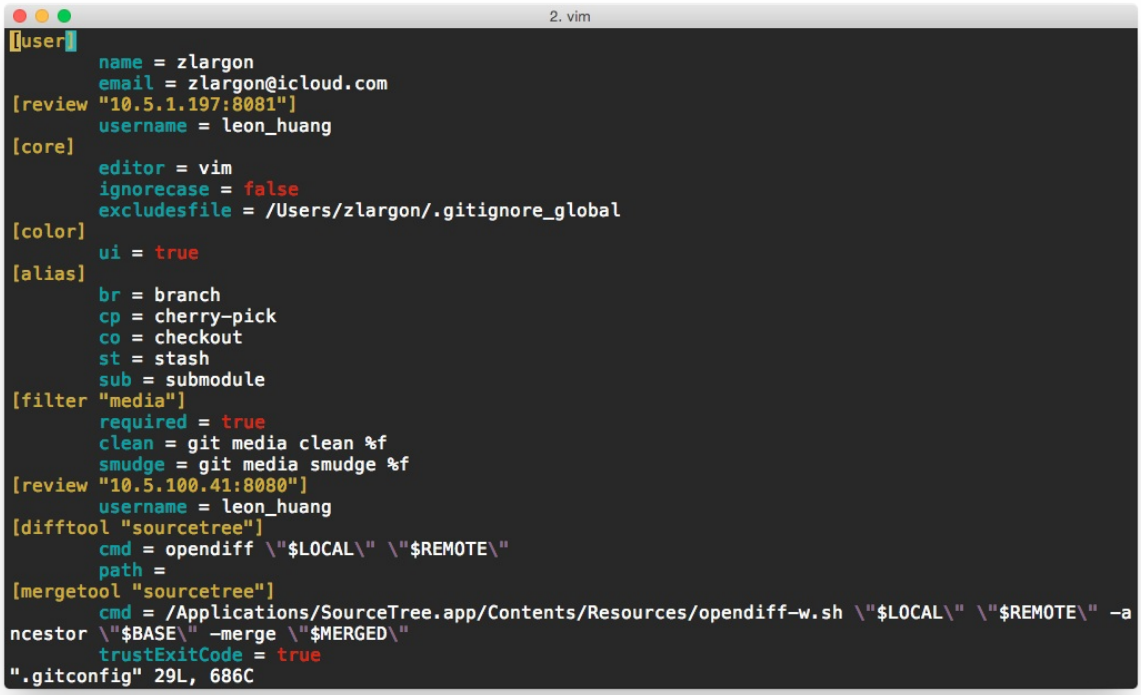
接著，我們可以透過 `git config -l` 的指令，來查看我們目前 git 的設定內容

```
$ git config -l
```



> 參數 `-l` 即爲 `--list`，等同於以下的指令
>
> `$ git config --list`

而所有 `git config --global` 的設定內容，都會被寫入 `~/.gitconfig`



預設開啓彩色

```
$ git config --global  color.ui true
```

預設編輯器 vim

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

檔案名稱的大小寫不同時，是否該視爲同一個檔案

在 Windows, Mac OSX 底下的 File System，會將名稱相同，但大小寫不同的檔案，視爲同一個檔案

例如：`file.txt` 和 `FILE.TXT` 會被視爲相同的檔案

只有在 Linux 下，才會把 `file.txt` 和 `FILE.TXT` 視爲不同檔案

我們可以從 git 裡面，去強制設定，是否要忽略檔案名稱的大小寫

```
$ git config --global  core.ignorecase true    # 忽略大小寫
$ git config --global  core.ignorecase false   # 強制區分大小寫
```

> Note:
>
> 我之前有遇過，使用 Linux 開發的人，上傳了兩個檔案，名稱完全一樣，只有開頭大小寫不同
>
> 導致使用 Mac 的人下載下來的時候，被系統視爲是同一的檔案
>
> 所以就會變得很怪，暨無法修改也無法刪除
>
> 從此之後，我們在 Linux 上都一律忽略大小寫

設定 git 指令的別名

可以把一些常用，而且較長的指令，簡化得短一點，方便自己使用

以下是我個人喜好的別名設定：

```
$ git config --global  alias.br   branch
$ git config --global  alias.co   checkout
$ git config --global  alias.cp   cherry-pick
$ git config --global  alias.st   stash
$ git config --global  alias.sub  submodule
```

設定自動完成 git 指令

官方 git-completion.bash 腳本

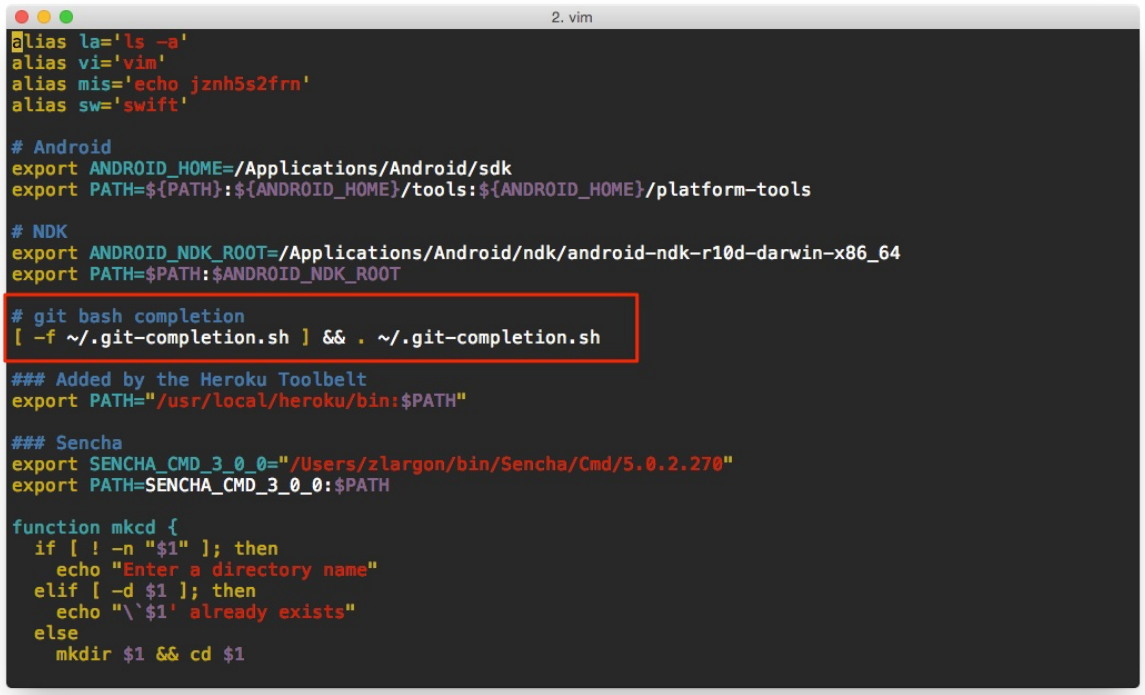
只要打指令的前幾的字，或是 branch 名稱的前幾個字，就可以用 TAB 自動完成剩下的部分

> 若你所安裝的 ***git*** 預設已經有這個功能，則不需要安裝

如何安裝設定：

- 下載 git-completion.bash，存到 ~/.git-completion.sh
- 在 `~/.bash_profile` 加上 `[ -f ~/.git-completion.sh ] && . ~/.git-completion.sh`
- 重開終端機

```
$ curl https://raw.githubusercontent.com/git/git/master/contrib/completion/git-complet
ion.bash > ~/.git-completion.sh
$ echo "" >> ~/.bash_profile
$ echo "# git completion" >> ~/.bash_profile
$ echo "[ -f ~/.git-completion.sh ] && . ~/.git-completion.sh" >> ~/.bash_profile
$ source ~/.bash_profile
```



初體驗

建立 Git 專案

```
git init
```

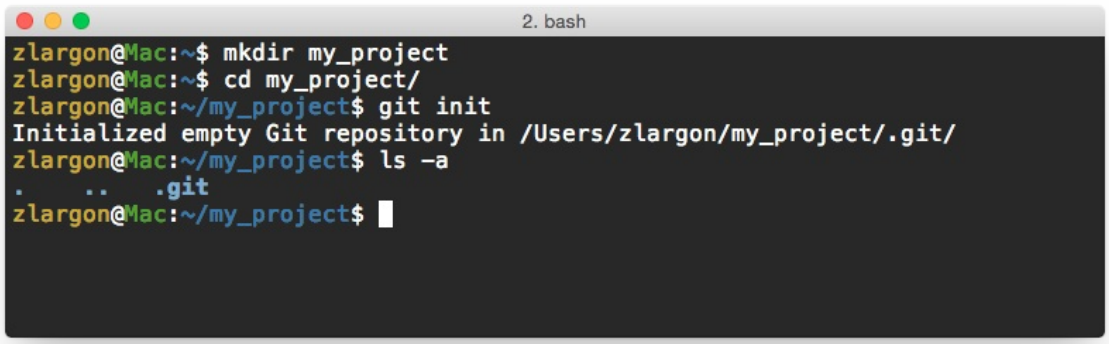
提交一個 Patch

```
git status
git add <file>
git commit
git log
git show
```

建立 Git 專案

初始化一個新的 git project，首先要建立一個新資料夾 (my_project)，進到資料夾並且執行 `git init` 指令

```
$ mkdir my_project
$ cd my_project
$ git init
```



從這裡我們可以看到，在 `my_project/` 底下，多了一個 `.git` 的資料夾

其實 `.git` 就是一個版本控制的小資料庫，裡面放了所有關於這個 project 的資訊

這樣一個 git 的專案就已經初始化完成了

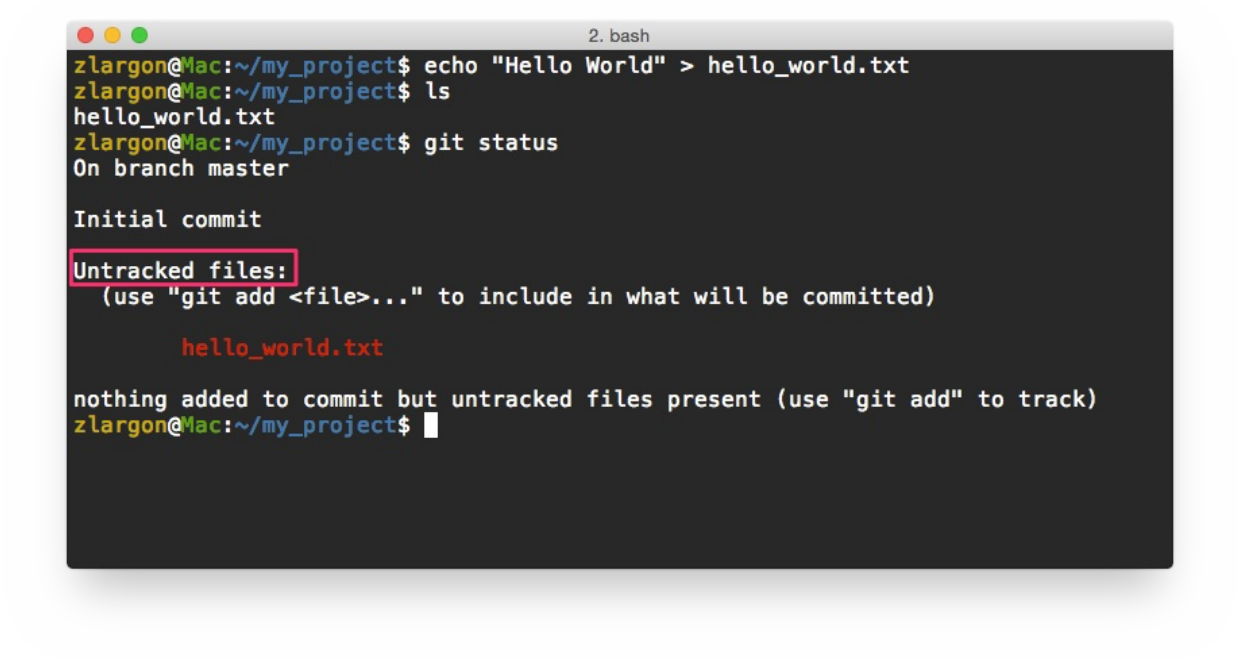
讓我們來新增檔案吧！

提交一個 Patch

首先，我們先新增一個 `hello_world.txt` 的檔案，內容爲一行字串 `Hello World`

使用 `git status` 來檢視所有檔案的狀態

```
$ git status
```



我們這裡可以看到，目前 `hello_world.txt` 的狀態爲 ***Untracked files***

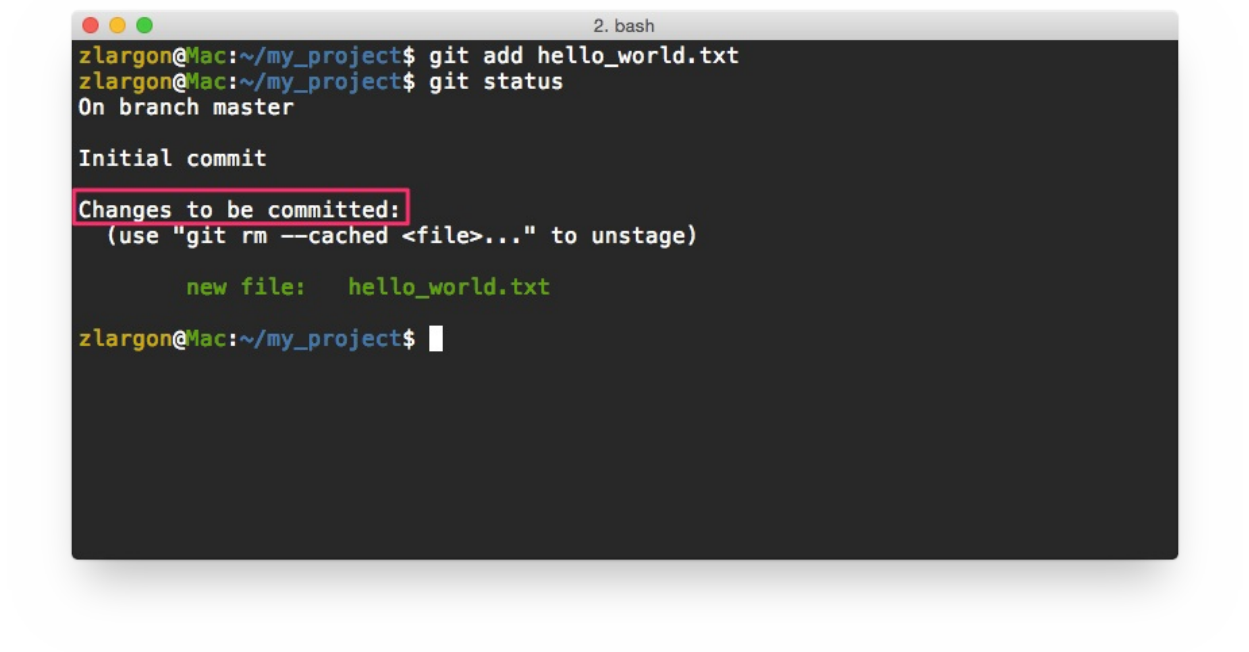
表示這是一個全新的檔案（稍後我們會在 "檔案狀態" 做更詳細的解釋）

使用 `git add <file>` 來告知 git，哪些是我們即將要提交（commit）的檔案

我現在現在打算要提交 `hello_world.txt`

```
$ git add hello_world.txt
$ git status
```

這時我們再用 `git status` 來檢視檔案的狀態



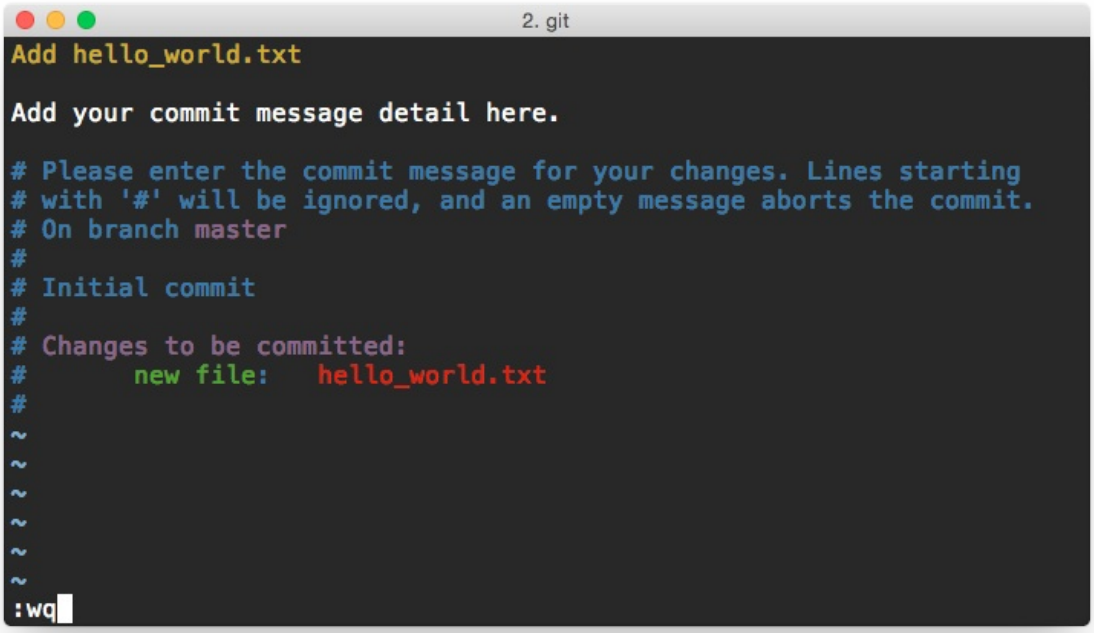
`hello_world.txt` 的狀態改變成綠色的 ***Changes to be committed***

表示這個檔案已經 "準備好" 要被提交（commit）了（稍後我們會在 "檔案狀態" 做更詳細的解釋）

使用 `git commit` 來提交一個 patch

```
$ git commit
```

這時候會進入 `vim` 的文字編輯模式，編輯提交訊息（commit message）

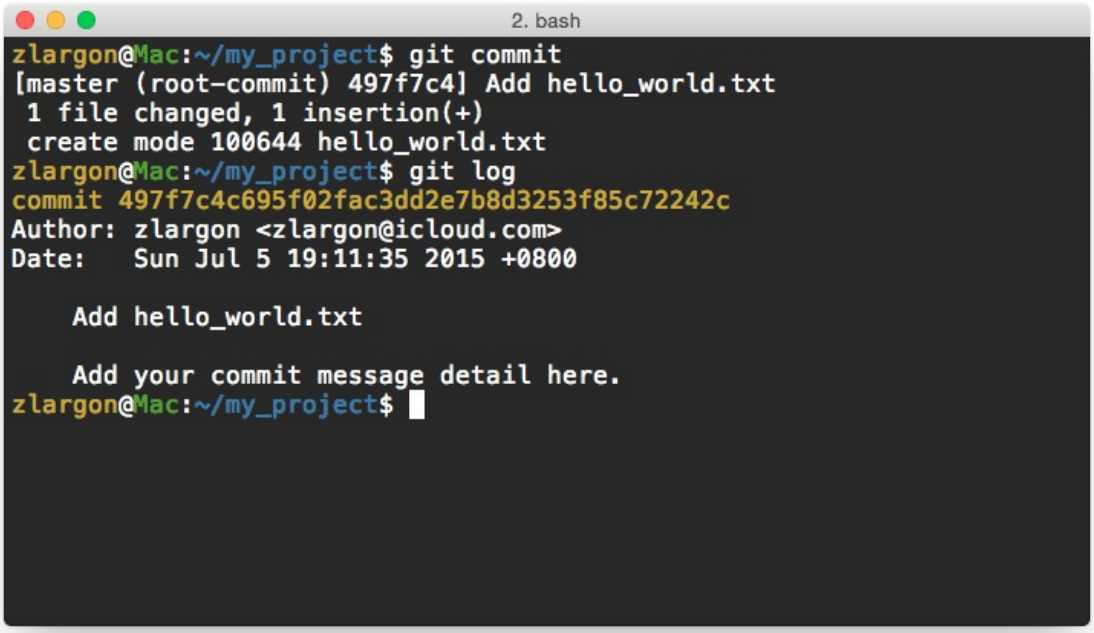


- 第一行爲 commit message 的標題（僅限一行）
- 第二行保留空白
- 第三行以後是 commit message 的內容（可略過不寫）

編輯完成後，存檔離開，便可完成這次的提交

使用 `git log` 檢視提交的歷史訊息

使用 `git log` 來檢視之前的提交的歷史訊息，他將會列出所有 patch 的資訊




```
commit 497f7c4c695f02fac3dd2e7b8d3253f85c72242c
Author: zlargon <zlargon@icloud.com>
Date:   Sun Jul 5 19:11:35 2015 +0800

    Add hello_world.txt

    Add your commit message detail here.
```

`commit` 是 git 幫我們自動產生的長度 40 字元的 Hash 值，並且保證 commit id 絕對不會重複，用來識別所有不同的 patch

`Author` 即爲我們在前幾個章節透過 `git config` 所設定的 username 跟 email

> username 及 email 爲 commit 時的必要欄位，因此必須事先設定好

下半部爲提交訊息，包含標題及內容

這時候再做 `git status` 時會顯示

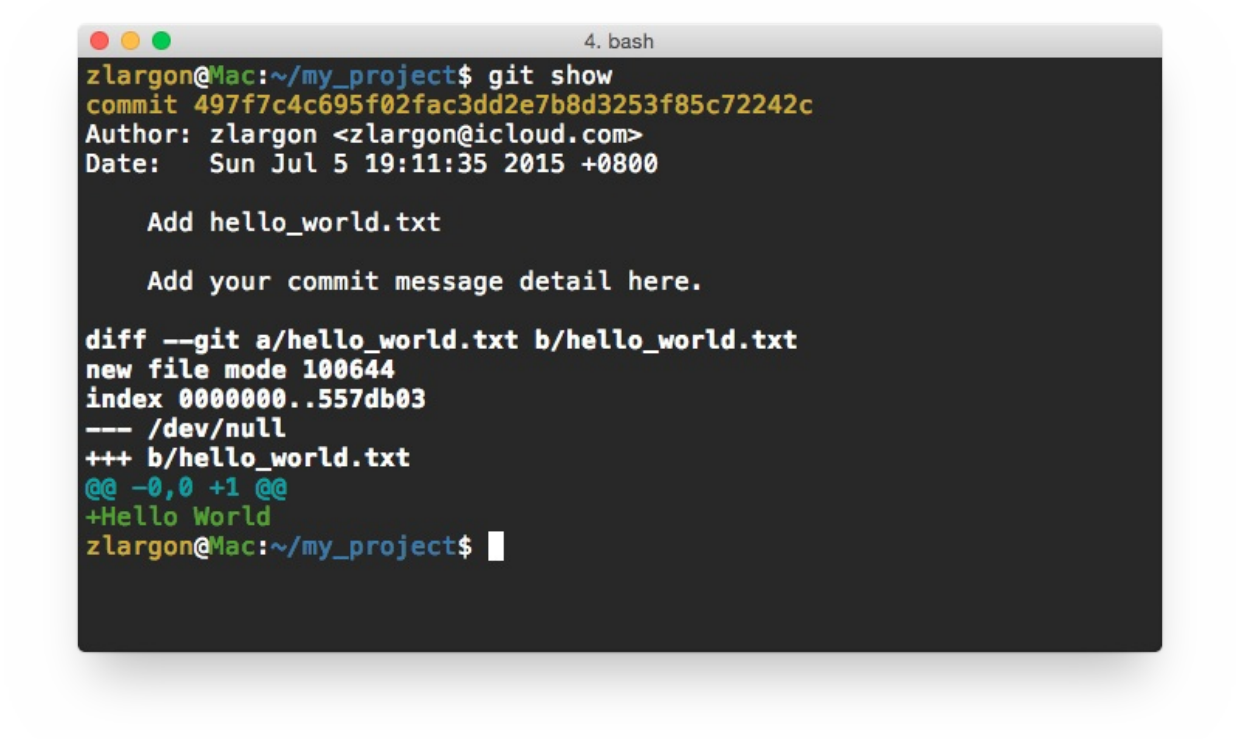
```
On branch master
nothing to commit, working directory clean
```

表示目前此目錄下，沒有發生任何的更動

使用 `git show` 檢視 patch 的修改內容

```
$ git show
```

使用 `git show` 來檢視最後一次提交的 patch 所修改的內容



從 `git show` 可以看出，這個 patch 新增了一個 `hello_world.txt` 的檔案，內容爲 `Hello World`

關於 Commit 與 Patch

本教學中所用到的 "patch" 其實正確來說，應該要用 "commit" 來稱呼

是沒錯，跟 git commit 一樣，也是叫 commit

不過一個是動詞，一個是名詞

對初學者來說，這還滿令人混淆的

比較正確的說法，應該是 "commit a commit"，而不是 "commit a patch"

然而在英文字典中，commit 只有動詞的字義，而找不到名詞的字義

```
commit (v.) 犯罪；託付
```

git commit 比較接近於 "託付" → 交託 → 提交（submit）

> 其實 commit 不只有這兩個動詞字義，有興趣的人可以去查大字典 :D
>
> 其名詞型爲 commitment，表示 "承諾" 的意思

Git 的作者已經完全將 commit 給名詞化了

不過 "commit a commit" 聽起來實在太繞口了，連外國人自己都不會這樣講

通常他們會講 "make/create a commit" 或是 "commit a change"

而我在教學裡面最後是決定用 patch 來取代稱呼 commit (n.)

希望可以降低 commit (v.) 跟 commit (n.) 對初學者造成的混淆

有些 open source 的文件在說明如何貢獻原始碼的時候，也可以看到 "commit/submit a patch" 的講法

會選用 patch 有部分是受到 Gerrit 的啓發

但最主要是因爲 git 裡面有一個指令就叫做 `format-patch`

`format-patch` → 將 patch 格式化，產生實體檔案（patch file）

簡單來說，就是可以將一個或多個 commit (n.) 打包起來，然後用其他方式傳給別人（例如：Email）

這是一個比較進階，而且少見的指令，幾乎是不會用到的

只有少部分 open source 會要求貢獻者使用 patch file 來提交程式碼（例如：FFmpeg）

整體來說，commit (n.) 跟 patch 的概念是很相似的

雖然這不是正統的講法，但會相對比較好理解

本章回顧

- 使用 `git status` 來檢視所有檔案的狀態
- 使用 `git add <file>` 來告知 git，哪些是我們即將要提交（commit）的檔案
- 使用 `git commit` 來提交一個 patch，並且使用 `vim` 編輯提交訊息（包含標題及內容）
- 使用 `git log` 檢視提交的歷史訊息
- 使用 `git show` 檢視最後一次提交的 patch 所修改內容

檔案管理

新增 / 修改檔案

```
git diff
git diff <file>
git diff --cached
git add -A
git commit -m <message>
git show <commit_id>
```

刪除檔案

```
git rm <file>
git add -u
```

搬移檔案

```
git mv <file> <directory>
```

重新命名檔案

```
git mv <file> <new_name>
```

檔案狀態

```
git reset HEAD
git reset HEAD <file>
git checkout -- <file>
```

檔案還原

```
git checkout -- <file>
git reset HEAD
git reset HEAD <file>
git reset --hard HEAD
```

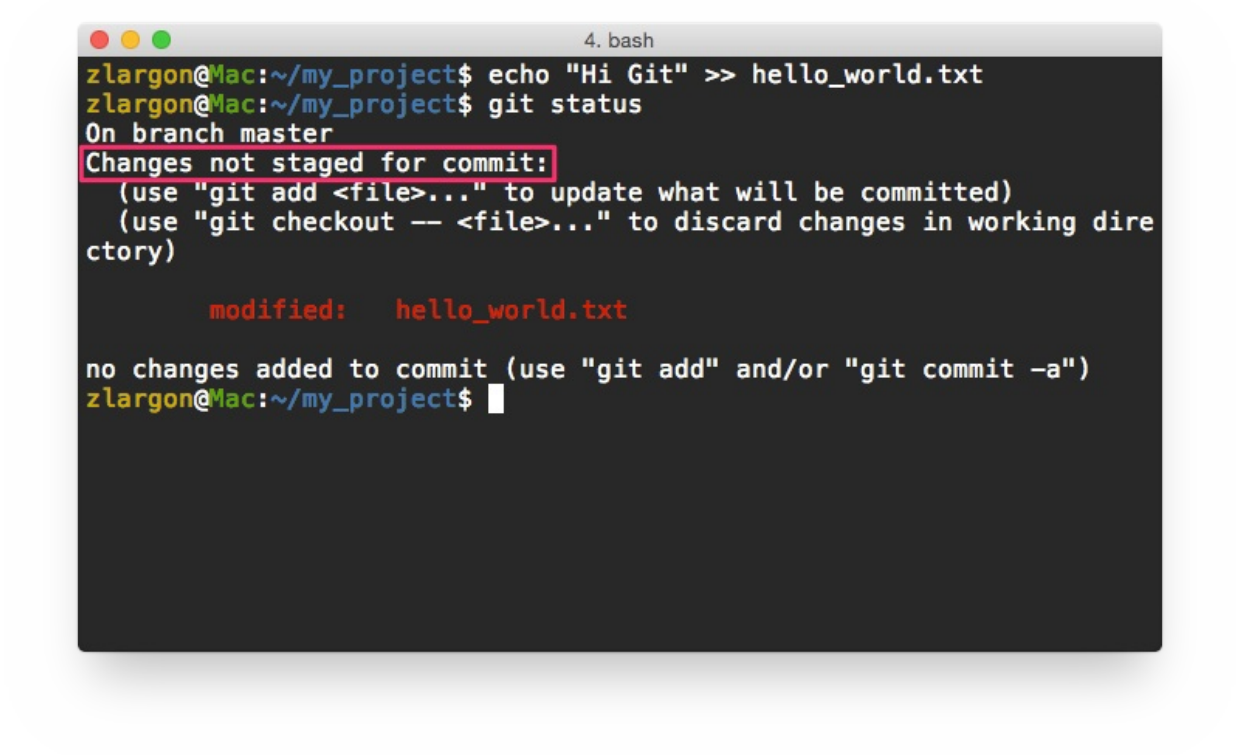
忽略檔案

```
git add -f <file>
```

新增 / 修改檔案

我們現在對剛才所新增的 `hello_world.txt` 進行修改

在檔案的最後面新增一行字串 "Hi Git"，並且使用 `git status` 來查看檔案狀態

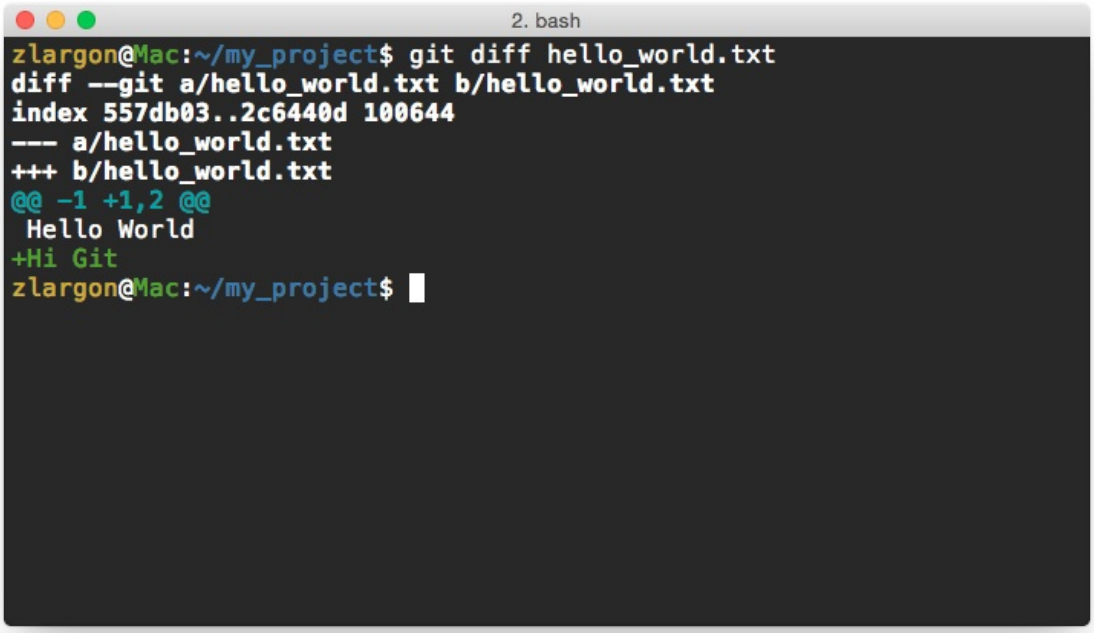


`hello_world.txt` 的狀態變成 ***Changes not staged for commit***

表示這個檔案被修改了，但是尚未要進行提交（稍後我們會在 "檔案狀態" 做更詳細的解釋）

使用 `git diff <file>` 查看 "特定" 被修改檔案的內容

```
$ git diff hello_world.txt
```



透過 `git diff` 可以看出 `hello_world.txt` 的最底下新增了一行 "Hi Git" 的字串

`git diff` 其實是把當前的狀態，與最後一個 patch 做比對

```
+Hi Git
```

"Hi Git" 前面的加號，表示新增的意思

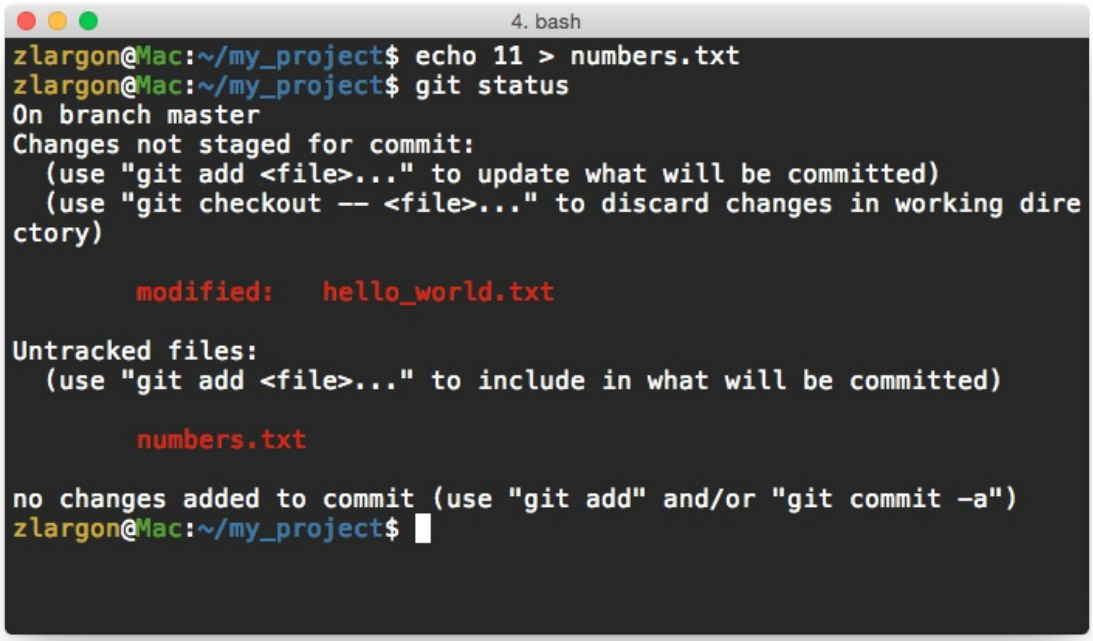
使用 `git diff` 查看 "全部" 被修改檔案的內容

若有多個檔案被同時修改，可以直接使用 `git diff` 查看所有的檔案被修改的內容

由於我們目前只有一個被修改的檔案，因此效果會與 `git diff hello_world.txt` 相同

現在我們另外新增一個檔案 `numbers.txt` 內容爲一行字串 "11"

然後再用 `git status` 去檢視狀態



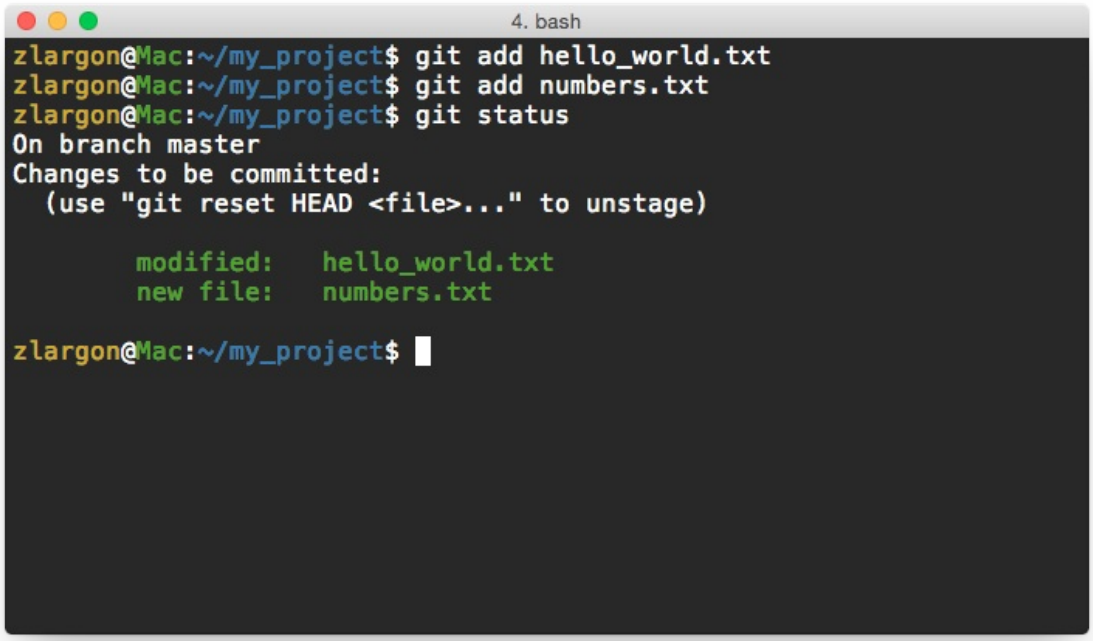
我們可以看到 `numbers.txt` 的狀態是 ***Untracked files***

這時候我們再做一次 `git diff` ，依然只能看到 `hello_world.txt` 的變化，但是看不到 `numbers.txt` 的部分

這是因爲 `numbers.txt` 的狀態是 ***Untracked files***，所以 git 本來就不會去追蹤他內容的變化

接下來，我們要用 `git add` 來告知 git，哪些是我們將要 commit 的檔案

```
$ git add hello_world.txt
$ git add numbers.txt
```



檔案狀態改變：

- **hello_world.txt**

  Changes not staged for commit → Changes to be committed

- **numbers.txt**

  Untracked files → Changes to be committed

使用 `git add -A` 加入全部的檔案

git 有提供一個快速的方法，可以一次 `add` 全部的檔案，那就是在 `git add` 後面加上 `-A` 或是 `--all` 的參數

```
$ git add -A        # 一次 add 所有的檔案
$ git add --all     # 同上
```

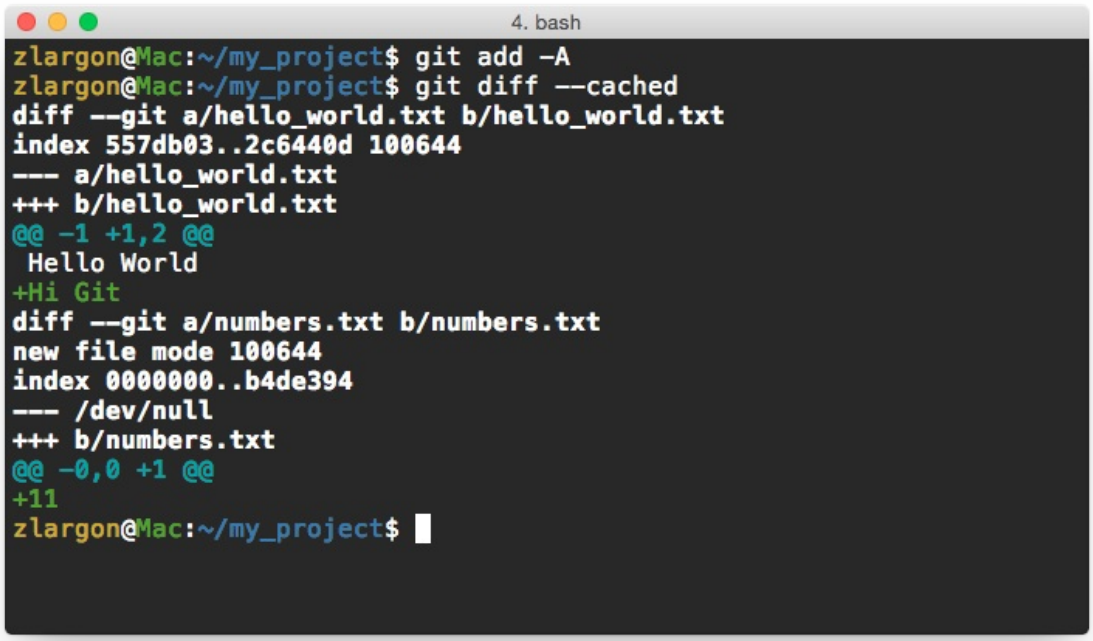
不論檔案狀態是 ***Untracked files*** 或是 ***Changes not staged for commit***（紅色），都會一口氣變成 ***Changes to be committed***（綠色）

使用 `git diff --cached` 來檢視 ***Changes to be committed*** 的修改內容

`git diff` 只能檢視 ***Changes not staged for commit*** 區塊的修改內容

如果想要檢視 ***Changes to be committed*** 的修改內容，就必須在後面加上 `--cached` 或是 `--staged` 的參數

```
$ git diff --cached     # 檢視綠色部分的內容
$ git diff --staged     # --staged 等同於 --cached
```



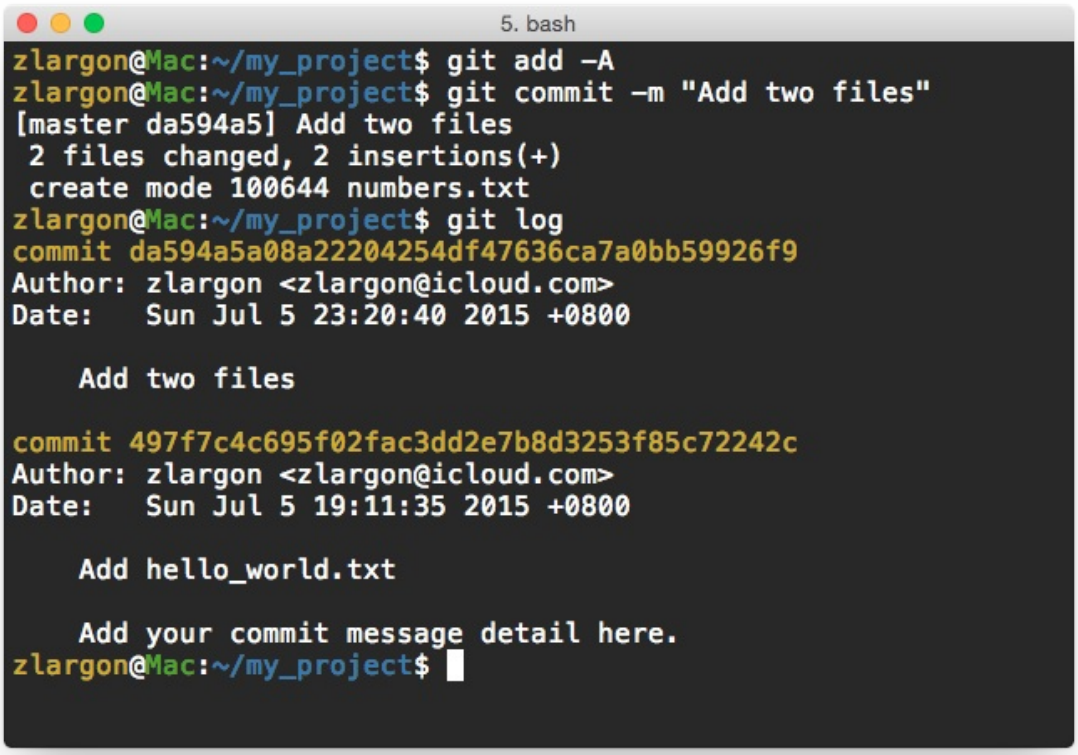
使用 `git commit -m <message>` 來提交（不啓動文字編輯模式）

`git commit` 後面可以接數個 `-m <message>`，每一段的 message 都會被一個空行隔開

```
$ git commit -m <title>                             # 只要提交 title
$ git commit -m <title> -m <message>                # 提交 title 以及 message
$ git commit -m <title> -m <msg1> -m <msg2> ....    # 提交多個 messages
```

使用 `git commit -m` 完之後，再用 `git log` 檢視提交歷史訊息

```
$ git commit -m "Add two files"
$ git log
```



使用 `git show <commit_id>` 來檢視之前提交的 patch 所修改的內容

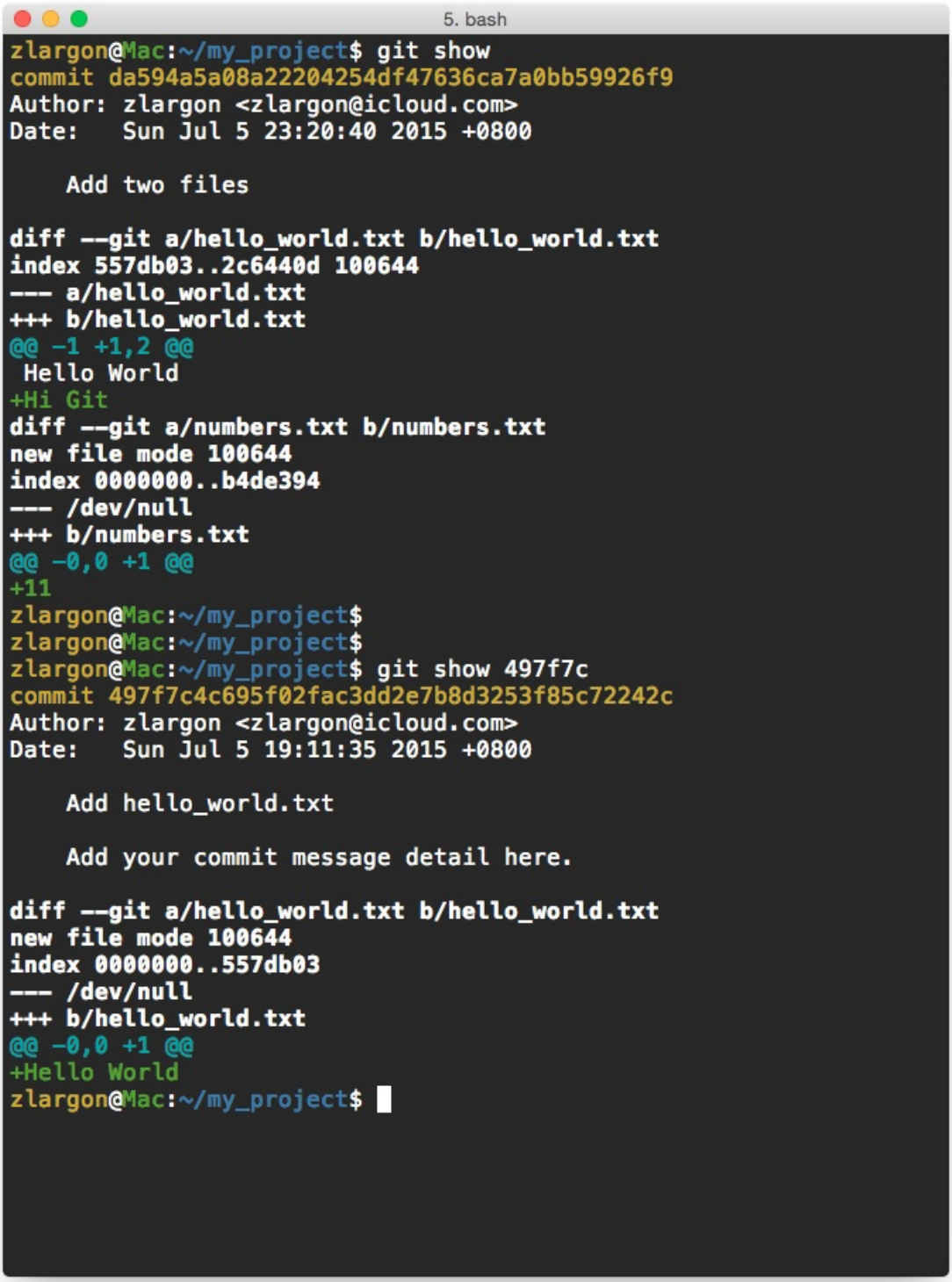
現在我們的專案裡面，已經有兩個 patch 了

`git show` 只能查看最後一次提交的 patch 所修改內容

若要看其他的 patch 就必須要在後面加上 `commit id`

通常只要給 `commit id` 的前六碼，git 就可以認得了

```
$ git show                                          # 查看最後一次 commit 的 patch
$ git show 497f7c4c695f02fac3dd2e7b8d3253f85c72242c # 查看首次 commit 的 patch
$ git show 497f7c                                   # 同上
```



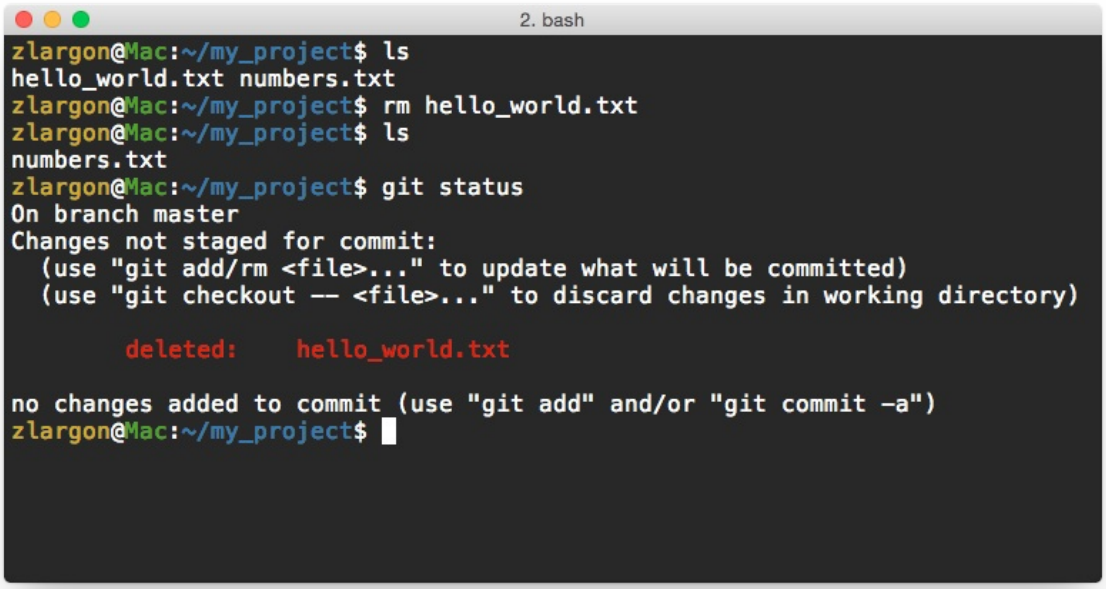
本章回顧

- 使用 `git diff` 查看 "全部" 檔案被修改的內容

- 使用 `git diff <file>` 查看 "特定" 檔案被修改的內容
- 使用 `git diff --cached` 查看 *Changes to be committed* 的修改內容
- 使用 `git add -A` 加入全部的檔案
- 使用 `git commit -m <message>` 來提交（不啓動文字編輯模式）
- 使用 `git show <commit_id>` 來查看特定 patch 修改的內容

刪除檔案

我們現在把 `hello_world.txt` 的檔案刪除，然後再用 `git status` 來檢視檔案的狀態



使用 `git rm <file>` 來告知 git，哪些是我們將要刪除的檔案

`hello_world.txt` 的檔案狀態是 ***Changes not staged for commit (deleted)***

由於 `hello_world.txt` 已經被刪除，所以在這裡不能用 `git add` 來 "新增" 檔案

這裡要用 `git rm` 來 "刪除" 檔案（ `rm` 其實就是 `remove` 的意思）

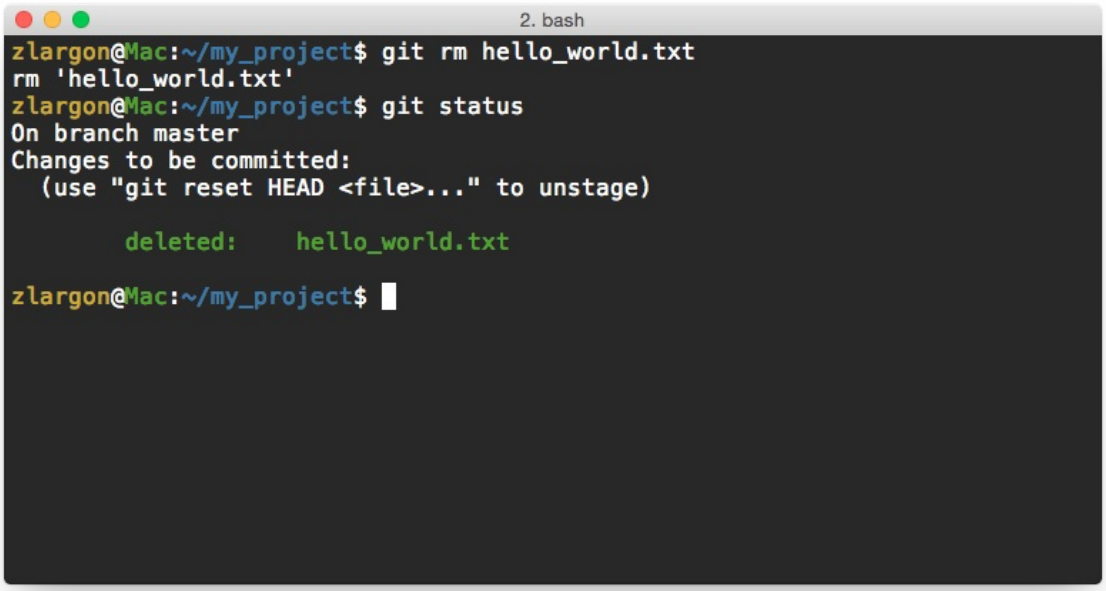
> 事實上，在某些新版的 git 是可以使用 `git add` 來 "新增" 將要被除的檔案

在觀念上，`git rm` 跟 `git add` 其實是完全一致的

	step 1	step 2
Add File	新增一個檔案	使用 `git add` 告知 git，哪些是我們將要 "新增" 的檔案
Remove File	移除一個檔案（可略過）	使用 `git rm` 告知 git，哪些是我們將要 "移除" 的檔案

```
$ git rm hello_world.txt
$ git status
```

我們已經告知 git 將要把 `hello_world.txt` 刪除，接著再用 `git status` 來查看檔案狀態



`hello_world.txt` 的狀態改變爲綠色的 ***Changes to be committed (deleted)***

使用 `git add -u` 加入所有被更動的檔案（包含 *modified* 及 *deleted*）

雖然我們不能用 `git add <file>` 來加入已被刪除的檔案

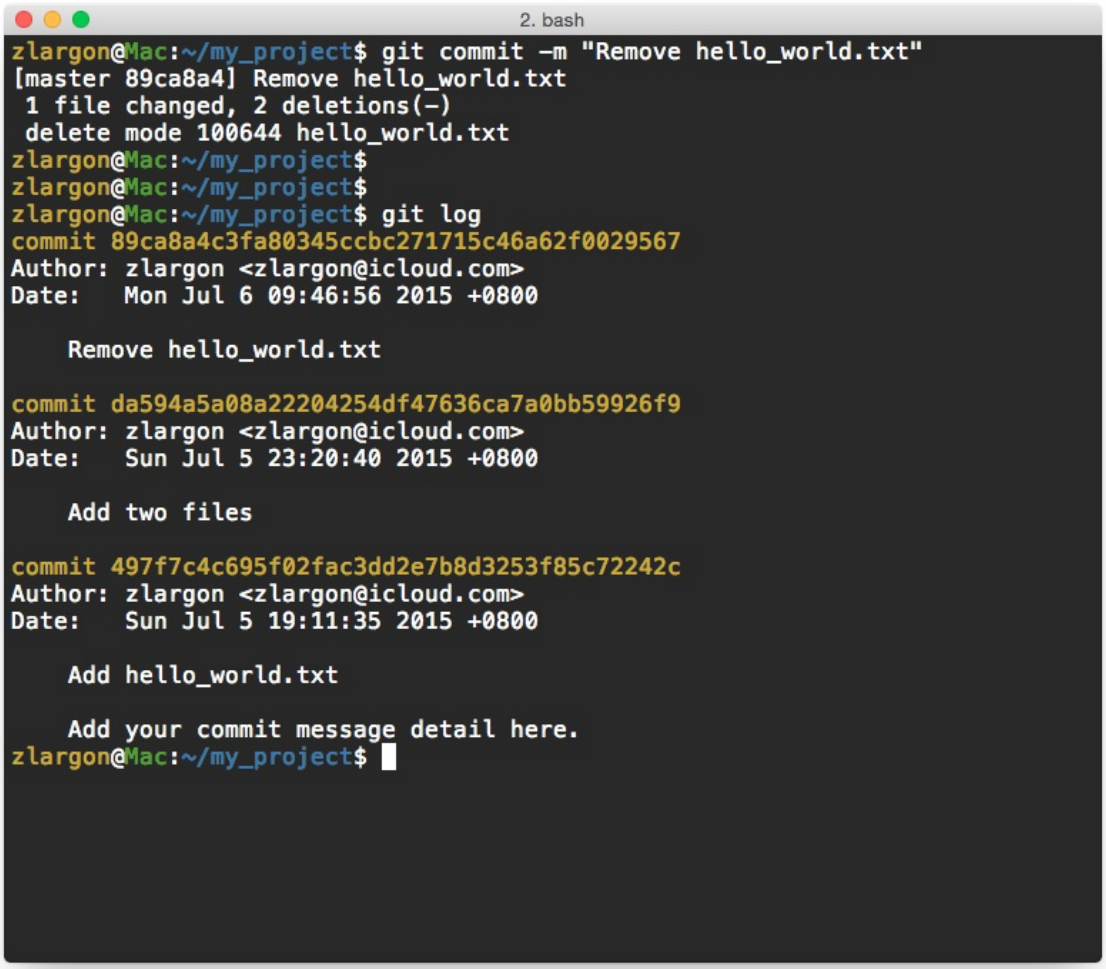
不過可以使用參數 `-u` 或是 `--update` 一次加入所有被更動的檔案，其包含的 ***modified*** 及 ***deleted*** 檔案

由於這裡只有一個 **deleted** 的檔案，所以 `git add -u` 效果跟 `git rm hello_world.txt` 會是一模一樣的

```
$ git add -u        # 一次加入所有被更動的檔案，包含 modified 及 deleted
$ git add --update  # 同上
```

提交 patch，並用 `git log` 來查看提交歷史紀錄

```
$ git commit -m "Remove hello_world.txt"
$ git log
```



本章回顧

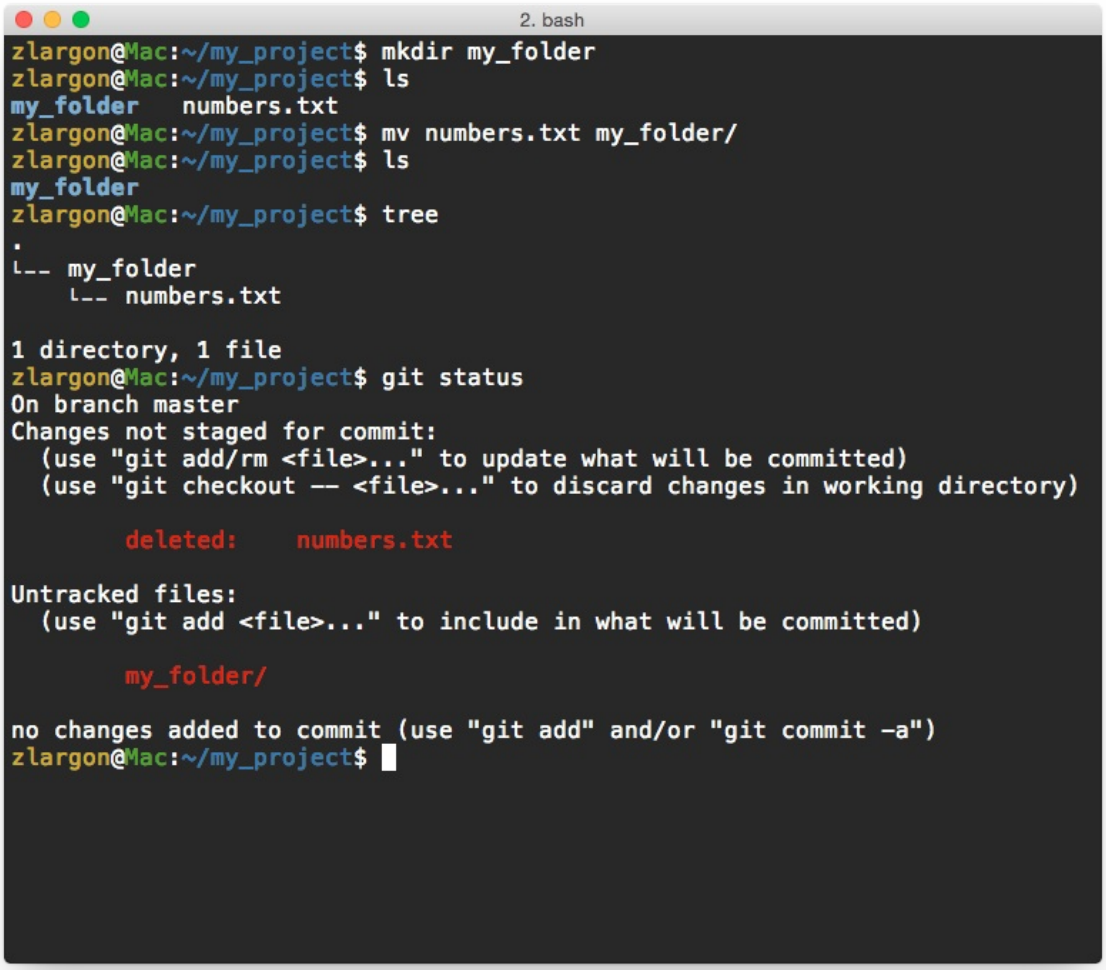
- 使用 `git rm <file>` 來告知 git，哪些是我們將要刪除的檔案
- 使用 `git add -u` 加入所有被更動的檔案（包含 *modified* 及 *deleted*）

搬移檔案

我們先新增一個資料夾 `my_folder`

然後再把 `numbers.txt` 移到 `my_folder/` 底下，並用 `git status` 觀察檔案的狀態

```
$ mkdir my_folder
$ mv numbers.txt my_folder/
$ git status
```



目前檔案結構爲

```
.
└── my_folder/
    └── numbers.txt
```

我們會看到原本 `numbers.txt` 的狀態變成 ***Changes not staged for commit (deleted)***，而 ***Untracked files*** 區塊多一個 `my_folder/` 的資料夾

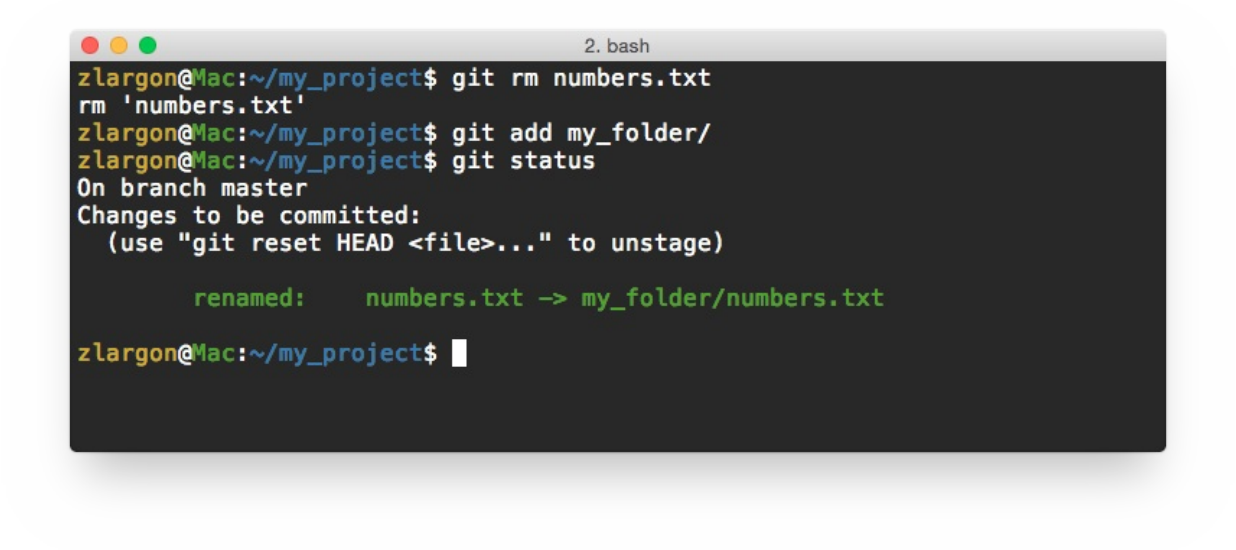
對 git 而言，可以看成是：

- 把原本的 `numbers.txt` 檔案刪除
- 在 `my_folder/` 底下，新增一個一模一樣的檔案 `numbers.txt`

現在我們用 `git add/rm` 來加入新增 / 刪除的檔案，並用 `git status` 查看目前的狀態

> 這裡也可以用 `git add -A` 一次加入全部的檔案

```
$ git rm numbers.txt
$ git add my_folder/
$ git status
```



`numbers.txt` 的狀態變成 ***Changes to be committed (renamed)***

我們可以看出搬移 `numbers.txt` 檔案的操作，對 git 而言，也可以看成是一種 "重新命名" 的概念

使用 `git mv <file> <directory>` 來搬移檔案

以上的搬移步驟，其實可以簡化成兩個步驟

- 新增資料夾 `my_folder/`
- 告訴 git，我要將 `numbers.txt` 移到資料夾 `my_folder/` 底下

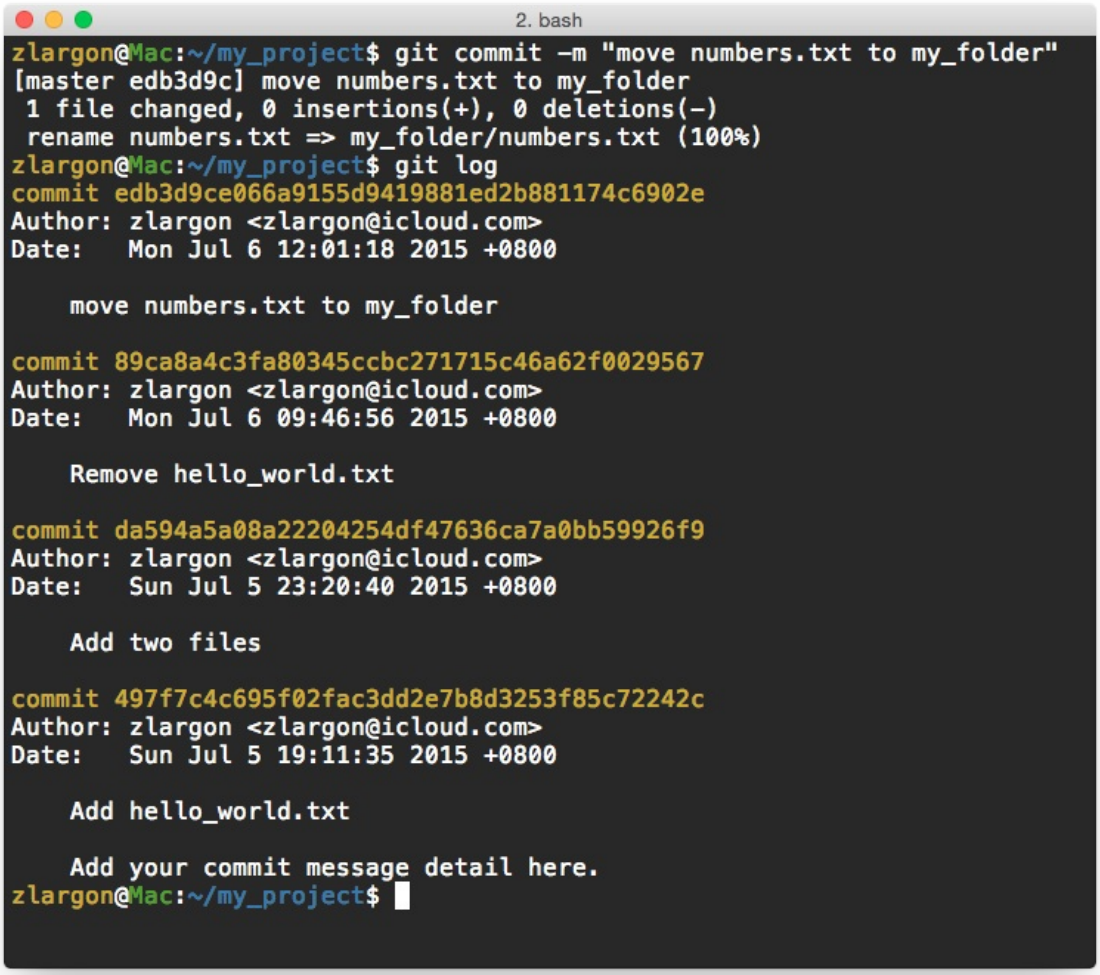
```
$ mkdir my_folder
$ git mv numbers.txt my_folder/
```

這樣就不用再多做 git add/rm 的動作了

這裡 `mv` 的意思就是 `move`

提交 patch，並用 `git log` 來查看提交歷史紀錄

```
$ git commit -m "move numbers.txt to my_folder"
$ git log
```



重新命名檔案

在前一個章節有提到，搬移檔案的操作對 git 而言，也可以看成是一種 "重新命名" 的概念

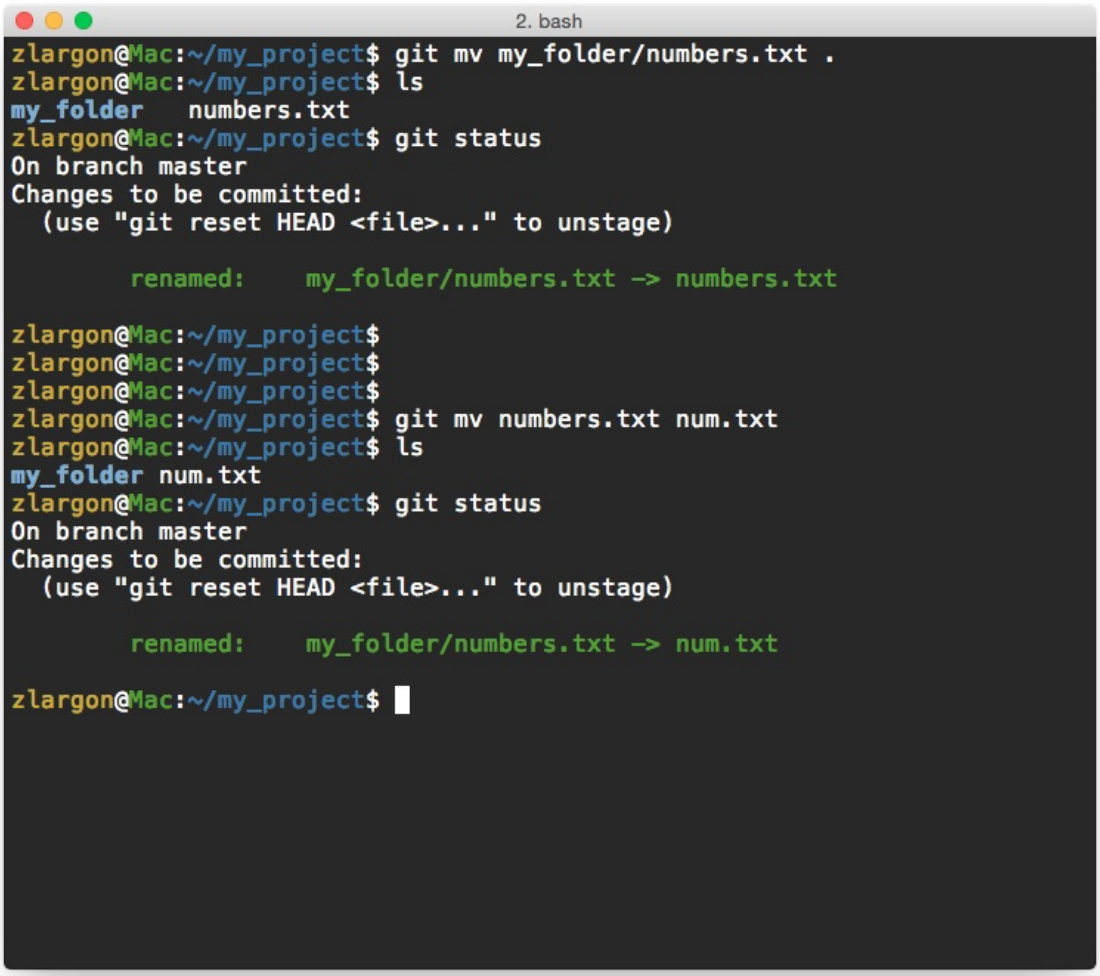
> 事實上，並不只有是 git，`Unix` 上的 `mv` 也是把 "搬移" 跟 "重新命名" 視爲相同的概念

使用 `git mv <file> <new_name>` 來重新命名檔案

	Command Format	Introduction
Move	`git mv <file> <directory>`	把檔案移到某個資料夾底下
Rename	`git mv <file> <new_name>`	把檔案重新命名爲 ... （移到 "同一個資料夾" 底下成爲 xxx 檔案）

我們現在將剛才 `my_folder` 底下的 `numbers.txt` 移回原目錄底下，並且重新命名成 `num.txt`

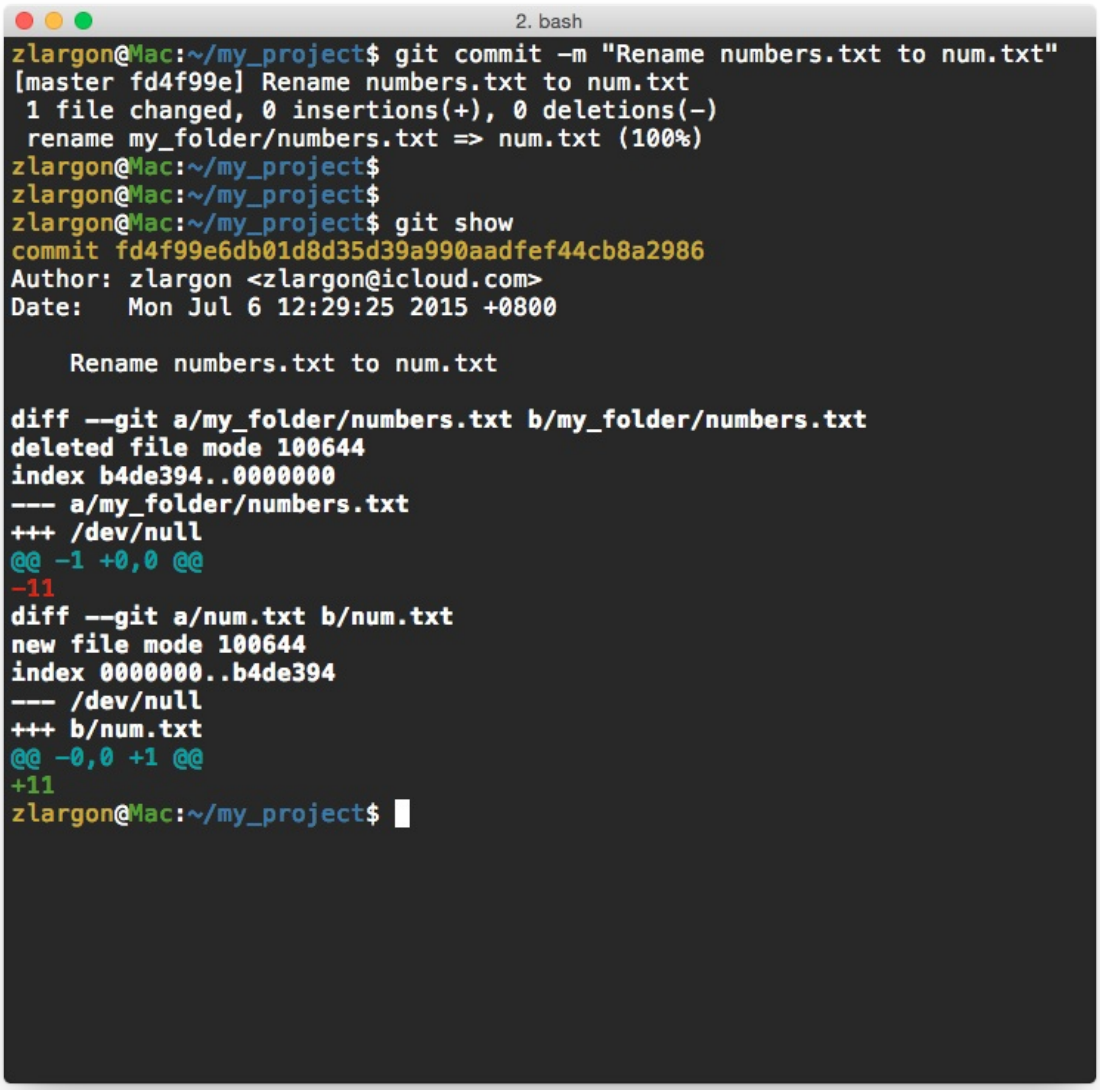
```
$ git mv my_folder/numbers.txt .
$ git mv numbers.txt num.txt
```



我們透過 `ls` 可以看出，`numbers.txt` 確實已經被重新命名爲 `num.txt` 了

提交 patch，並用 `git show` 來查看這次提交所修改的內容

```
$ git commit -m "Rename numbers.txt to num.txt"
$ git show
```

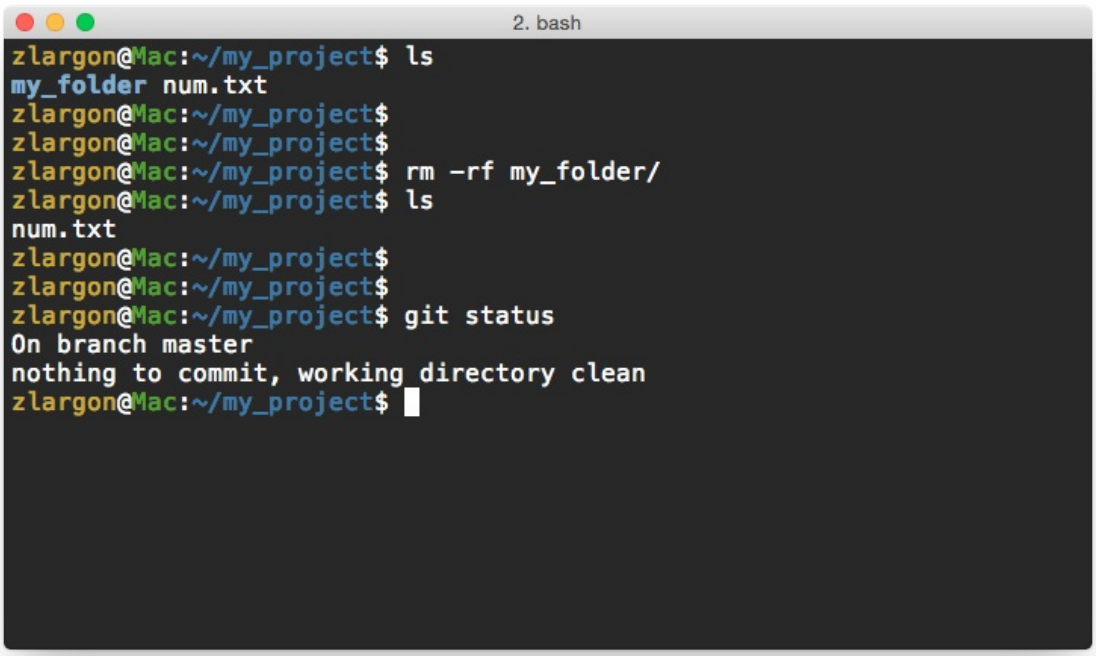


git 會自動忽略空資料夾

現在我們透過 `ls` 可以看出，現在目錄下有一個 `my_folder/` 的空資料夾，與一個 `num.txt` 的檔案

既然 `my_folder/` 已經用不到了，那麼我們就把他移除，並用 `git status` 來查看狀態

```
$ rm -rf my_folder/
$ git status
```



雖然我們已經把 `my_folder` 刪掉了，可是 `git status` 卻沒有發生任何的狀態改變？

這是因爲這些空資料夾對於整個 project 來說，本身可有可無

因此 git 認爲，既然資料夾裡面沒有任何東西，那麼就沒有必要去在意他的存在

git 唯一在意的是檔案的內容

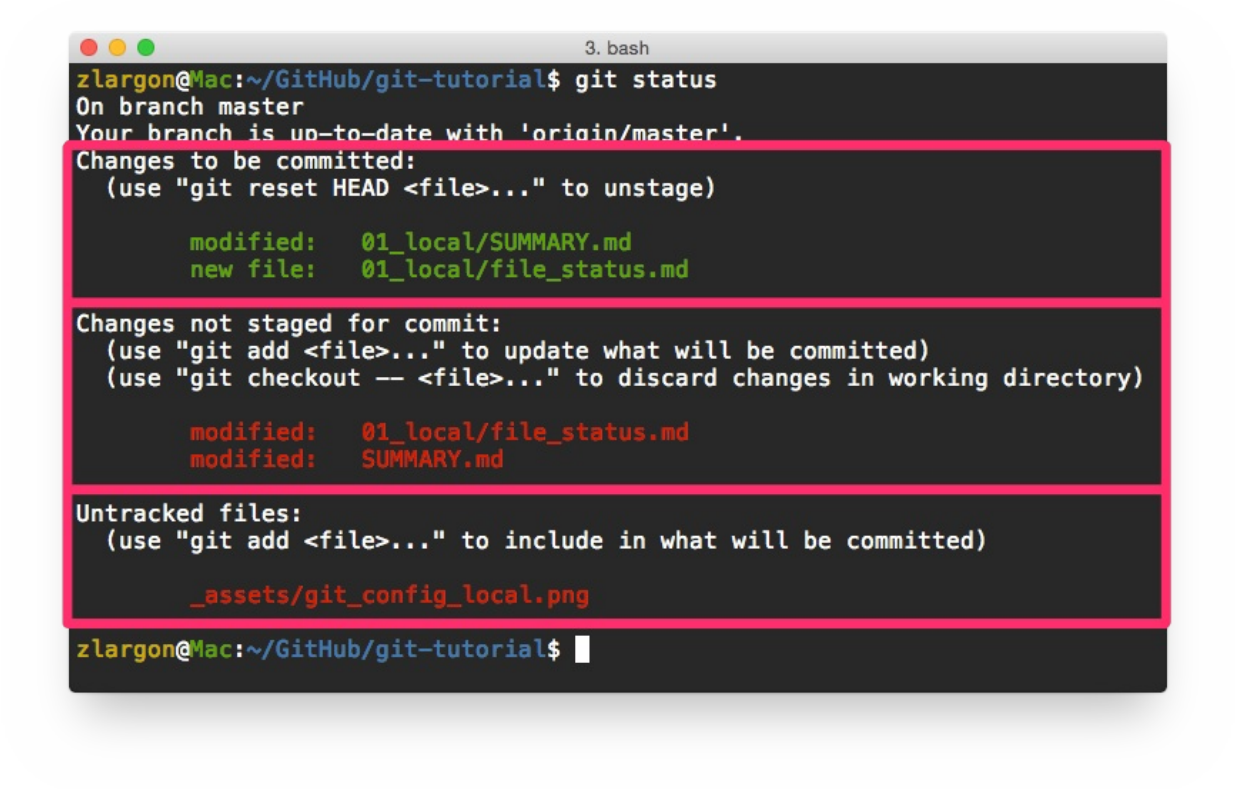
而資料夾本身並不是一個檔案

檔案狀態

在前面幾個章節，介紹完 git 的基本指令之後，我們現在要來說明 git 的檔案狀態

Git 將 "尚未被提交" 的檔案分成三個區塊，由上而下分別是

- **Changes to be committed**（將要提交的檔案）
- **Changes not staged for commit**（被更動但尚未要提交的檔案）
- **Untracked files**（未被追蹤的檔案）



Untracked files（未被追蹤的檔案）

意指完全新加入的檔案，不曾被提交過

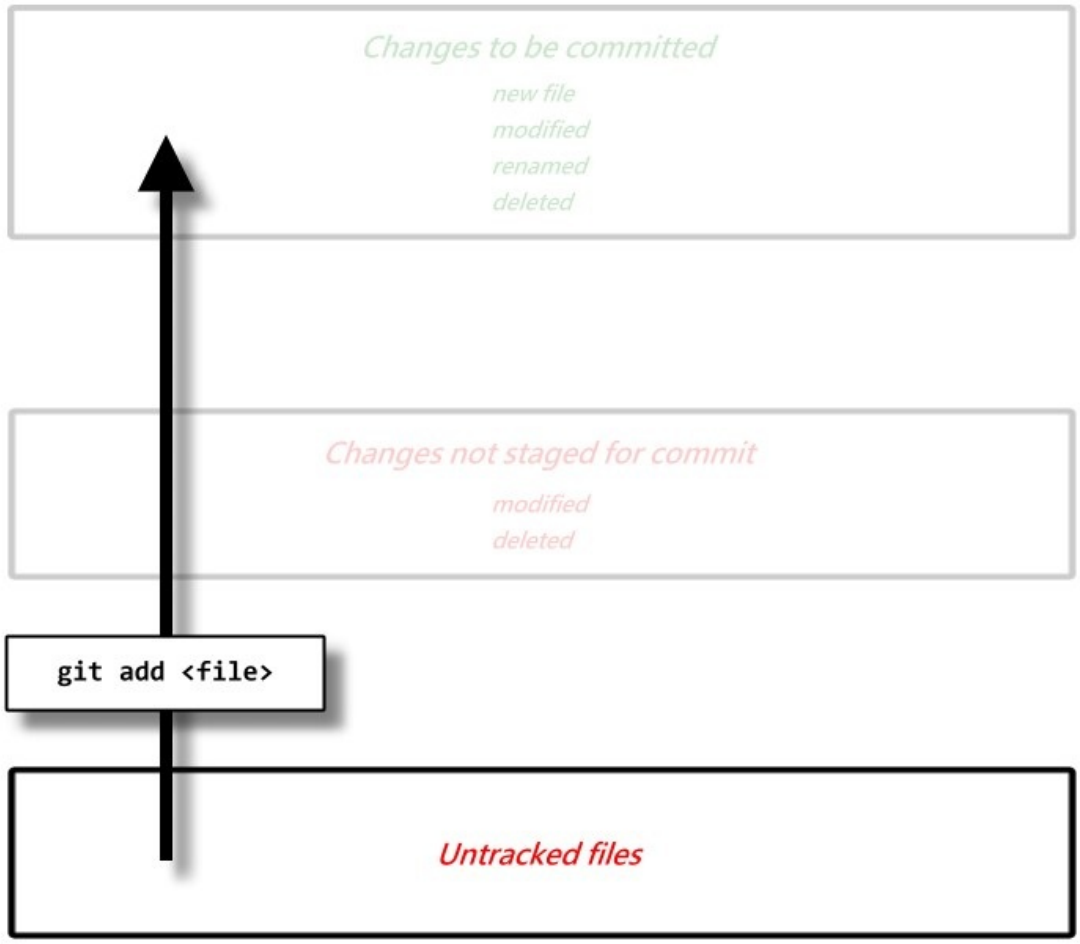
由於 git 只會去追蹤（track）被提交過（committed）的檔案的被修改狀態，而不會去追蹤新進的檔案

因此這類型屬於 ***Untracked files***

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

        _assets/git_config_local.png
```

- ***use "git add <file>..." to include in what will be committed***

  使用 `git add <file>` 的指令，使檔案狀態改爲 ***Changes to be committed***



Changes not staged for commit（被更動但尚未要提交的檔案）

這一類的檔案，都是先前就已經被提交過的檔案

git 會去 track 這些檔案的狀態，當檔案被修改（***modified***）或是刪除（***deleted***）的時候，就會出現在這裡

> 這裡所有的檔案（***modified & deleted***）可以使用 `git add -u` 一次全部轉到綠色區塊

```
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:   01_local/file_status.md
        modified:   SUMMARY.md
```

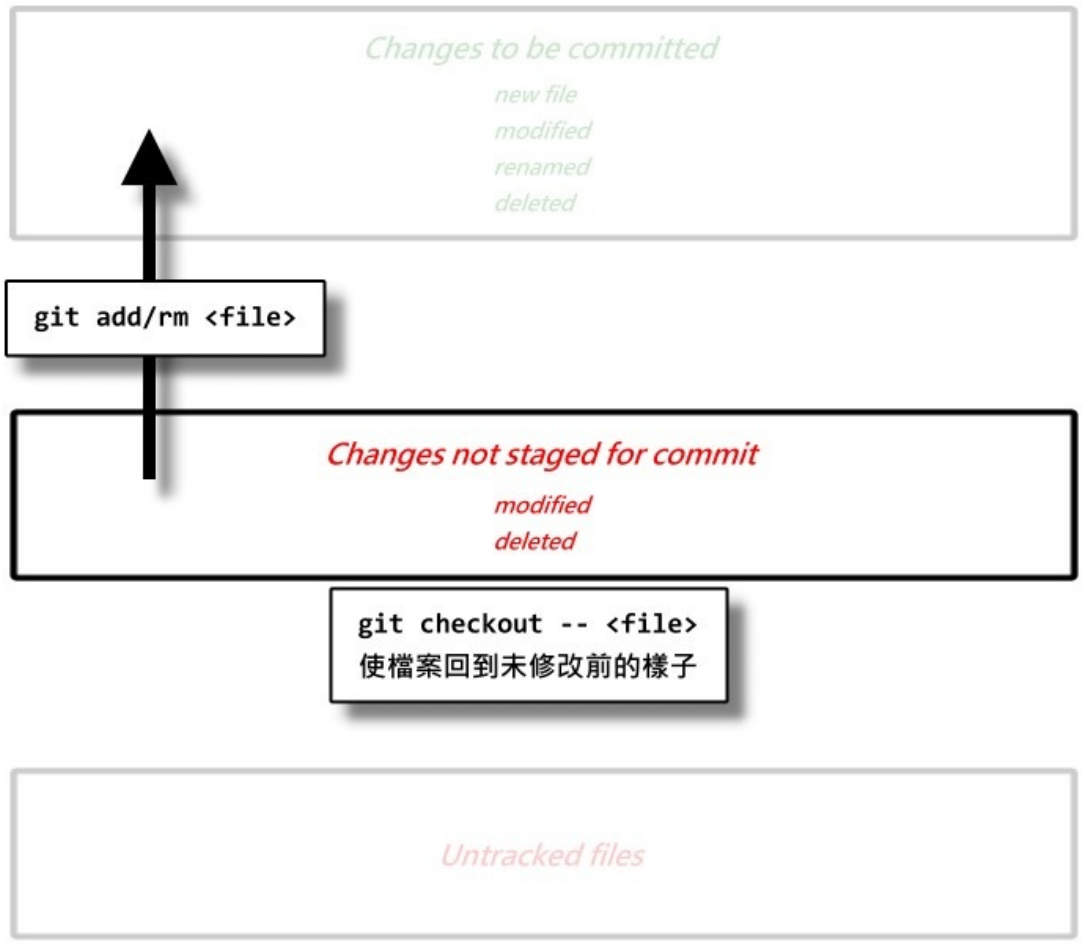
- ***use "git add <file>..." to update what will be committed***

  使用 `git add <file>` 的指令，使檔案狀態改爲 ***Changes to be committed***

  > 若狀態爲 ***deleted*** 的檔案，必須要用 `git rm <file>` 才能轉到 ***Changes to be committed*** 區塊
  >
  > 某些較新版本的 git 才可以用 `git add` 加入 "被刪除" 的檔案

- ***use "git checkout -- <file>..." to discard changes in working directory***

  使用 `git checkout -- <file>` 的指令，使檔案回到 "未修改" 前的樣子



Changes to be committed（將要提交的檔案）

這些是我們透過 `git add, rm, mv` 轉移過來的檔案

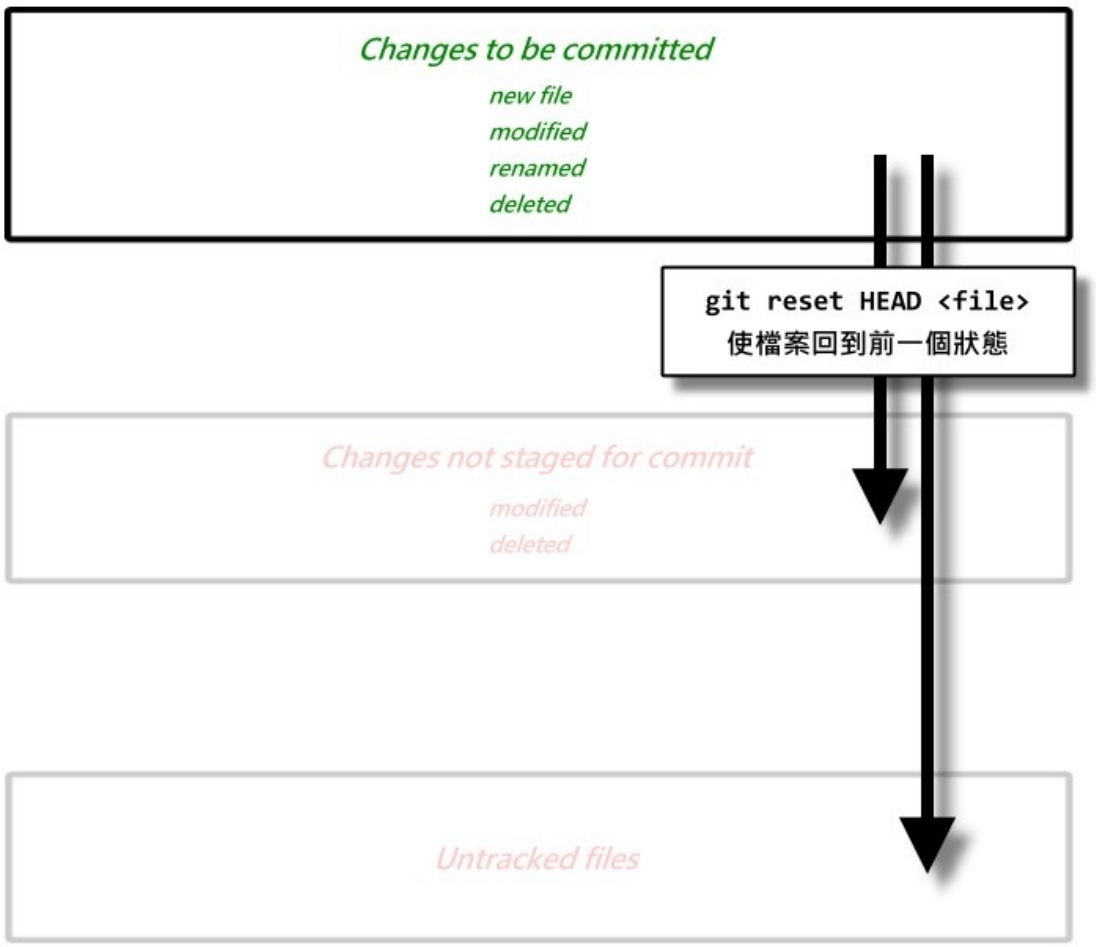
在執行 `git commit` 之後，將會被一起提交到一個 patch 裡面

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   01_local/SUMMARY.md
        new file:   01_local/file_status.md
```

- ***use "git reset HEAD <file>..." to unstage***

  使用 `git reset HEAD <file>` 可以將檔案還原到 **"未準備提交"** 前的狀態

  > 若使用 `git reset HEAD` 而後面不帶 `<file>` 的話，會將這個區塊 "所有" 的檔案都一併還原到 **"未準備提交"** 前的狀態
  >
  > `HEAD` 是 git 特殊的關鍵字，用來表示你目前所在的 patch 的位置（我們後面會在做詳細的說明）

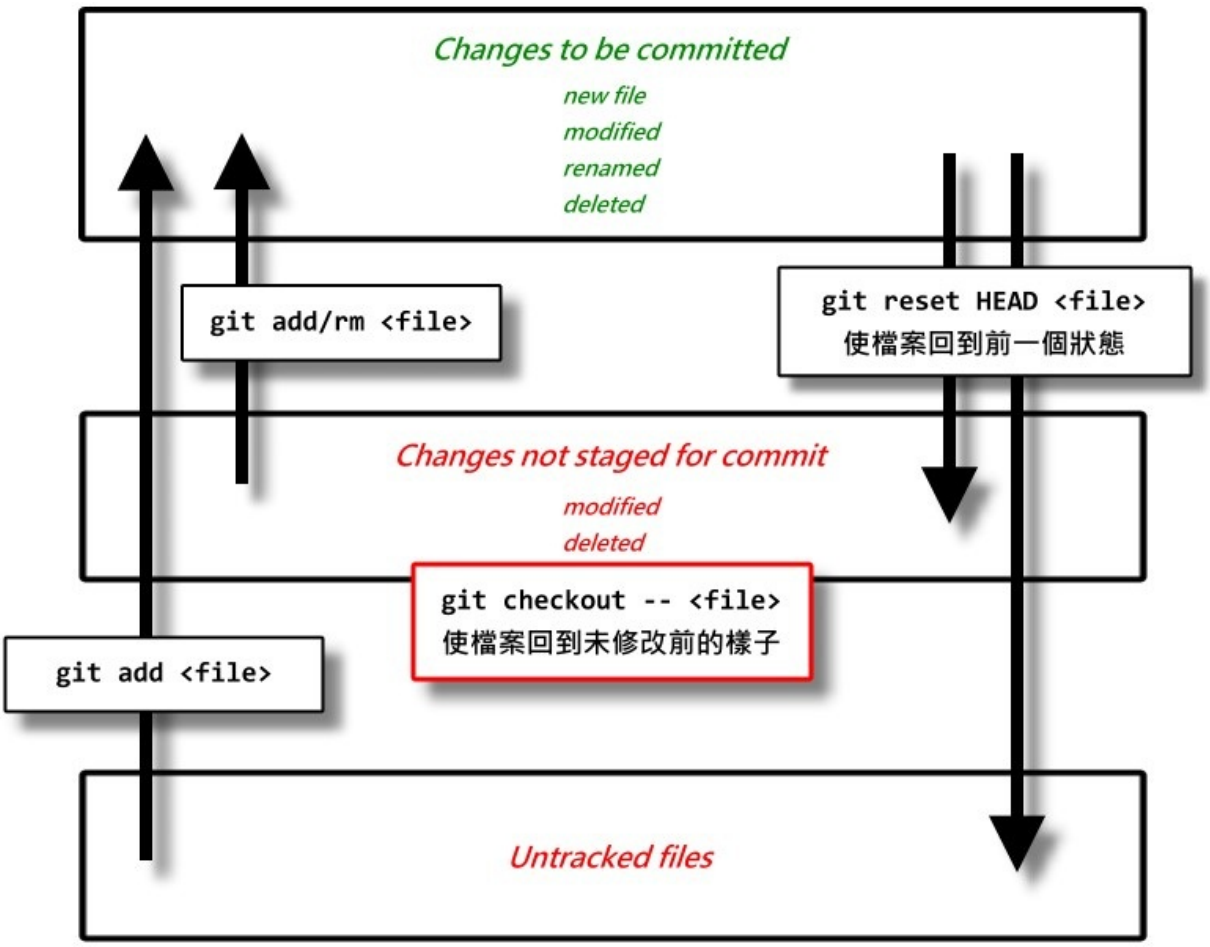


本章回顧

這些狀態切換的指令都不需要去記他

因爲在做 `git status` 的時候，每個區塊都會有指令提示

只要心裡對這個圖有個概念就好了



> Git 官方網站有關於檔案狀態的介紹 [英文版] [中文版]
>
> 我覺得看 `File Status Lifecycle` 的圖並不是很好理解，不過大家還是可以參考一下

檔案還原

在之前幾節裡面，都是用 `git add / rm` 來使狀態變成 ***Changes to be committed***

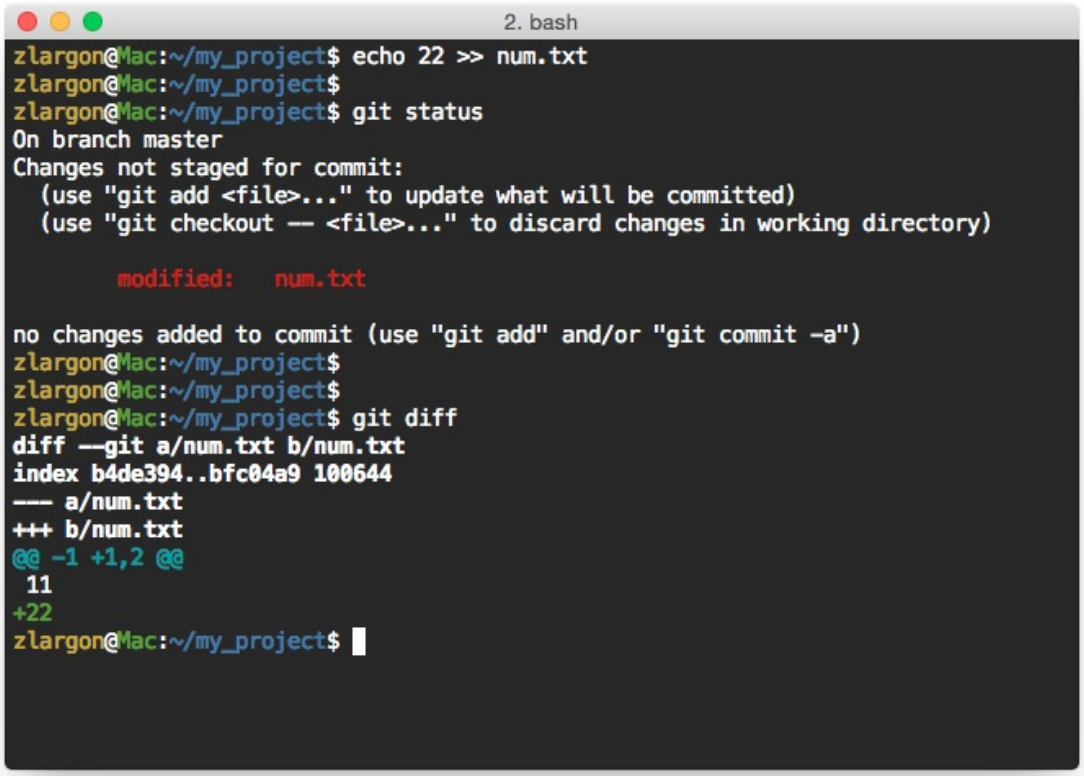
在我們了解 "檔案狀態" 間的轉換之後

現在我們來示範一下檔案還原的部分

使用 `git checkout -- <file>` 來還原 "檔案內容"

首先我們先來修改檔案 `num.txt` ，在後面新增一行字串 "22"，然後檢視其狀態與改變的內容

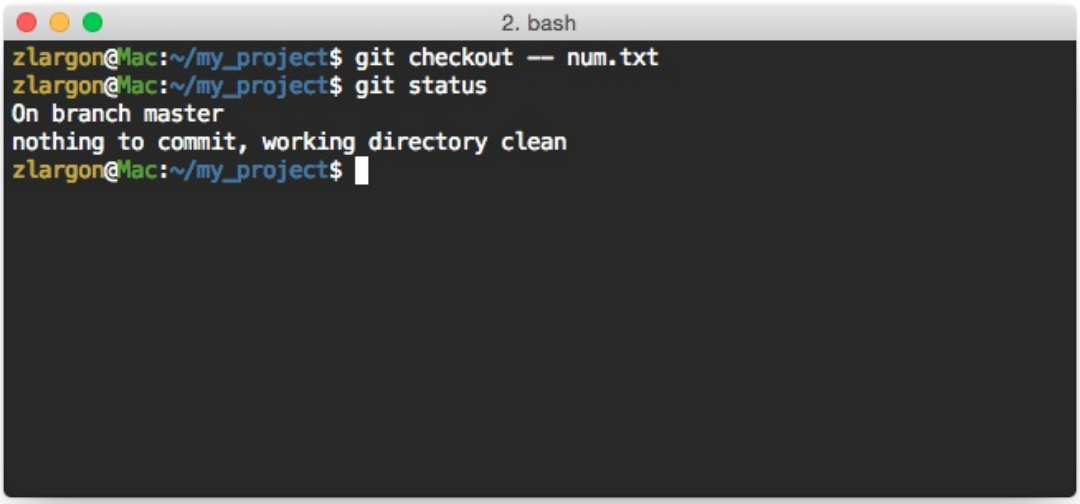
```
$ git status
$ git diff
```



這時候 `num.txt` 的狀態是 ***Changes not staged for commit (modified)***

接著我們可以透過 `git checkout -- <file>` 來還原檔案的內容

```
$ git checkout -- num.txt
$ git status
```

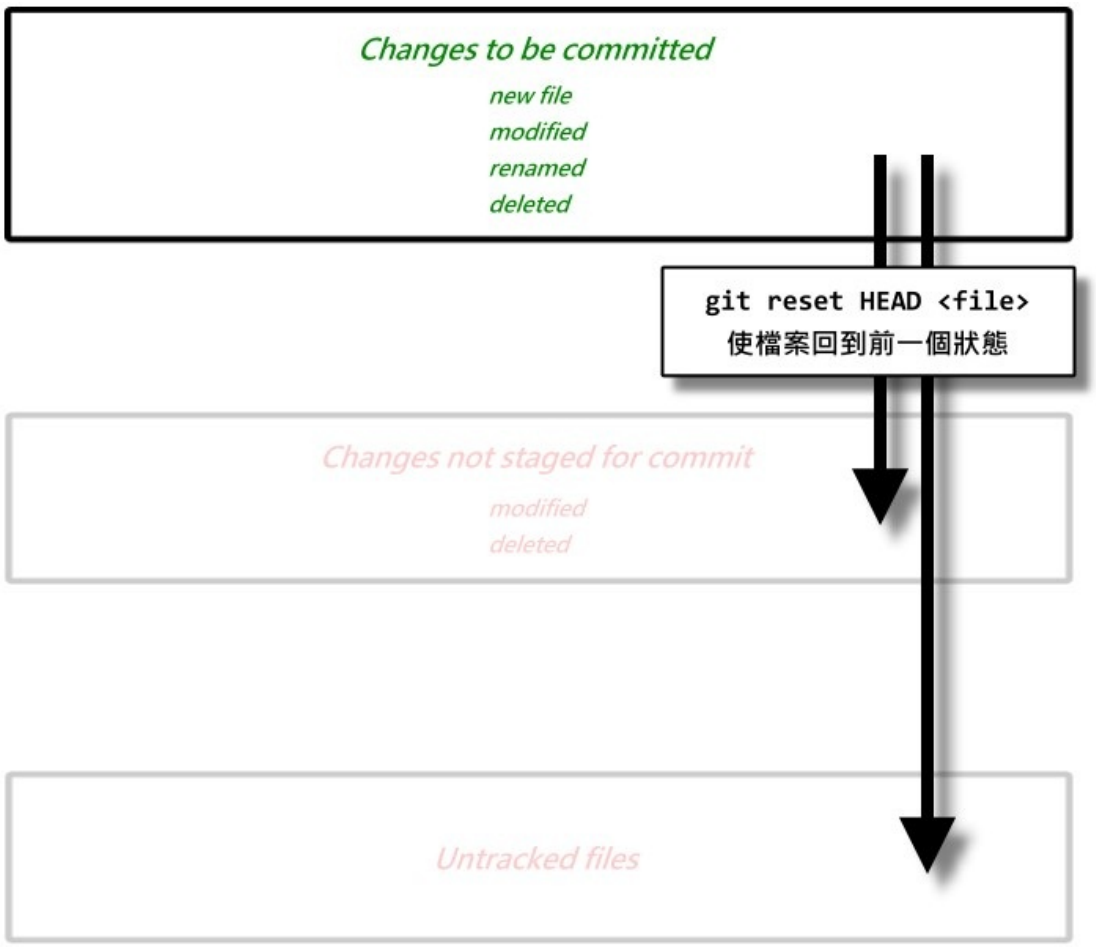


這裡可以看到，檔案已經還原修改前的內容

不論檔案狀態是 ***modified*** 或是 ***deleted*** 都可以用 `git checkout -- <file>` 來還原檔案的內容

使用 `git reset HEAD <file>` 來還原 "檔案狀態"

簡單來說，這是一個把綠色變回紅色的指令



根據 `HEAD`（目前的 patch，也就是最後一次提交的 patch）來還原 `file` 的 "檔案狀態"

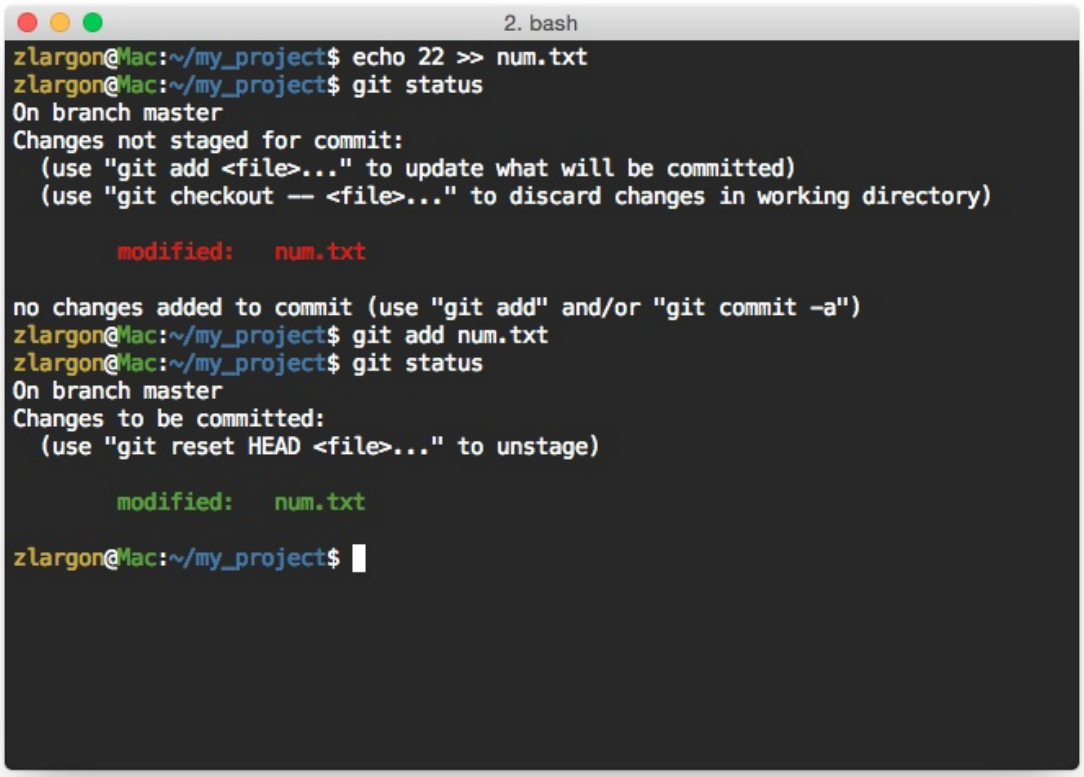
他只會還原檔案 "狀態"，而不會還原檔案的 "內容"

若後面不加 `file` 來指定檔案的話，就表示要還原全部的檔案

```
$ git reset HEAD <file>     # 還原 "指定" 的檔案狀態
$ git reset HEAD            # 還原 "全部" 的檔案狀態
```

我們現在一樣在檔案 `num.txt` 後面新增一行字串 "22"，並且使用 `git add` 來告知 git 我們將要提交這個檔案

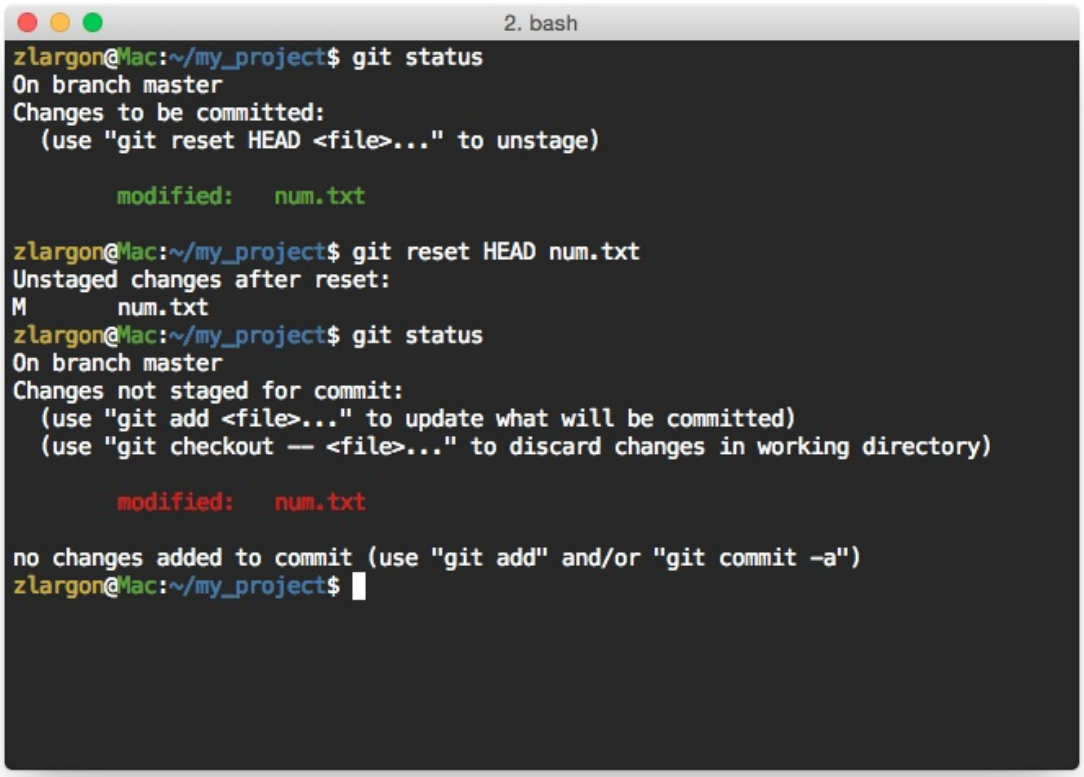
```
$ git status
$ git add
$ git status
```



這時候 `num.txt` 的狀態，從 ***Changes not staged for commit (modified)*** 變成 ***Changes to be committed (modified)***

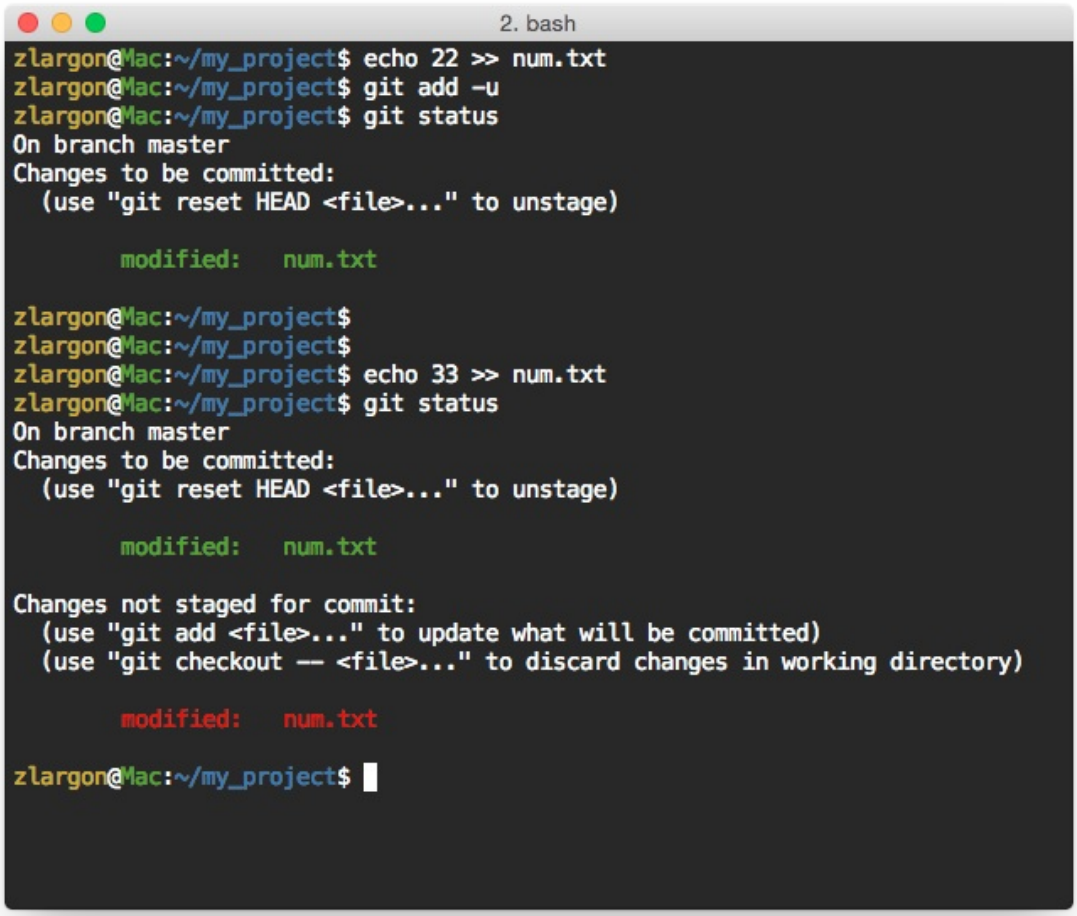
我們可以使用 `git reset HEAD <file>` 使他回到原本的狀態

```
$ git status
$ git reset HEAD num.txt
$ git status
```



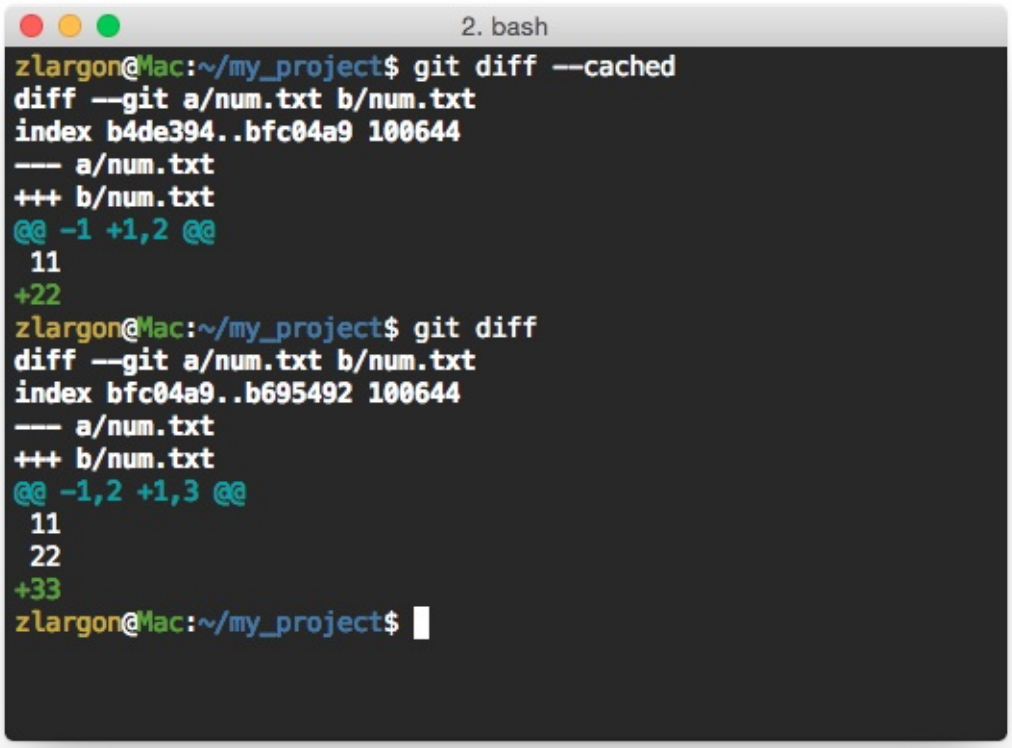
一個檔案可同時爲 *Changes not staged for commit* 及 *Changes to be committed*

首先我們先新增一行 "22"，然後 `git add` 到 ***Changes to be committed***，然後再對 `num.txt` 新增一行 "33"



這時候我們用 `git status` 可以看到 `num.txt` 同時屬於兩種狀態

這時候我們再用 `git diff` 來查看改變的內容



從 `git diff --cached` 可以看到新增 "22" 的部分

從 `git diff` 可以看到新增 "33" 的部分

git 可以只 ***commit*** 一份檔案中部分的內容，不一定要以整份檔案爲單位來提交

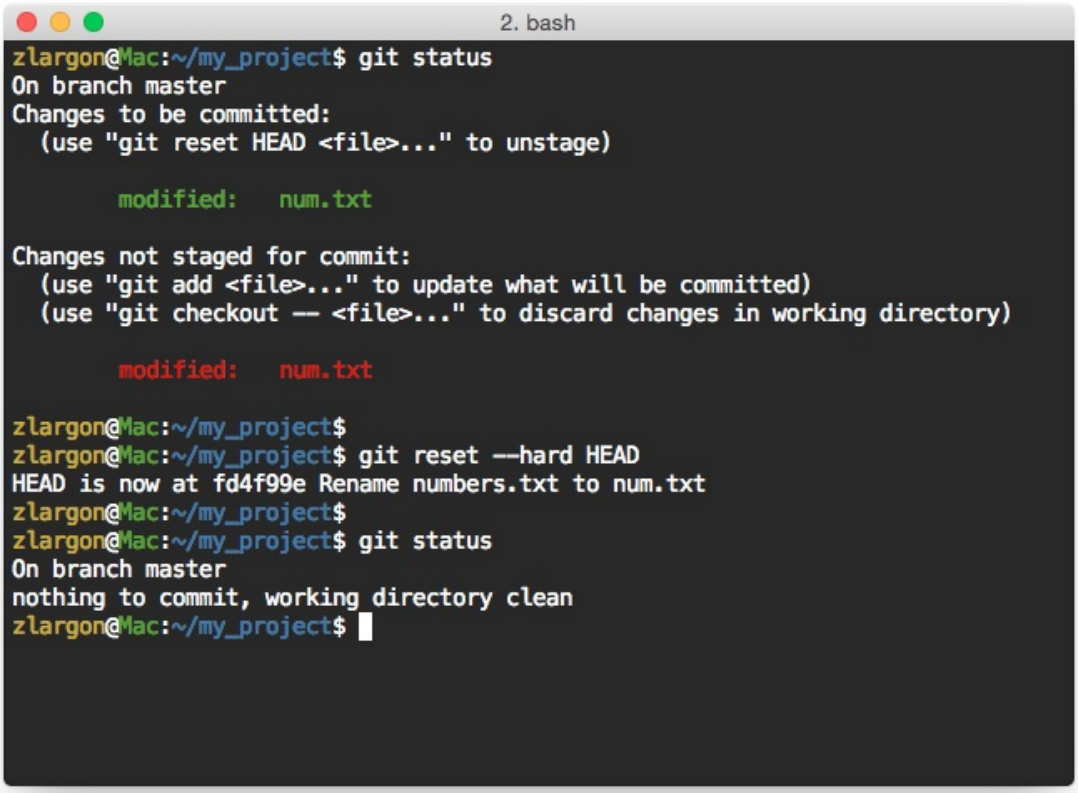
使用 `git reset --hard HEAD` 一次還原所有的檔案內容

`git reset HEAD` 可以用來還原 ***Changes to be committed*** 所有的檔案狀態

但是如果要還原檔案內容的話，必須對所有檔案再做一次 `git checkout -- <file>`

而 `git reset --hard HEAD` 可以一次將 ***Changes not staged for commit*** 和 ***Changes to be committed*** 的區域清空

```
$ git status
$ git reset --hard HEAD
$ git status
```



我們可以看到，所有的在非 ***Untracked files*** 的內容，都被清空了

由於 git 不會去追蹤 ***Untracked files*** 的內容，因此這裡的檔案只能手動刪除

不過還是可以透過以下方式把他們刪除

```
$ git add -A                # 把所有檔案加到 Changes to be committed
$ git reset --hard HEAD     # 一次還原所有檔案的內容
```

我個人平時還滿常會這樣用的

本章回顧

- 使用 `git checkout -- <file>` 來還原 "檔案內容"
- 使用 `git reset HEAD <file>` 來還原 "檔案狀態"
- 使用 `git reset --hard HEAD` 來清空 ***Changes not staged for commit*** 和 ***Changes to be committed*** 區塊

忽略檔案

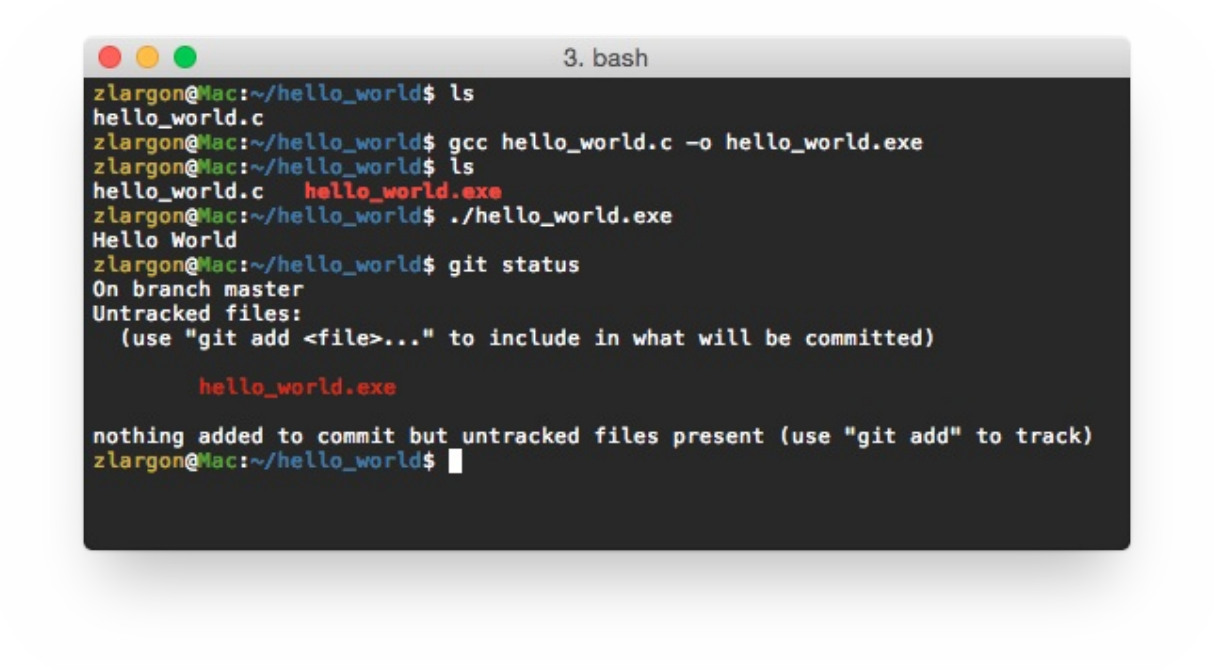
有一些檔案是我們在開發的時候 build 出來的，他可能是 binary 的檔案

通常我們不會提交這些檔案，因爲我們每次 build 出來，binary 的內容都不盡相同

我們不會直接去編輯修改這些檔案，他們從 `git diff` 上看，也只是看到 binary 不同而已

提交這些檔案對開發來說沒有意義的

舉例來說，C 程式碼可以透過 `Makefile` 或是 `gcc` 編譯出可執行檔，我們通常不會提交這些可執行檔



這些 build 出來的檔案會一直出現在 ***Untracked files***

而且這種檔案一多起來，每次做 `git status` 的時候，就會看起來很繁雜

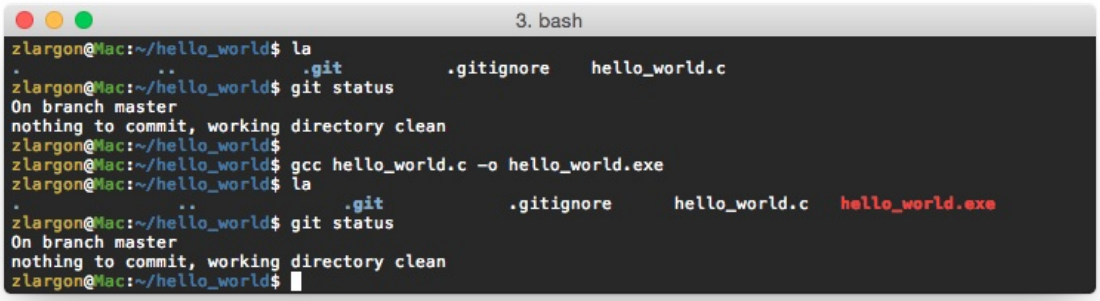
因此，git 提供爲了一個方式讓我們避免這種狀況

那就是在目錄下新增一個名爲 `.gitignore` 的檔案

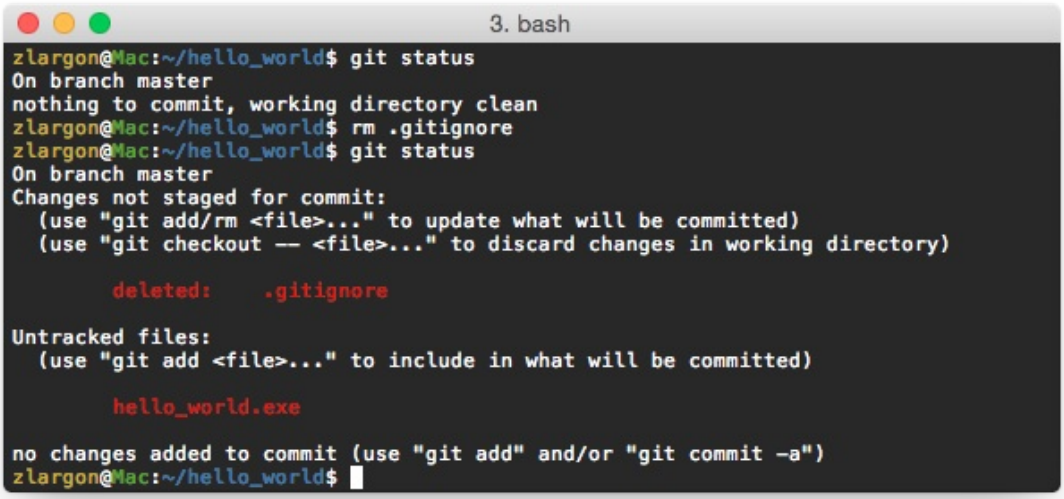
只要把你想忽略的檔案，通通寫在這裡就可以了

.gitignore

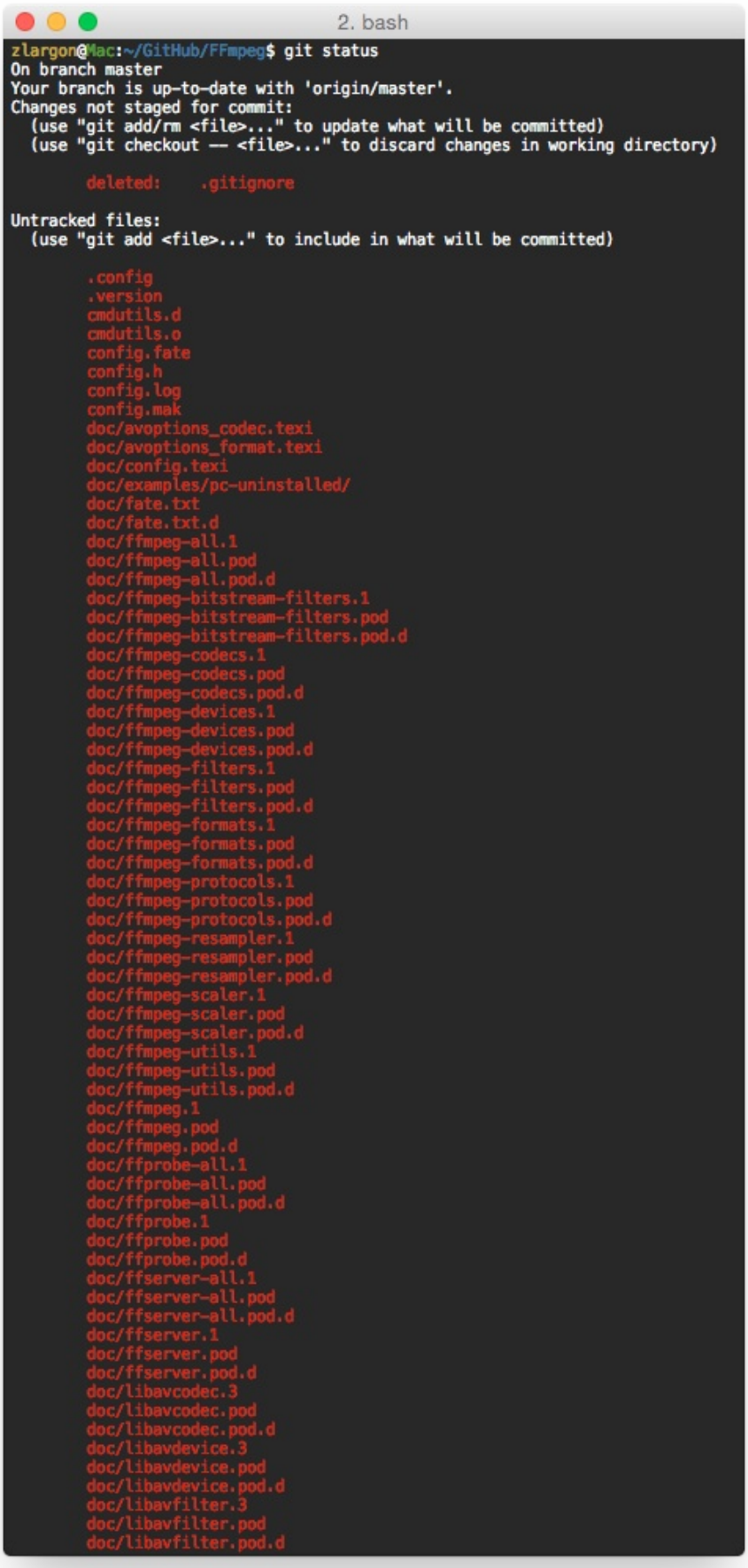
```
*.exe        # 忽略所有 xxx.exe 的檔案
```



若我們將 `.gitignore` 刪除，那麼這些檔案就會重回到 ***Untracked files***



若以 FFmepg 爲例，我們把 `.gitignore` 刪除，就會看到一大堆的 .o 或是 .d 檔案


2. bash
zlargon@Mac:~/GitHub/FFmpeg$ git status
On branch master
Your branch is up-to-date with 'origin/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: .gitignore
Untracked files:
  (use "git add <file>..." to include in what will be committed)
.config
.version
cmdutils.d
cmdutils.o
config.fate
config.h
config.log
config.mak
doc/avoptions_codec.texi
doc/avoptions_format.texi
doc/config.texi
doc/examples/pc-uninstalled/
doc/fate.txt
doc/fate.txt.d
doc/ffmpeg-all.1
doc/ffmpeg-all.pod
doc/ffmpeg-all.pod.d
doc/ffmpeg-bitstream-filters.1
doc/ffmpeg-bitstream-filters.pod
doc/ffmpeg-bitstream-filters.pod.d
doc/ffmpeg-codecs.1
doc/ffmpeg-codecs.pod
doc/ffmpeg-codecs.pod.d
doc/ffmpeg-devices.1
doc/ffmpeg-devices.pod
doc/ffmpeg-devices.pod.d
doc/ffmpeg-filters.1
doc/ffmpeg-filters.pod
doc/ffmpeg-filters.pod.d
doc/ffmpeg-formats.1
doc/ffmpeg-formats.pod
doc/ffmpeg-formats.pod.d
doc/ffmpeg-protocols.1
doc/ffmpeg-protocols.pod
doc/ffmpeg-protocols.pod.d
doc/ffmpeg-resampler.1
doc/ffmpeg-resampler.pod
doc/ffmpeg-resampler.pod.d
doc/ffmpeg-scaler.1
doc/ffmpeg-scaler.pod
doc/ffmpeg-scaler.pod.d
doc/ffmpeg-utils.1
doc/ffmpeg-utils.pod
doc/ffmpeg-utils.pod.d
doc/ffmpeg.1
doc/ffmpeg.pod
doc/ffmpeg.pod.d
doc/ffprobe-all.1
doc/ffprobe-all.pod
doc/ffprobe-all.pod.d
doc/ffprobe.1
doc/ffprobe.pod
doc/ffprobe.pod.d
doc/ffserver-all.1
doc/ffserver-all.pod
doc/ffserver-all.pod.d
doc/ffserver.1
doc/ffserver.pod
doc/ffserver.pod.d
doc/libavcodec.3
doc/libavcodec.pod
doc/libavcodec.pod.d
doc/libavdevice.3
doc/libavdevice.pod
doc/libavdevice.pod.d
doc/libavfilter.3
doc/libavfilter.pod
doc/libavfilter.pod.d

.gitignore 範例

如果不清楚該把哪些檔案加到 `.gitignore` 的話，可以參考這個專案，他有大部份的專案類型所用的 `.gitignore` 範例

https://github.com/github/gitignore

.gitignore 作用範圍

`.gitignore` 作用範圍包含整個資料夾以及其所有子資料夾

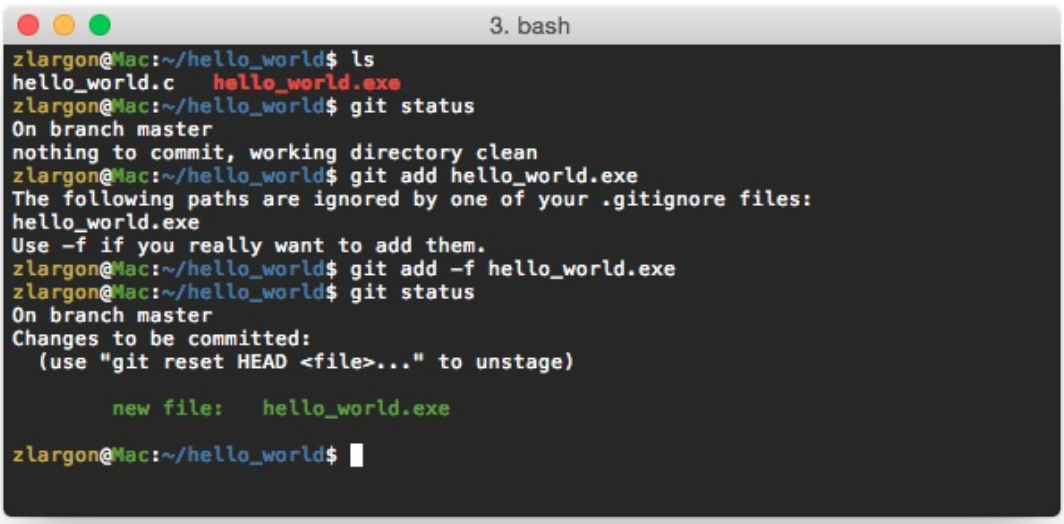
`.gitignore` 也可以存在多個資料夾中

每個資料夾都可以另外定義 `.gitignore` 的內容

使用 `git add -f <file>` 強制 add 被忽略的檔案

若有一些情況，我們必須要提交這些被 git 忽略的檔案，就可以使用這個 `-f` 參數，強制加入檔案

`-f` 同等於 `--force`，表示強制的意思



Patch 管理

基本觀念

```
git log --pretty=raw
```

關鍵字 HEAD

```
git show HEAD^
git show HEAD~3
```

Reset Patch

```
git log --oneline
git reset HEAD^
git reset --hard HEAD^
git reset --hard <commit id>
```

找回消失的 Patch

```
git reflog
git log -g
```

修改 / 訂正 Patch

```
git commit --amend
git commit --amend -m <message>
git reset --soft HEAD^
git reset --soft HEAD@{1}           # 保命技
```

移除單一個 Patch

```
git cherry-pick <commit id>
```

Rebase 互動模式

```
git rebase -i <after this commit>
```

Cherry-Pick 版本衝突

```
git cherry-pick --continue
git cherry-pick --abort
```

Rebase 版本衝突

```
git rebase --continue
git rebase --skip
git rebase --abort
```

Revert Patch

```
git revert <commit id>
git revert --continue
git revert --abort
```

基本觀念

在上一章檔案管理裡面，我們都是對單一個 patch 進行操作

修改完檔案，使用 commit 來提交，並且產生一個新的 patch

在 git 的世界裡「每一個 ***patch*** 其實都只會記錄其所發生 **"**變化**"**，而不是整份完整的 ***code***」

相對於前一個 patch（git 稱之爲 `parent`）所發生的變化

然而，當 git 我們把所有的 patch 一個接著一個全部串起來之後，就可以變成一份完整的 code

我們就先前的例子來說明

使用 `git log --pretty=raw` 來顯示 `parent` 的資訊

```
$ git log --pretty=raw
```

我們這裡可以看出，除了 `p1` 是初始化的 patch 沒有 `parent` 之外

其他的 patch 的 `parent` 都是一個緊接著一個

- `p2` 記錄了 `p1 → p2` 的變化
- `p3` 記錄了 `p2 → p3` 的變化
- `p4` 記錄了 `p3 → p4` 的變化
- `p5` 記錄了 `p4 → p5` 的變化

就剛才所說的「每一個 ***patch*** 其實都只會記錄其所發生 **"變化"**，而不是整份完整的 ***code***」

git 注重的是你 **"**改**"** 了什麼

git 這樣的設計非常聰明，而且有非常多的好處

我們會在後面的章節一一的講清楚說明白

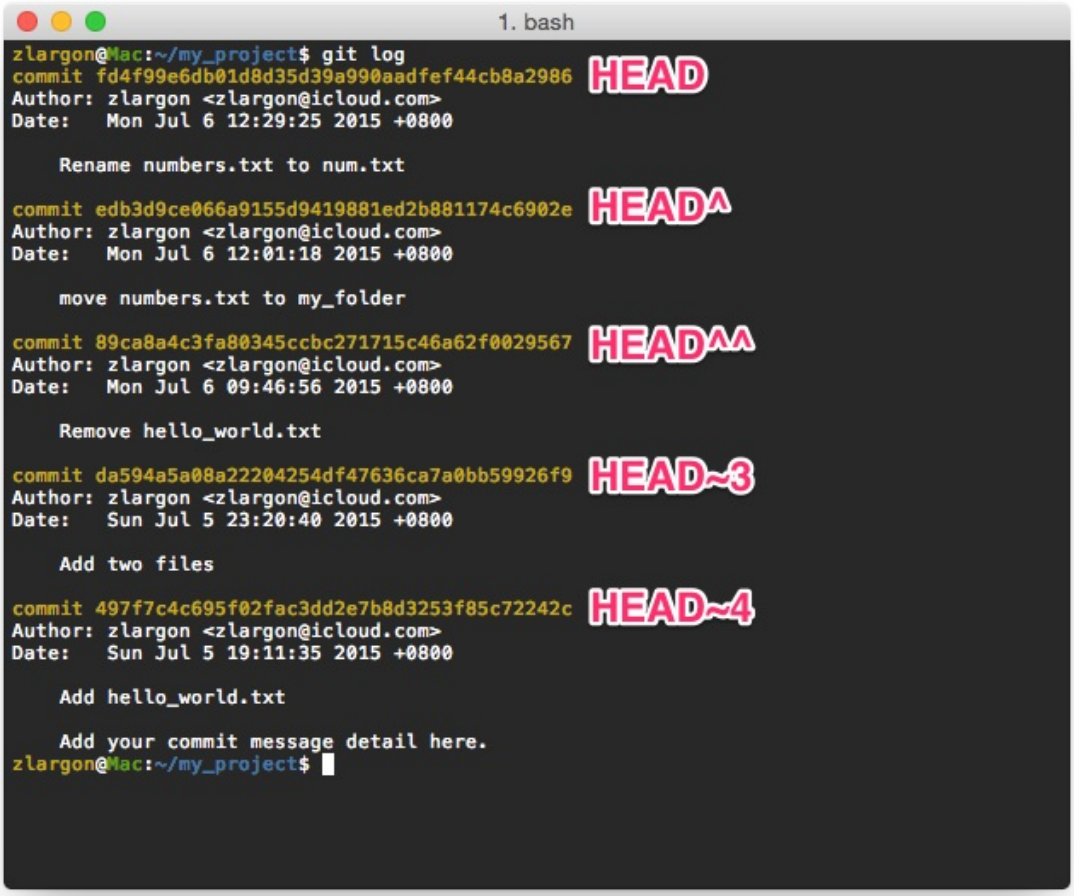
這裡只要有知道 patch 的概念是一個一個串起來的，這樣就行了

關鍵字 HEAD

`HEAD` 爲 git 特殊的關鍵字，意指目前所在 patch 的位置；顧名思義，就是「頭」

我的頭在哪裡，就是我所在的位置

我們可以把 `HEAD` 當作基準點，表示其他相對位置的 patch

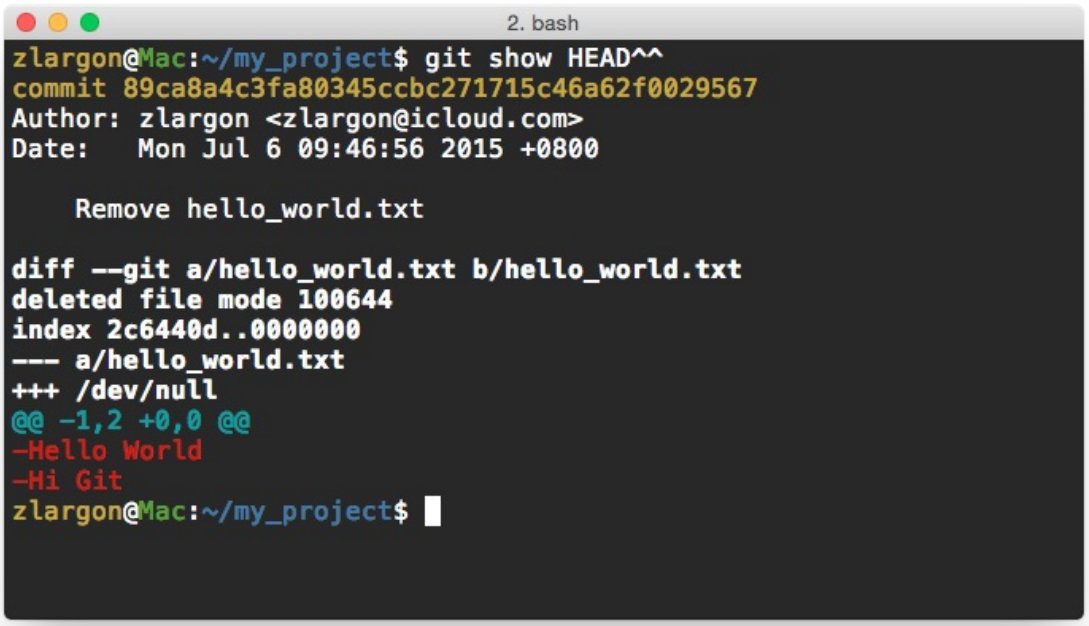


- `HEAD^` 或 `HEAD~1` 可以用來表示上一個 patch
- `HEAD^^` 或 `HEAD~2` 可以用來表示上兩個 patch
- `HEAD^^^` 或 `HEAD~3` 可以用來表示上三個 patch
- 依此類推...

因此，我們之前有教過使用 `git show <commit id>` 來查看之前的 patch 的修改內容

我們也可以改成用 `HEAD` 來查看

例如我要看前兩個的 patch 所修改的內容，就可以用 `git show HEAD^^`



Reset Patch

前面在 "檔案還原" 有提到 `git reset HEAD` 可以根據 `HEAD`（目前所在的 patch）還原全部的檔案狀態

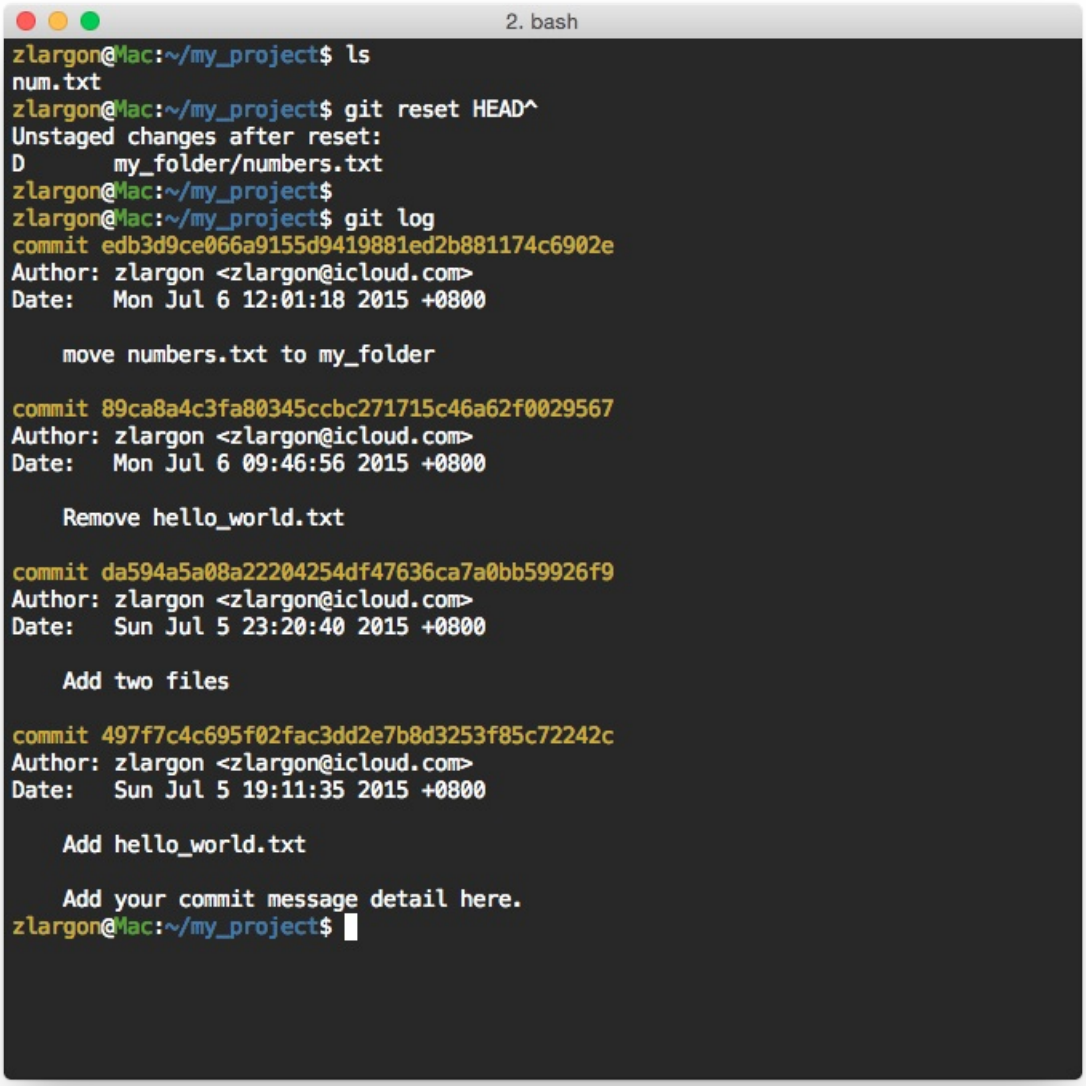
其實換句話說，***`git reset HEAD`*** 就是還原到 ***`HEAD`*** 這個 ***patch*** 的意思

有了這個概念之後，我們就可以試著切換到之前的 patch

使用 `git reset HEAD^` 回到上一個 patch

既然 `git reset HEAD` 是還原到 `HEAD` 這個 patch 的意思，那我們要回到上一個 patch，就只要改成用 `HEAD^` 就行了

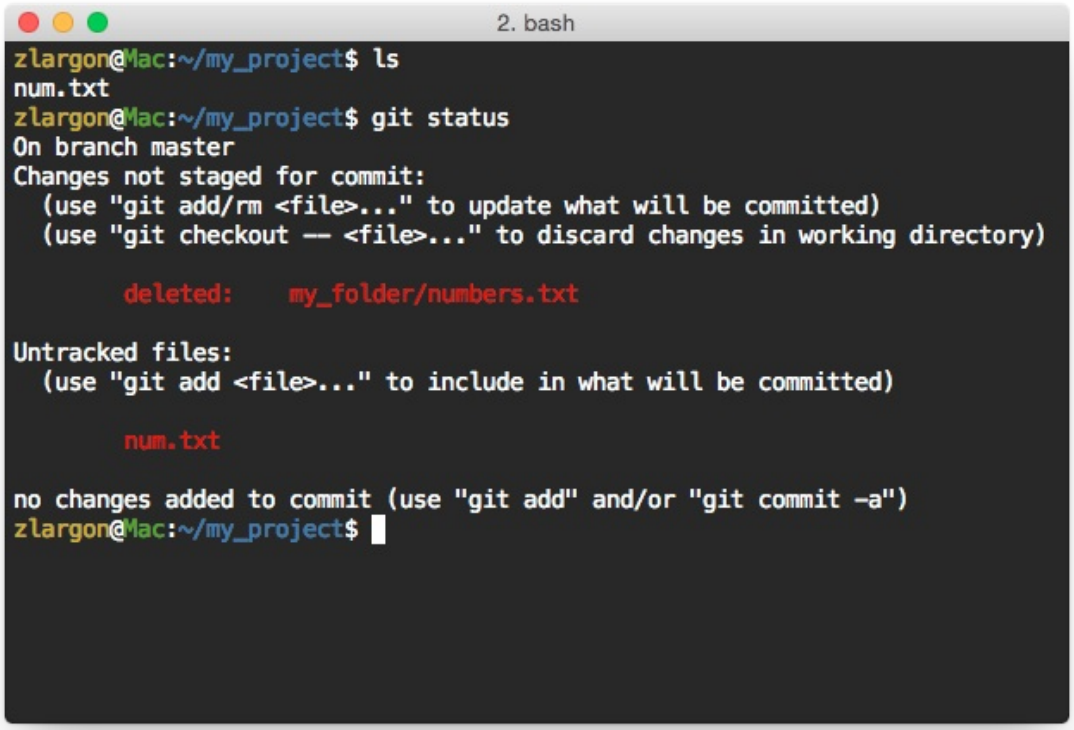
```
$ ls
$ git reset HEAD^
$ git log
```



我們可以從 `git log` 看出，原本最後一個 patch 的提交紀錄已經不見了

接著我們在用 `git status` 來看目前的檔案狀態

```
$ ls
$ git status
```



我們從 `ls` 可以看出，目錄下的檔案內容並沒有發生改變

跟 reset 之前一樣只有 `num.txt` 一個檔案

只有 "檔案狀態" 發生改變而已

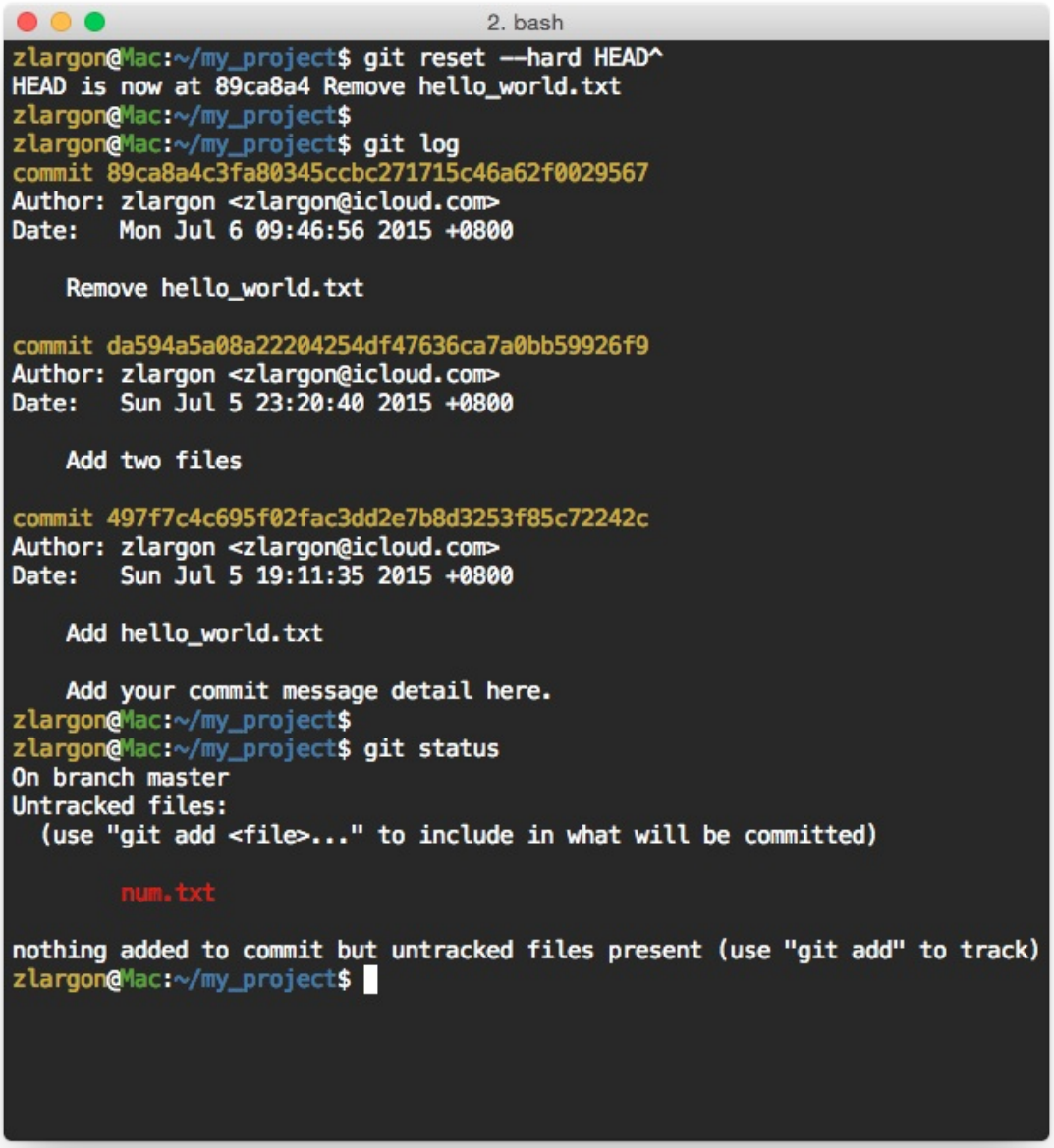
使用 `git reset --hard HEAD^` 回到上一個 patch，並且強制清除修改的內容

我們在 "檔案還原" 有提到，使用 `git reset --hard HEAD` 來還原所有檔案的內容

因此我們可以舉一反三，如果把 `HEAD` 改成 `HEAD^`

那麼就是回到上一個 patch 並且強制清除修改的內容

```
$ git reset --hard HEAD^
$ git log
$ git status
```



現在從 `git log` 可以看初，又少了一個 patch，現在只剩下 3 個 patch

從 `git status` 也只剩下 `num.txt` 檔案落在 ***Untracked files***

注意，git 不會去清除 `num.txt` 的內容，是因爲他先前已經轉到 ***Untracked files*** 區塊

而 git 本來就不會去追蹤這個區塊的任何變化

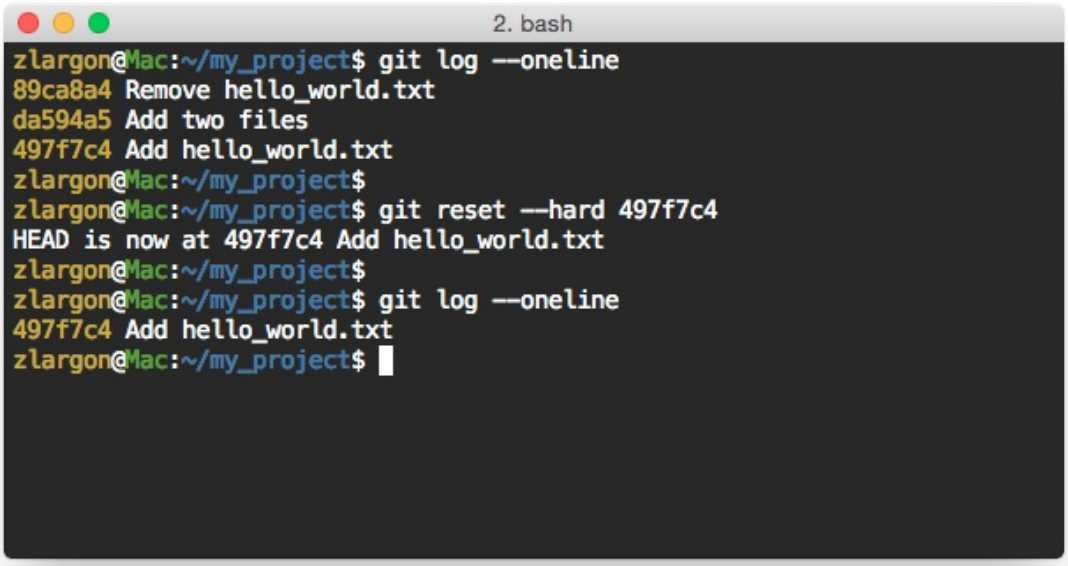
使用 `git reset --hard <commit id>` 直接 reset 成指定的 patch

我現在要 reset 到我們的第一次 commit 的 patch

我們可以先用 `git log` 來查詢我們第一個 patch 的 `commit id`

接著用 `git reset --hard <commit id>` 直接跳至該 patch

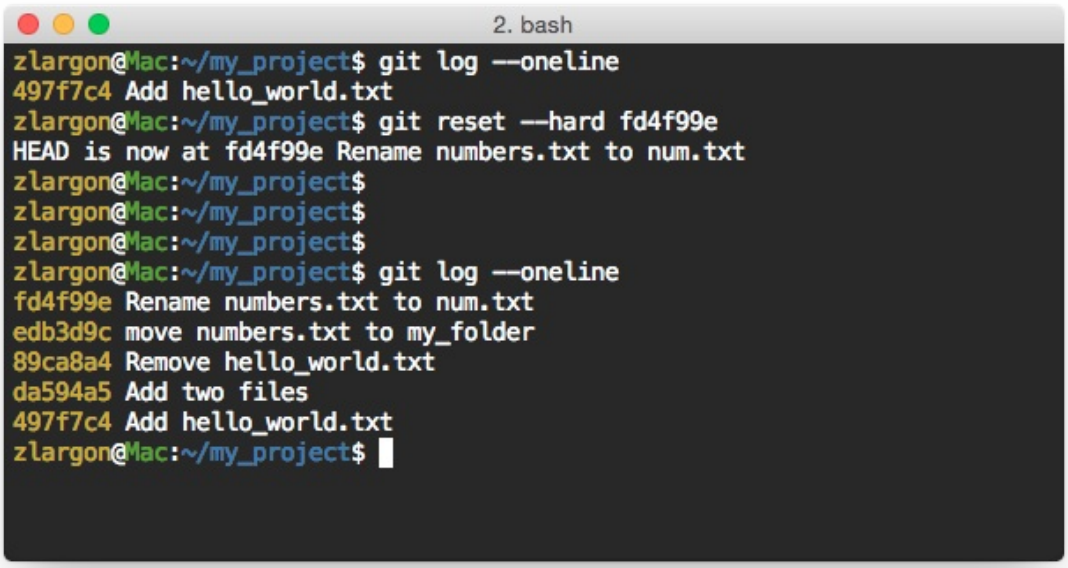
```
$ git log --oneline         # 只顯示一行的 log 訊息 → 第一個 patch 的 commit id 爲 497f7c4
$ git reset --hard 497f7c4
$ git log --oneline
```



我們除了可以還原之先前的 patch 之外，我們也可以透過 `git reset --hard <commit_id>` 跳至我們最後一次 commit 的 patch

之前我們最後一次 commit 的 patch id 爲 `fd4f99e6db01d8d35d39a990aadfef44cb8a2986`

```
$ git log --oneline
$ git reset --hard fd4f99e
$ git log --oneline
```



從 `git log` 可以看出，我們全部五個 patch 都回來了！

這個是一個非常實用的功能，可以快速的切換 patch！

`git reset --hard` 的注意事項

- 如果我程式改內容到一半，不小心下了指令 `git reset --hard HEAD`，那我原本改的內容都就都不見了嗎？

  > 是的，這無法補救... Orz
  >
  > 但是如果你是使用 IDE 或是文字編輯器的話（例如：Sublime Text、Notepad++），請趕快回到編輯器使用 Ctrl + Z 大法
  >
  > 第一時間還有機會救回來

- 如果我不小心用 `git reset --hard HEAD^^^...` 切回先前的 **patch**，然後又沒有把後來的 **commit id** 記下來，那該怎麼辦呢？

> 這一點可以不用擔心
>
> 「凡走過必留下痕跡，凡 commit 過的 patch 都會留下紀錄」
>
> git 會幫我們將所有 commit 過 patch 都存下來
>
> 我們下一個章節將會教大家怎麼把 patch 救回來

本章回顧

- 使用 `git reset HEAD^` 回到上一個 patch（檔案內容不清空）
- 使用 `git reset --hard HEAD^` 回到上一個 patch，並且強制清除修改的內容
- 使用 `git reset --hard <commit id>` 直接 reset 成指定的 patch

找回消失的 **Patch**

在 git 的世界裡，凡事 `HEAD` 發生改變都會被記錄下來

哪些時候 `HEAD` 會發生改變呢？

- **`git commit`**

  提交一份新的 patch，`HEAD` 會轉移到新的 patch

- **`git reset --hard <commit_id>`**

  切換 patch 的時候

- ***`git cherry-pick/revert ...`***

  挑入/挑出 patch 的時候

- **`git checkout <branch>`**

  切換分支的時候

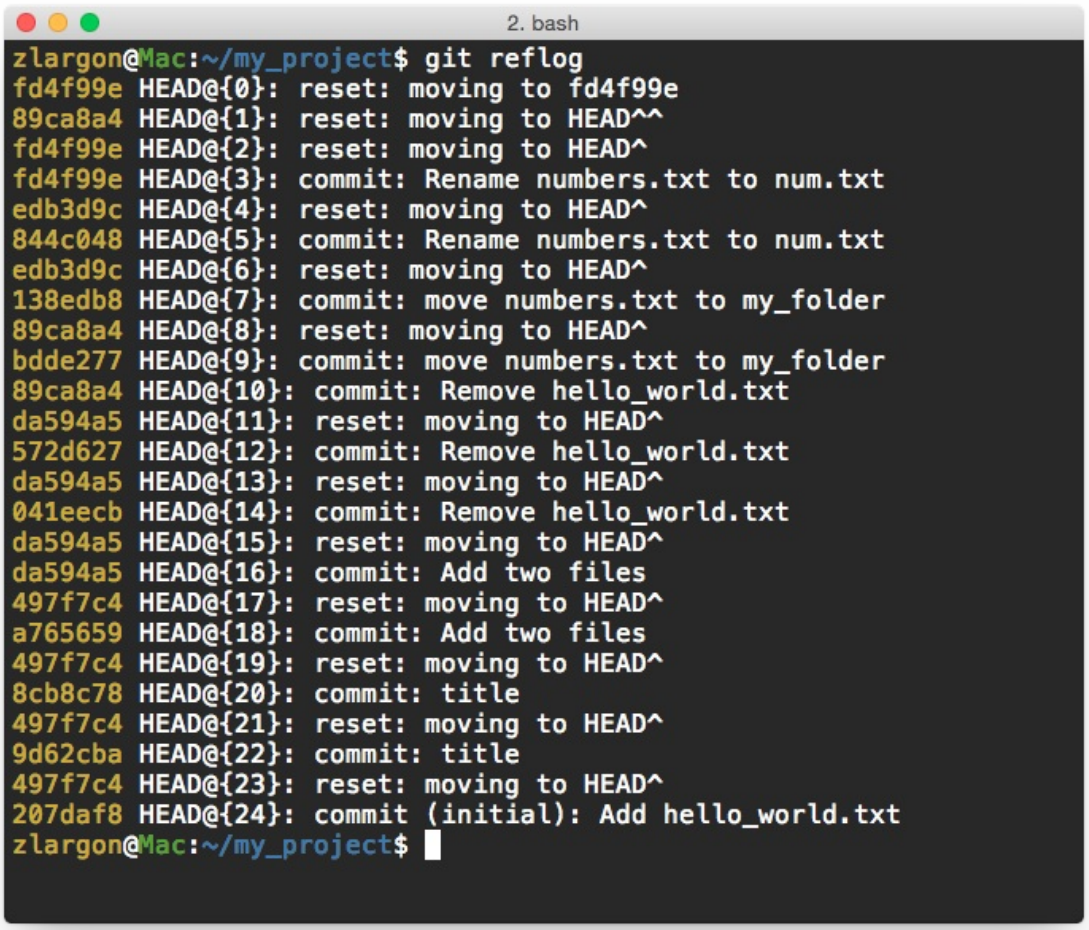
- **`git merge/rebase ...`**

  合併分支的時候

> 上述一些還沒提到的指令，我們在後面都會陸續爲大家介紹

使用 **`git reflog`** 來查看 **`HEAD`** 的修改紀錄

```
$ git reflog
```



`git reflog` 列出了 `HEAD` 改變的歷史紀錄

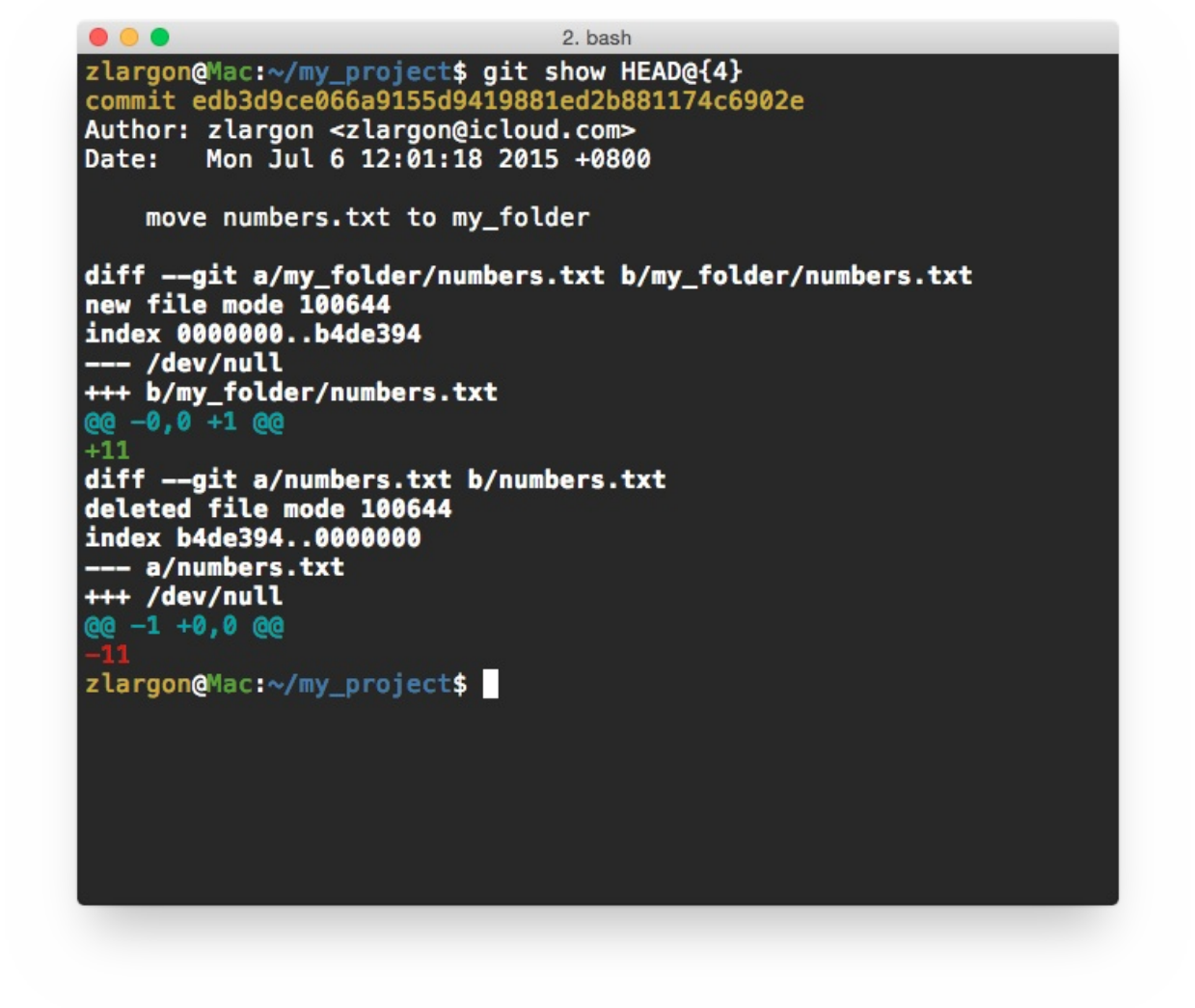
Commit Id	Short Name	Command	Description
`fd4f99e`	`HEAD@{0}`	`reset`	moving to fd4f99e
`89ca8a4`	`HEAD@{1}`	`reset`	moving to HEAD^^
`fd4f99e`	`HEAD@{2}`	`reset`	moving to HEAD^
`fd4f99e`	`HEAD@{3}`	`commit`	Rename numbers.txt to num.txt
`edb3d9c`	`HEAD@{4}`	`reset`	moving to HEAD^
...	...	...	...

我們可以從這裡看出我們當時在切換 `HEAD` 時下了什麼指令，以及做了什麼操作的簡易說明

我們可以使用 `git show` 來查看任一個 patch 的內容

可以用 `commit id` 或是 `short name` 來查詢

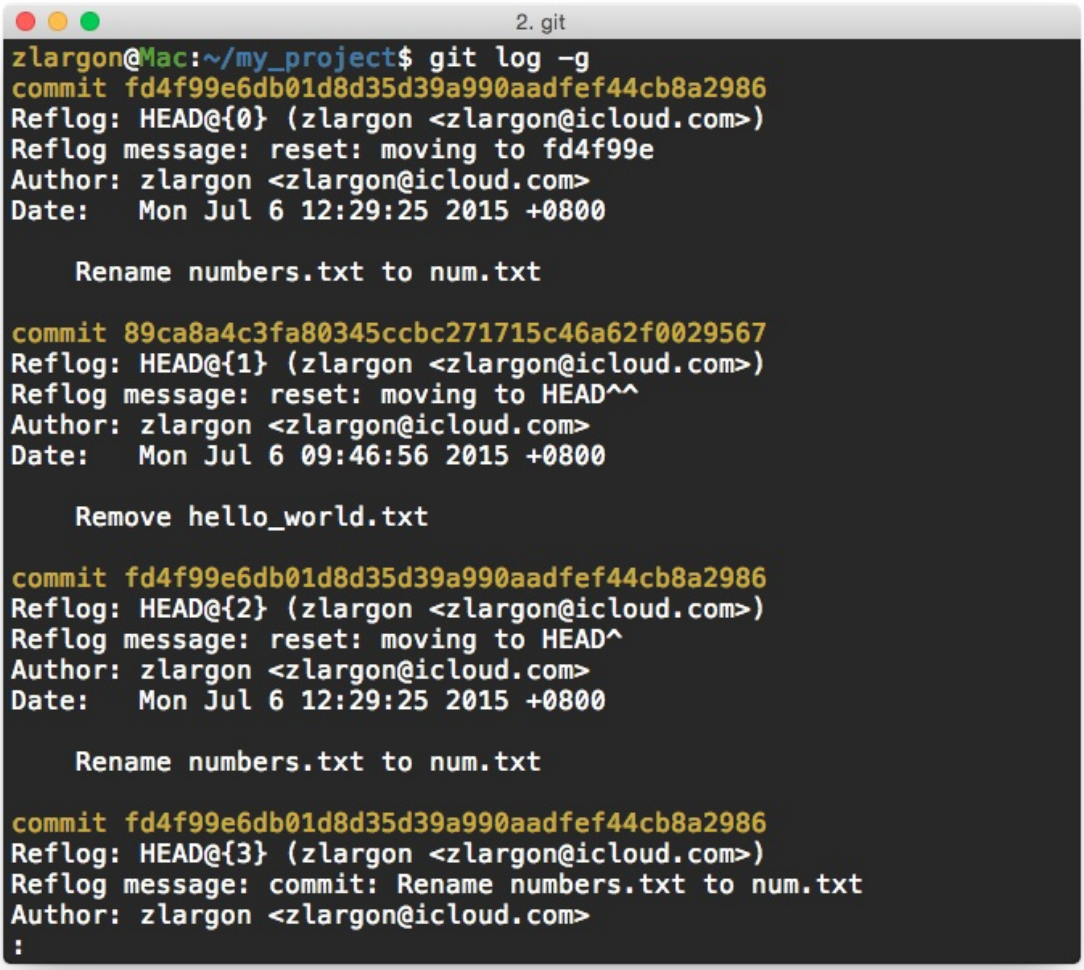
```
$ git show <commit id>
$ git show <short name>
```



這份紀錄至少會保存一個月，所以你完全不用擔心 reset 掉的 patch 會不見

都可以從這裡找得到

使用 `git log -g` 查看 reflog 的詳細內容



本章回顧

- 使用 `git reflog` 來查看 `HEAD` 的修改紀錄
- 使用 `git log -g` 查看 reflog 的詳細內容

修改 / 訂正 Patch

有時候我們在 commit 時，會不小心 add 錯檔案，或是 commit message 寫錯、打錯字

我們不想要再 commit 一個 patch，只是想修改原本的內容而已

這時候我們該怎麼做呢？

應該已經有人想到可以這樣做了

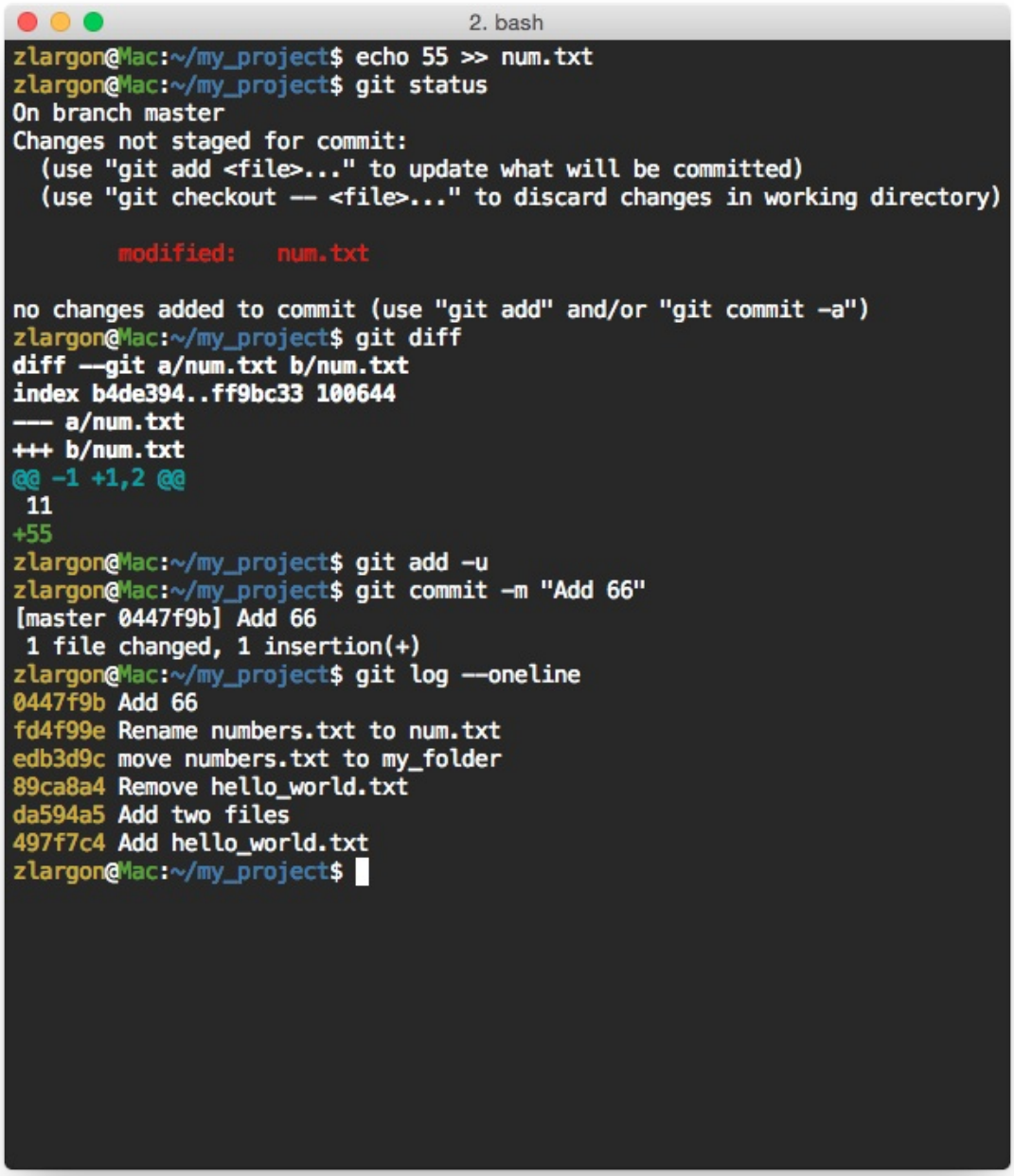
```
$ git reset HEAD^       # 恢復上一個 patch 的狀態
$ git add <file>        # 重新 add files
$ git commit            # 重新 commit
```

除了以上這種方法之外，我們在這邊另外提供兩種做法供參考

1. 使用 `git reset --soft HEAD^` 使 patch 回到上一個階段的 *Changes to be committed*

首先我們先在 `num.txt` 的最後一行新增 "55" 然後提交

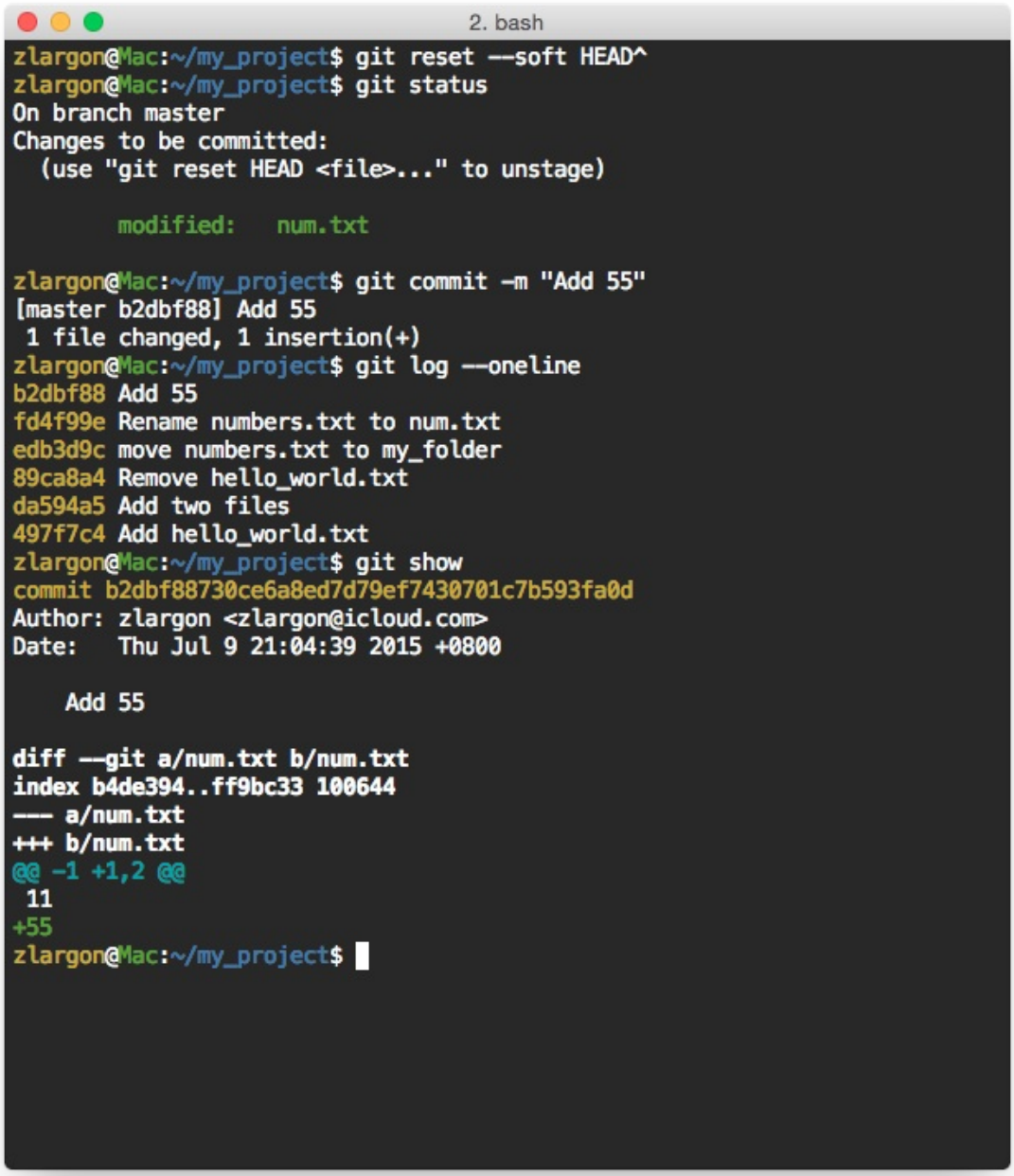
```
$ echo 55 >> num.txt            # 新增 55
$ git add -u                    # add 修改的部分
$ git commit -m "Add 66"        # 提交（這是錯誤的 message）
```



顯然這次的提交訊息是錯誤的，應該改成 "Add 55" 才對

在這裡我們可以使用 `git reset --soft HEAD^` 來回到上一步，但是不需要重新 `add` 檔案

```
$ git reset --soft HEAD^        # 會把前一次 add 過的內容，保留在 Changes to be committed 區塊
$ git commit -m "Add 55"
```



請注意，原本的 commit id 爲 `0447f9b`，但是重新提交後 id 變成 `b2dbf88`

儘管第二次的內容跟第一次是一模一樣的，凡事只要做了 commit 的動作，git 就會重新產生一組新的 `commit id`

其實這個方法就只是省掉重做 `git add` 的時間而已

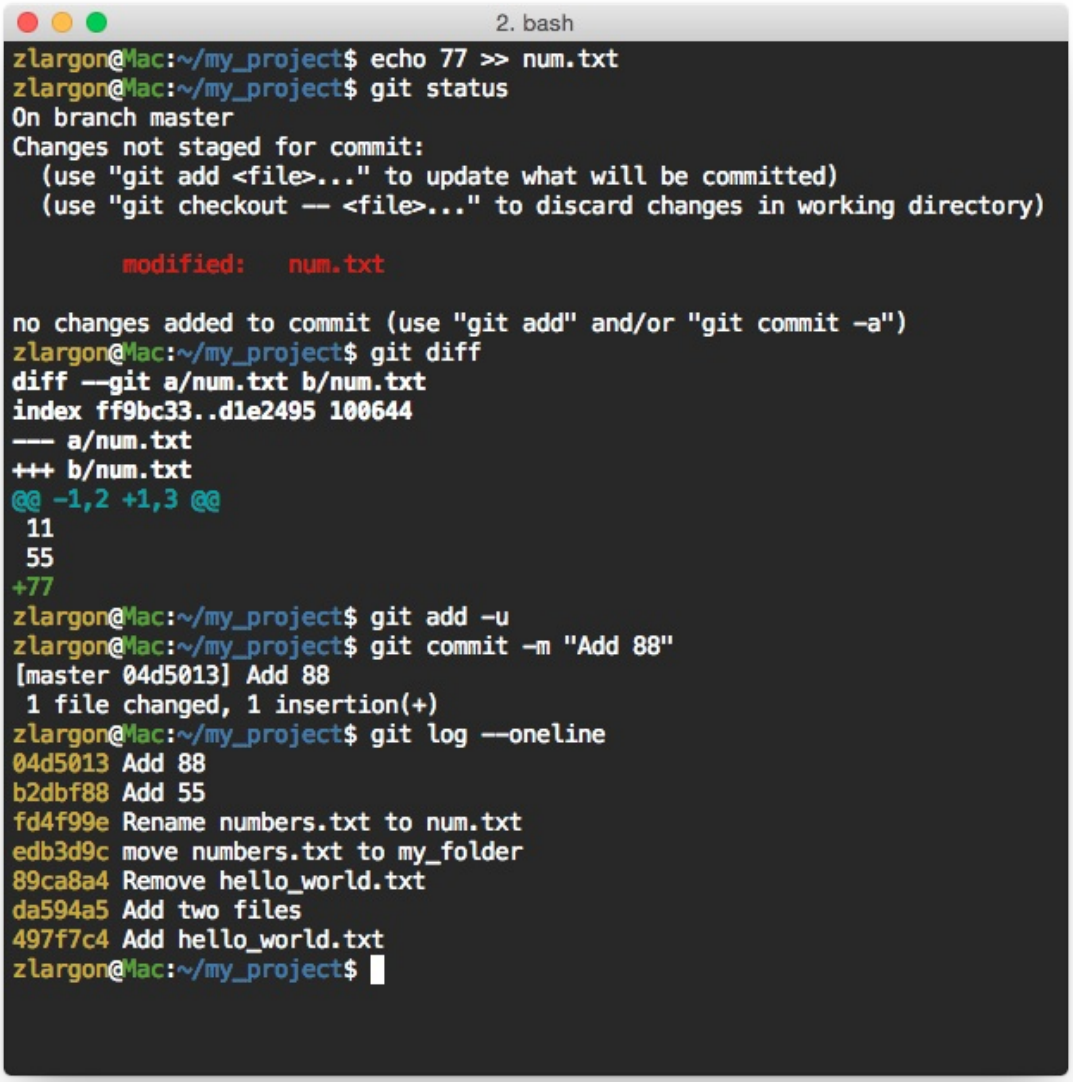
比較 `git reset HEAD^` 的參數

指令	效果
`git reset HEAD^`	回到前一個 patch，且恢復檔案的狀態
`git reset --soft HEAD^`	回到前一個 patch，但保持檔案狀態爲 ***Changes to be committed***
`git reset --hard HEAD^`	回到前一個 patch，且強制清除檔案的修改內容

2. 使用 `git commit --amend` 修改提交訊息

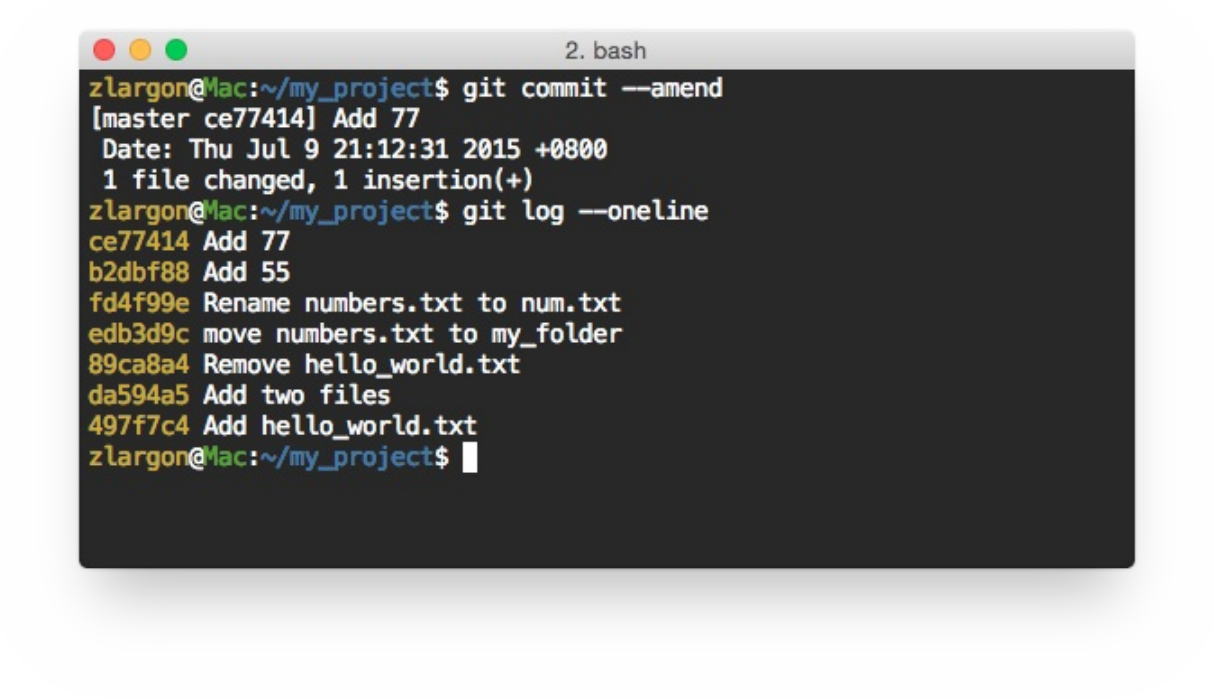
我們在提交一個新增 "77" 的 patch，但是提交內容故意打錯成 "Add 88"

```
$ echo 77 >> num.txt            # 新增 77
$ git add -u                    # add 修改的部分
$ git commit -m "Add 88"        # 提交（這是錯誤的 message）
```



這次我們使用 `git commit --amend` 直接修改提交訊息

```
$ git commit --amend    # 使用後會進入 vim 文字編輯模式。把標題改成 "Add 77" 然後存檔離開
```



跟前一個方法一樣，原本的 commit id 已經被改變

因爲凡事只要做了 commit 的動作，git 就會重新產生一組新的 `commit id`

而 `git commit --amend` 後面也可以加 `-m` 的參數帶入新的 commit message

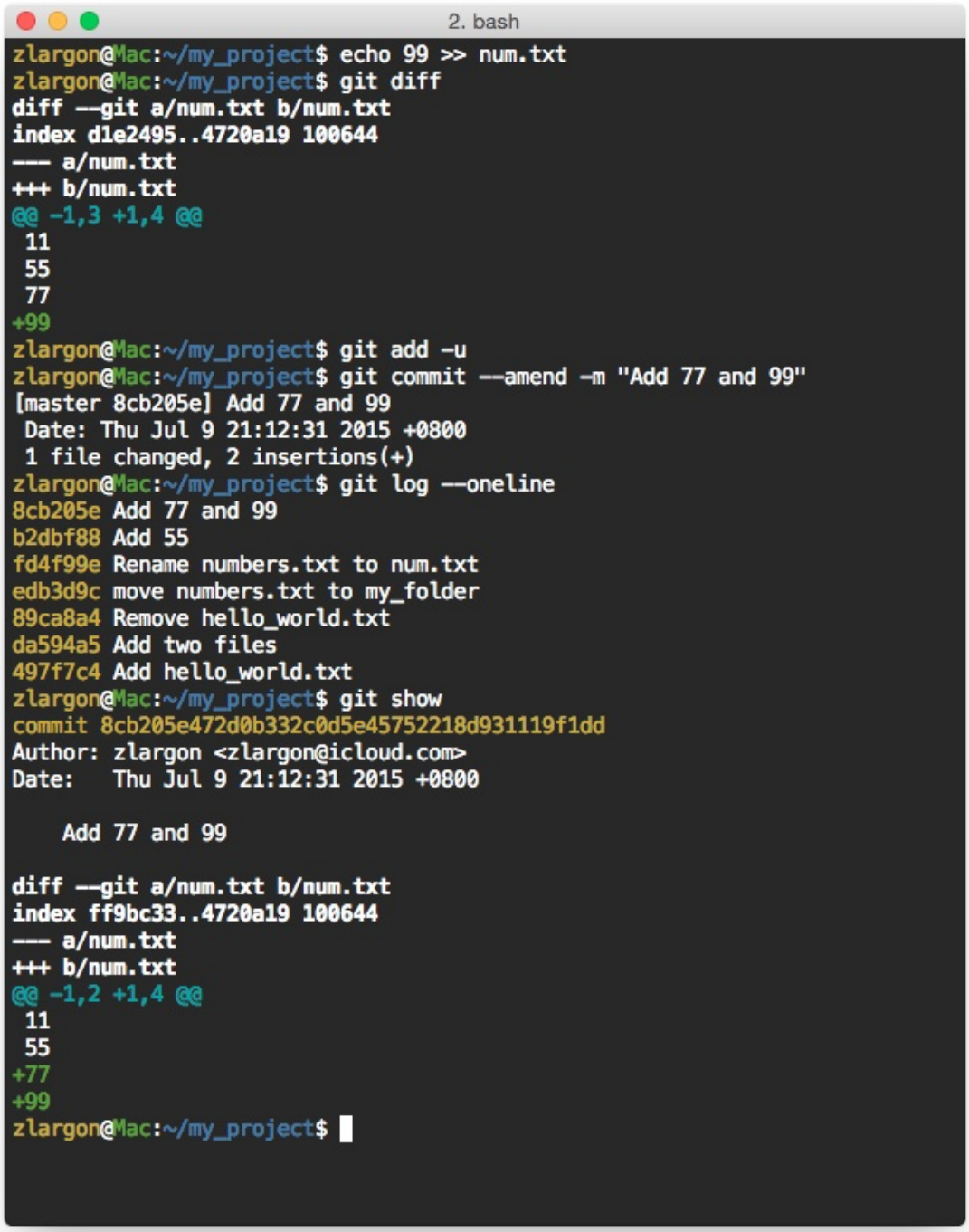
```
$ git commit --amend -m <title>                              # 只要提交 title
$ git commit --amend -m <title> -m <message>                 # 提交 title 以及 message
$ git commit --amend -m <title> -m <msg1> -m <msg2> ....     # 提交多段 messages
```

使用 `git commit --amend` 修改提交的內容

如果我想要修改剛才的 patch，除了新增 77 之外，還要在同一個 patch 新增 99 的內容

那可以用以下的方法，把修改 99 的部分，合併到原本的 patch 裡面

```
$ echo 99 >> num.txt                          # 新增 99 到最後一行
$ git add -u                                  # add 修改的部分
$ git commit --amend -m "Add 77 and 99"
```



從這裡可以看到，我們已經成功把 99 合併到前一個 patch 裡面了

同時 commit id 也發生改變了

`git commit --amend` 的注意事項（非常雷 x 2）

使用指令 `git commit --amend` 後，會馬上進入 vim 的文字編輯模式

git 不論你是否有修改 ***commit message*** 或是用 `:q!` 不存檔直接離開 ***vim***

都會爲你重新產生新的 ***commit id***

有時候這並不一定是我們想要的結果，要特別留意！

> 例如說，我們從 server 上抓了最新的 code 下來，準備要新增一個功能再上傳至 server
>
> 但是你好死不死的手殘按到 `git commit --amend` 先改到了最新的 patch（P0）
>
> 這時候儘管你按 `:q!` 退出 vim，但是 commit id 早就已經被改掉了（P0'）
>
> 然後我們基於 P0' 來開發新功能，並且 commit 了一個新的 patch（P1）
>
> 這時候我們要上傳 patch 的時候，會遇到被 git server 拒絕的情形
>
> 被拒絕的原因是，server 認爲你提交的 patch 跟 server 的 code 並沒有兩樣
>
> 但其實主要原因是，server 認爲你上傳了兩個的 patch，分別是 P0' 跟 P1
>
> server 先用 P0' 跟 P0 最比對，發現兩個 patch 根本就一模一樣，就直接拒絕了
>
> 當你絞盡腦汁的想要找出 P1 哪裡有問題的時候，其實是被 `git commit --amend` 背後默默捅了一刀
>
> 因此上傳 patch 前請確保 parent id 正確無誤

另外這個指令還有一個很雷的地方是

有時候我們已經開發好一個新功能，也都測過沒問題了

把全部修改的內容都 `add` 好之後，準備要用指令 `git commit` 來提交

但是卻不知道爲什麼很順手的在後面又加上了 ***`--amend`*** ***...***

就在你按下 enter 的瞬間，悲劇已經發生了...

Changes to be committed 的內容已經完美的跟上一個 patch 融合在一起了...

這時候該怎麼把他們分離呢？

就在我們學完 `git reflog` 跟 `git reset --soft HEAD^` 之後

只要一行指令就可以把他們完美的分離了 XD

```
$ git reset --soft HEAD@{1}
```

本章回顧

- 使用 `git reset --soft HEAD^` 使 patch 回到上一個階段的 ***Changes to be committed***
- 使用 `git commit --amend` 修改提交訊息及內容
- 保命技 `git reset --soft HEAD@{1}`

移除單一個 **Patch**

先前所教到 `git reset` 都是回到過去的某個點上

例如說，我們現在的 patch 如下；我們要從 P5 回到 P3 的點上，就會用到 `git reset`

```
P0    →    P1    →    P2    →    P3    →    P4    →    P5
                                                      (HEAD)
```

```
$ git reset --hard P3    # 回到 P3 這個點上

P0    →    P1    →    P2    →    P3
                               (HEAD)
```

回到 P3 的同時，P4 跟 P5 也會跟著被拿掉

但如果我今天只是想要把 P3 這個點單獨抽掉，要怎麼做呢？

```
我們直覺想到的畫面，可能會是這樣

P0    →    P1    →    P2    →    P4    →    P5
                                          (HEAD)
```

```
但實際上是這樣的

P0    →    P1    →    P2    →    P4'    →    P5'
                                           (HEAD)
```

爲什麼說是 P4' 跟 P5' 呢？

還記得我們之前有講過 patch 的 "基本概念" 嗎？

每個 patch 都有它的 parent，而 patch 只是記錄他與 parent 間的變化

patch 會一個接著一個的串連在一起的

因此當我們想要把 P3 抽掉

其實就是把 P4 接到 P2 的後面，然後就會獲得一個新的 P4'

再把原本的 P5 接到新的 P4' 後面，獲得一個新的 P5'

使用 `git cherry-pick <commit id>` 挑入 patch

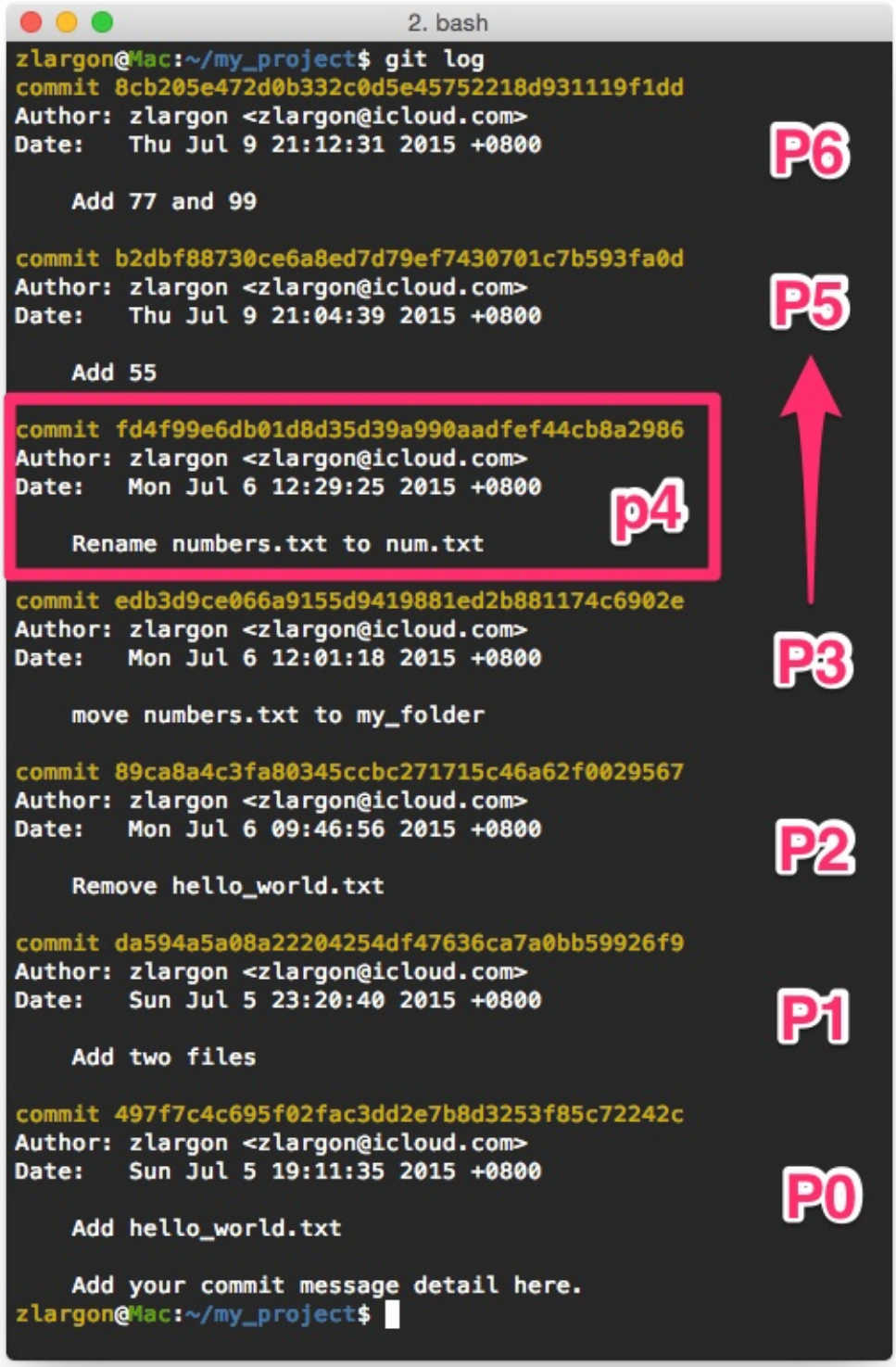
因此我們的策略是這樣的

我們先回到 ***P2***，然後再把 ***P4*** 跟 ***P5*** 重新加進來

這時候我們就會用 `cherry-pick` 的指令來挑入 patch

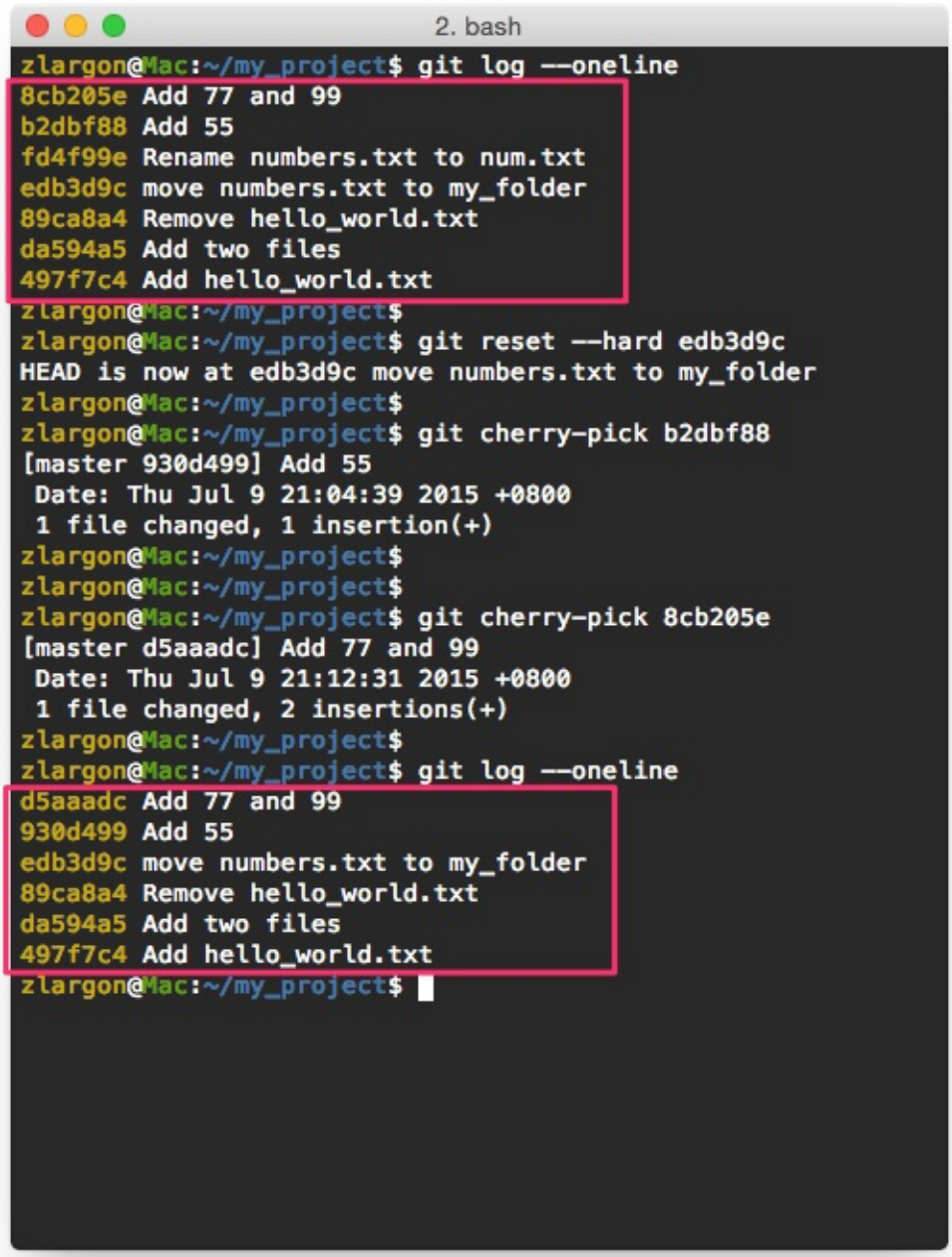
```
$ git reset --hard  P2      # 回到 P2
$ git cherry-pick   P4      # 挑入 P4 → P4'
$ git cherry-pick   P5      # 挑入 P5 → P5'
```

現在我們來用剛才的範例實際操作一次



我們現在的目標是，把 `P4` 移除，讓 P5 直接接在 P3 後面

```
$ git reset --hard  edb3d9c     # 回到 P3
$ git cherry-pick   b2dbf88     # 挑入 P5 → P5'
$ git cherry-pick   8cb205e     # 挑入 P6 → P6'
```



這裡我們已經成功把 P4 移除

而且我們看到最後的兩個 P5' 和 P6' 的 commit id 都已經改變

而原本的 P5 和 P6 會被 git 記錄在 `reflog` 裡面

小結

git 之所以可以這樣任意挑 patch，要歸功於他把每個 patch 都拆成一個一個的小變化量

這樣的設計可以很容易的打散，再重組起來

用起來非常輕巧靈活

- 由於 `cherry-pick` 的指令太長了，我通常會設置別名 `cp`，這樣操作起來會比較方便（參考 "配置"）

延伸：調換 **Patch** 的順序

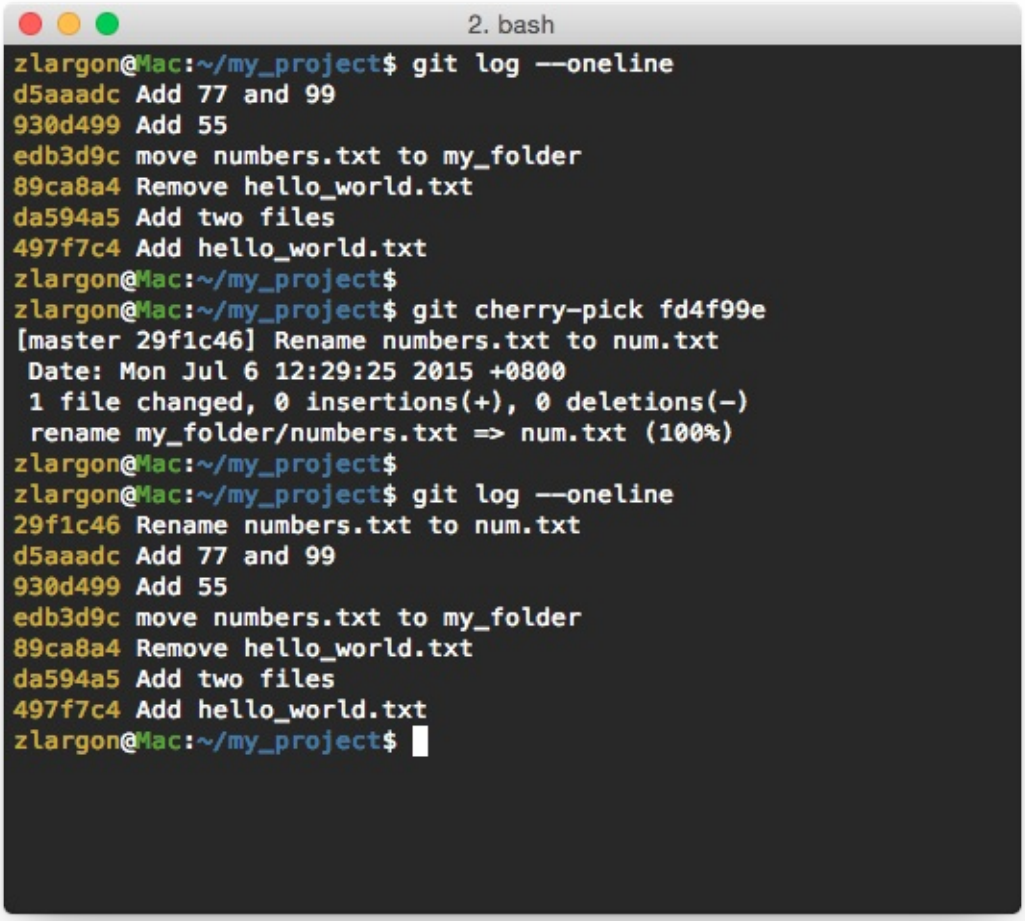
在我們學會 `cherry-pick` 之後，其實我們就已經學會如何調換 patch 的順序了

依照剛剛的範例，我們想把 P4 的位置跟 P5、P6 調換

```
P3   →   (P4)   →   (P5   →   P6)

P3   →   (P5'   →   P6')  →   (P4')
```

其實就只要把 P4 再挑回來就ＯＫ了

```
$ git cherry-pick  fd4f99e      # 挑入 P4 → P4'
```



`cherry-pick` 的注意事項

大部份我們在做 `cherry-pick` 的時候，只要每個 patch 的改動幅度不要太大

git 都可以幫我們順利接好，就像剛才我們範例那樣

但是 git 也會遇到無法判斷的時候，這就是所謂的版本衝突 ***Confilct***

也就是說，兩個 patch 改動的內容實在是太相似了，以至於無法判斷該如何合併 code

這時候 git 會要求我們手動去解 ***Confilct***

解 ***Confilct*** 常常是許多初學者的夢魘，看到 ***Confilct*** 就整個人呆掉了

我們在後面的 "Cherry-Pick 版本衝突" 會再爲大家介紹什麼是 ***Confilct***，如何解 ***Confilct***

以及如何降低 ***Confilct*** 發生的機會

本章回顧

- 使用 `git cherry-pick <commit id>` 挑入 patch

Rebase 互動模式

`git cherry-pick <commit id>` 指令在操作少量的 patch 的時候非常實用，概念也很直覺

但是如果要一次操作多量的 patch 的時候，就要下非常多次指令，這樣就會很容易出錯

因此 git 提供一個基於 `cherry-pick` 之上，更高階的 git 指令 `rebase`

這個 `git rebase` 通常用來分支的問題（我們在之後的章節才會講到 git 的分支）

不過 `git rebase` 有提供一個叫做「互動模式」的功能，可以讓我們很方便的處理多個 patch

讓我們可以輕鬆的抽掉單一個 patch 或變更 patch 的順序

使用 `git rebase -i <after this commit>` 啓動「互動模式」

```
$ git rebase -i <after this commit>               # 啓動 rebase 互動模式
$ git rebase --interactive <after this commit>    # 同上
```

`<after this commit>` 是用來告訴 git，我們要修改的範圍到是 **"從哪個 *patch* 之後到 `HEAD` "**

以先前的例子來說，假設我們想要更動的範圍是 P4 ～ `HEAD`

那麼這裡的 `<after this commit>` 就會是 P3

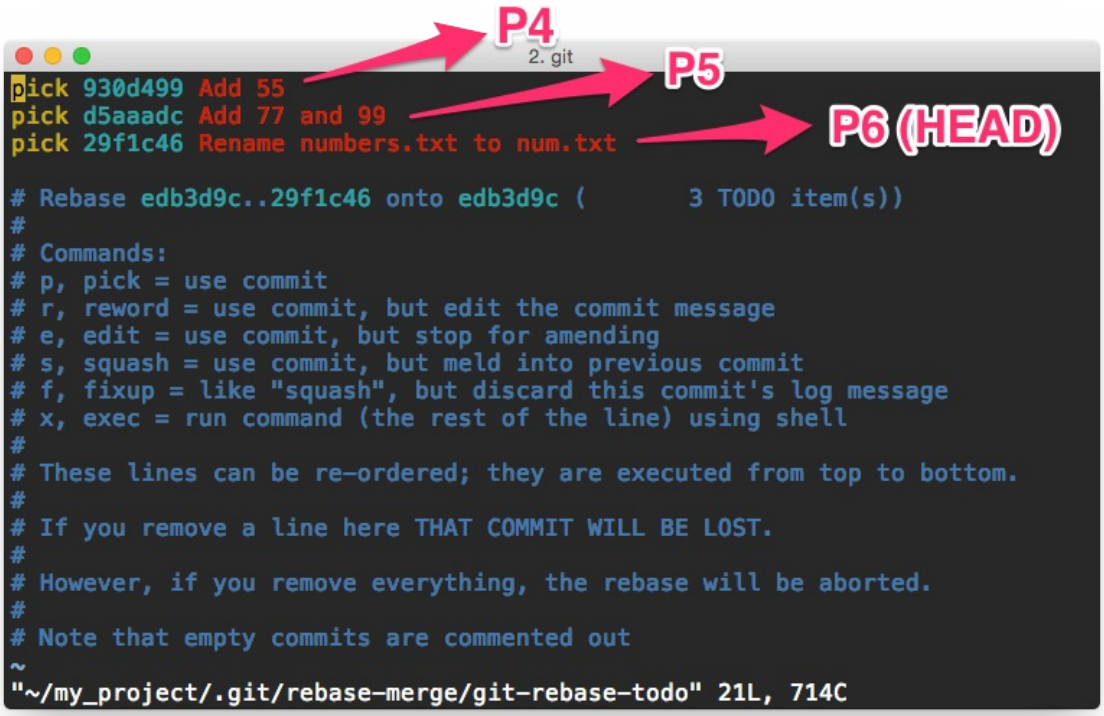
```
P0    →    P1    →    P2    →    [P3]    →    P4    →    P5    →    P6

                         (after this commit)                   (HEAD)
```

表示我要修改的範圍是從 P4 ～ `HEAD` (P6)


```
$ git rebase -i edb3d9c
```

按下 enter 後，就會進入 vim 的文字編輯模式，用來編輯 `Rebase TODO`

文件的上方是 TODO 的項目，下方是使用說明



Rebase TODO 使用說明

- ***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***

> ***pick***（預設值）
> `cherry-pick` 這個 patch
>
> ***reword***
> `cherry-pick` 這個 patch，並會在執行到這個步驟的時候，會停下來讓你修改提交訊息
>
> ***edit***
> `cherry-pick` 這個 patch，並會在執行到這個步驟的時候，會停下來讓你修改提內容
> 可以在這個時候 `git add/rm` 檔案
>
> ***squash***
> `cherry-pick` 這個 patch，但是會和前一個 patch 合併在一起
>
> ***fixup***
> `cherry-pick` 這個 patch，會和前一個 patch 合併在一起，但是會捨棄這個 patch 的提交訊息
>
> ***exec***
> `cherry-pick` 這個 patch，並且執行一個 shell script

- ***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***

> 你可以隨意的重新排序
> 他會由上而下的執行 `Rebase TODO`
> 如果你把某一行刪掉或是註解掉，那個 patch 將會不見
> 如果把 `TODO` 都清空或註解掉，那麼這次的 rebase 將會被放棄

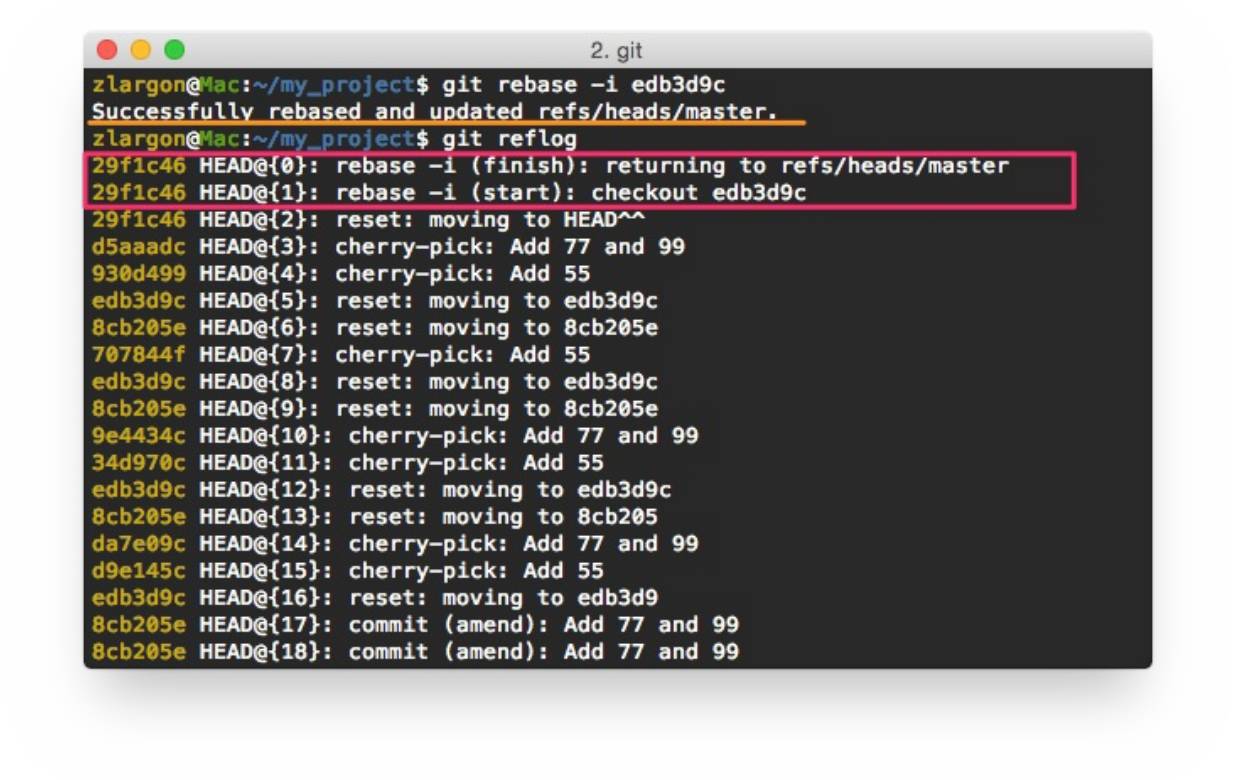
編輯 Rebase TODO

git 會由上而下依序執行 `Rebase TODO` 裡的指令

而他一開始就會給一份預設的 `Rebase TODO`，我們只要對他貼貼剪剪就可以了

指令	Commit Id	Commit Title	對應的 **git** 操作
pick	**930d499（P4）**	Add 55	`git cherry-pick 930d499`
pick	**d5aaadc（P5）**	Add 77 and 99	`git cherry-pick d5aaadc`
pick	**29f1c46（P6）**	Rename numbers.txt to num.txt	`git cherry-pick 29f1c46`

如果直接存檔離開的話，patch 將不會發生任何變化，但 `git reflog` 會多兩筆 `rebase -i (start)` 和 `rebase -i (finish)` 的紀錄

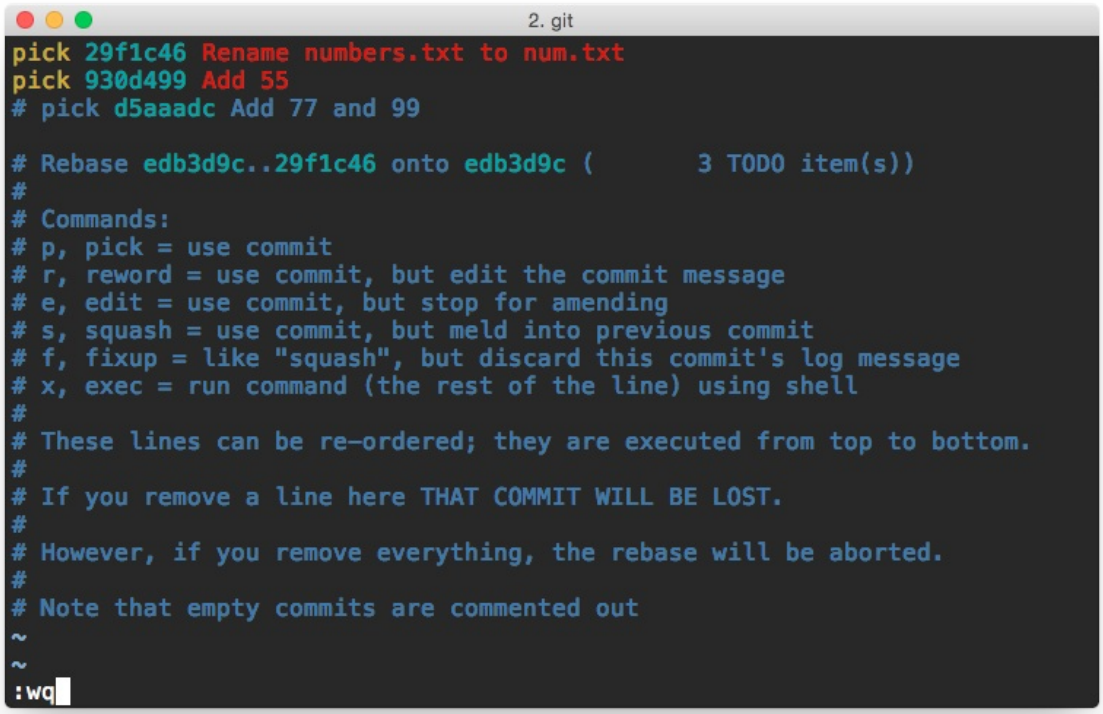


如果不想要有 `git reflog` 的紀錄的話，就要將把 `TODO` 的內容清空或是註解掉

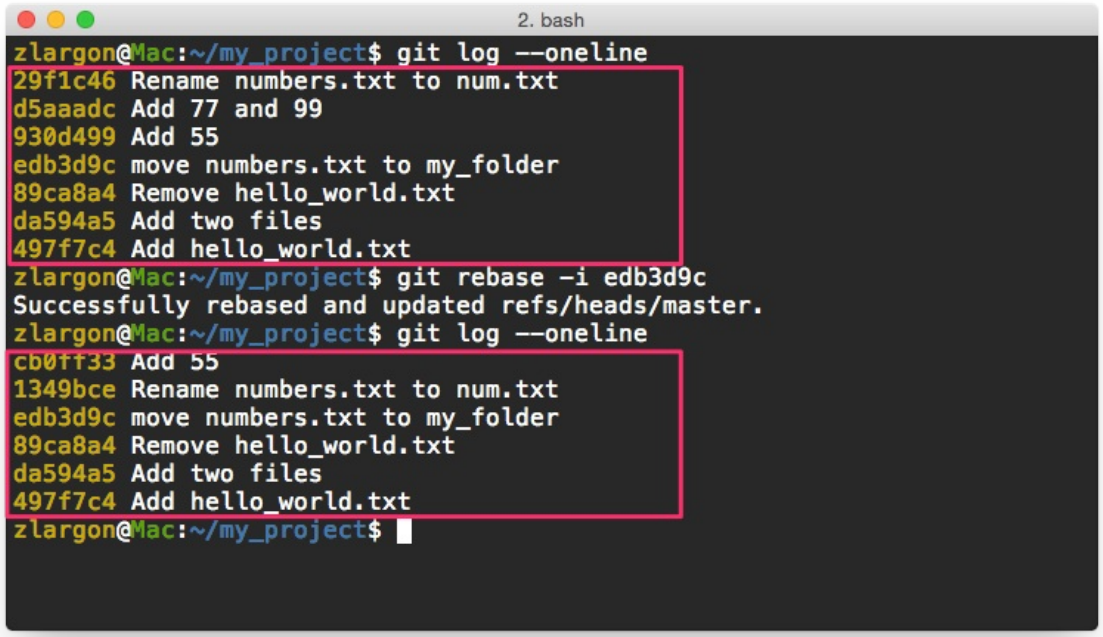
> 我個人是覺得多兩行 reflog 應該沒什麼關係 @@"

我們現在修改 `Rebase TODO`，讓他把 P6 移到最前面，並且把 P5 刪掉

指令	Commit Id	Commit Title	對應的 **git** 操作
pick	**29f1c46（P6）**	Rename numbers.txt to num.txt	`git cherry-pick 29f1c46`
pick	**930d499（P4）**	Add 55	`git cherry-pick 930d499`
~~***pick***~~	~~**d5aaadc（P5）**~~	~~Add 77 and 99~~	註解或整行刪除，表示要捨棄這個 patch



改完之後存檔離開，並且用 `git log` 來查看結果



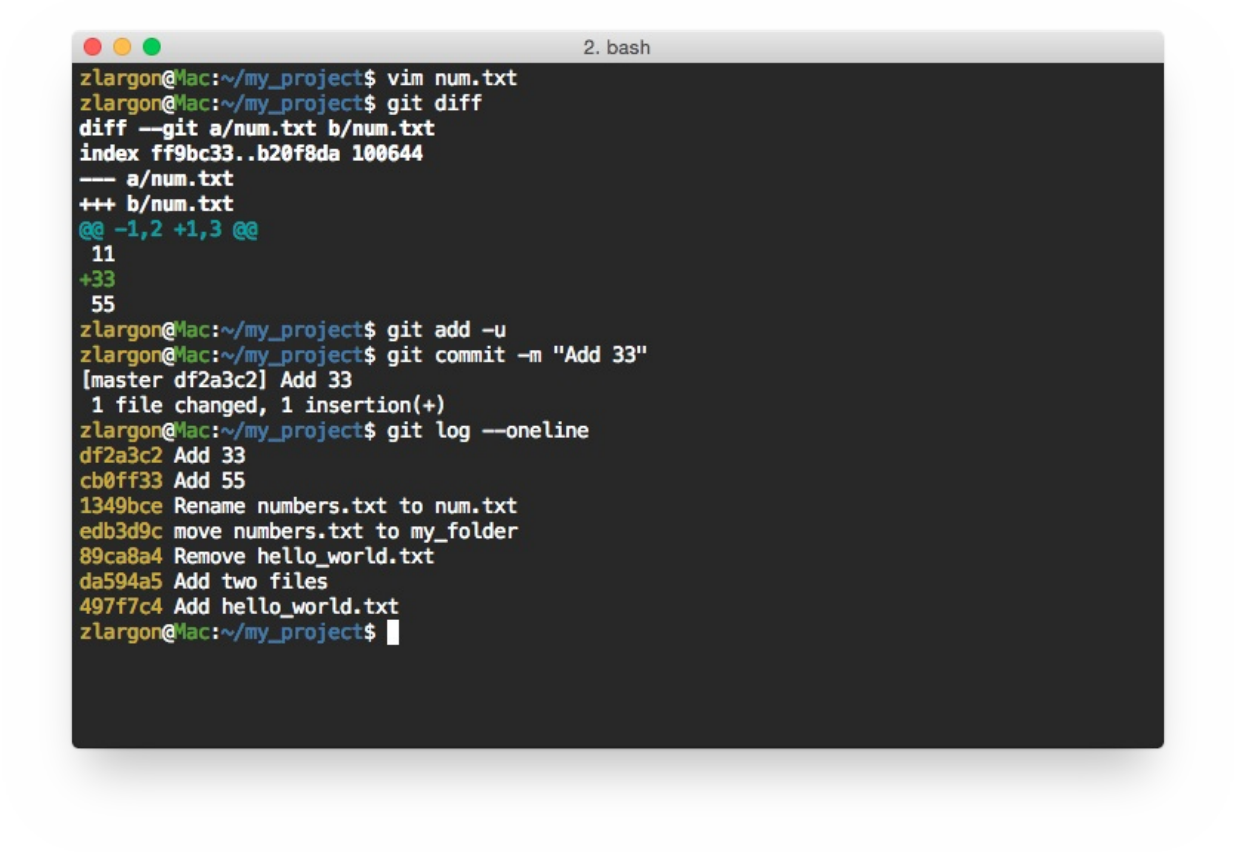
所有的 patch 確實是按照我們想要的成功改好了！

Cherry-Pick 版本衝突

在 "移除單一個 Patch" 的結尾，我們有提到有時候我們在做 `cherry-pick` 的時候會遇到版本衝突的問題

我們現在就來製造一個版本衝突的情況

現在先提交一個 patch "Add 33" 爲 `num.txt` 新增 "33" 的字串



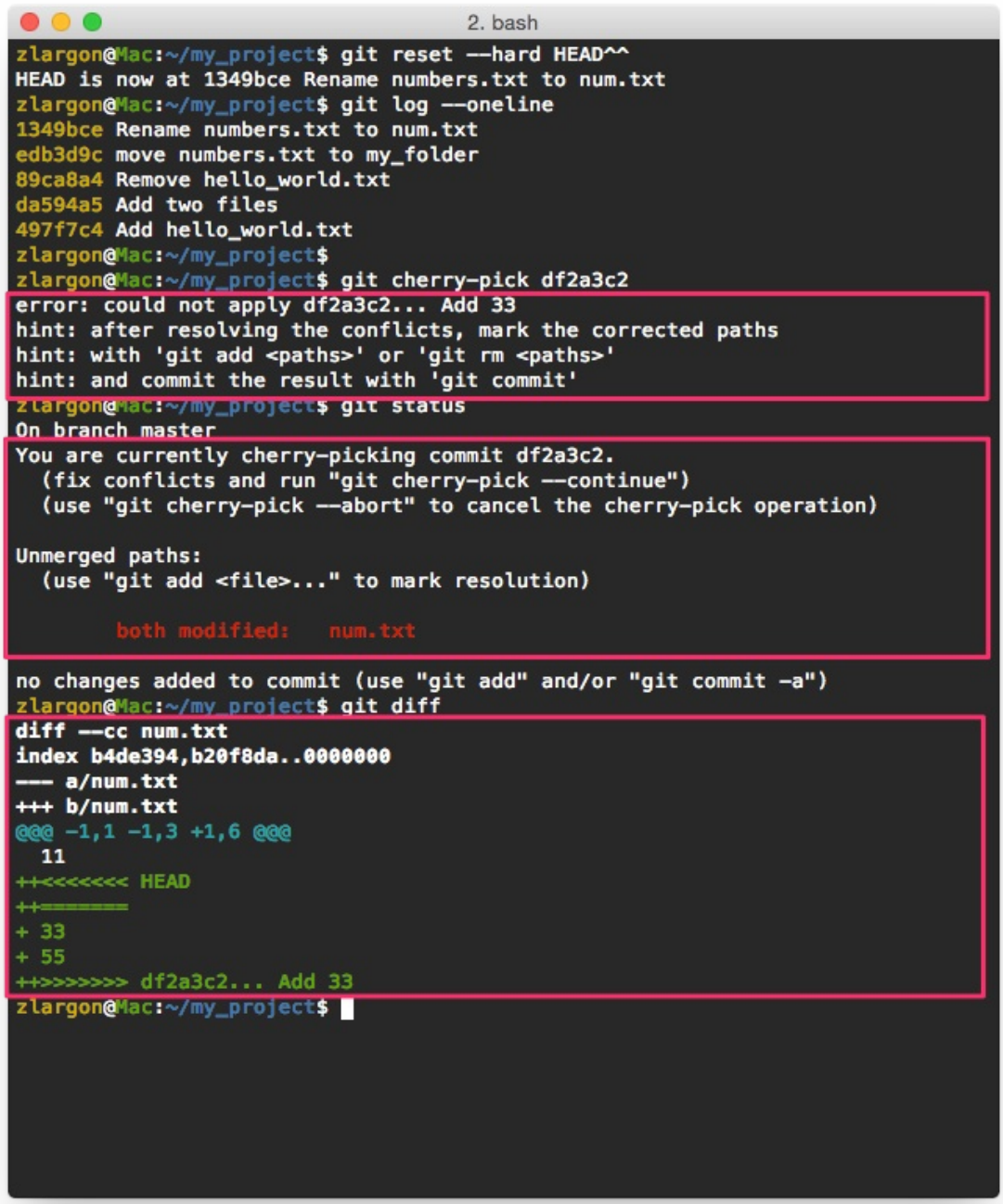
而我們故意把最後兩個 patch 的順序對調，使他發生 ***conflict***

也就是說，我們希望新的順序是這樣的

```
P0    →    ...    →    P4    →    P5    →    P6        原本的順序

P0    →    ...    →    P4    →    P6'   →    P5'       新的順序
```

我們來看看會發生什麼事情

```
$ git reset --hard HEAD^^       # 回到 P4
$ git cherry-pick df2a3c2       # cherry-pick  P6  →  發生 conflict
$ git status                    # 查看檔案狀態
$ git diff                      # 查看檔案內容
```



git 印出 cherry-pick 時，所發生的錯誤訊息：

```
error: could not apply df2a3c2... Add 33
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add ' or 'git rm '
hint: and commit the result with 'git commit'
```

git 要求你解決 ***conflict*** 的問題，並且解完後使用 ***`git add/rm`*** 你的檔案，再執行指令 ~~***`git commit`***~~

> 稍後我會解釋，爲什麼要把他劃掉

使用 `git status` 來查看檔案狀態

```
You are currently cherry-picking commit df2a3c2.
  (fix conflicts and run "git cherry-pick --continue")
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

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

    both modified:   num.txt
```

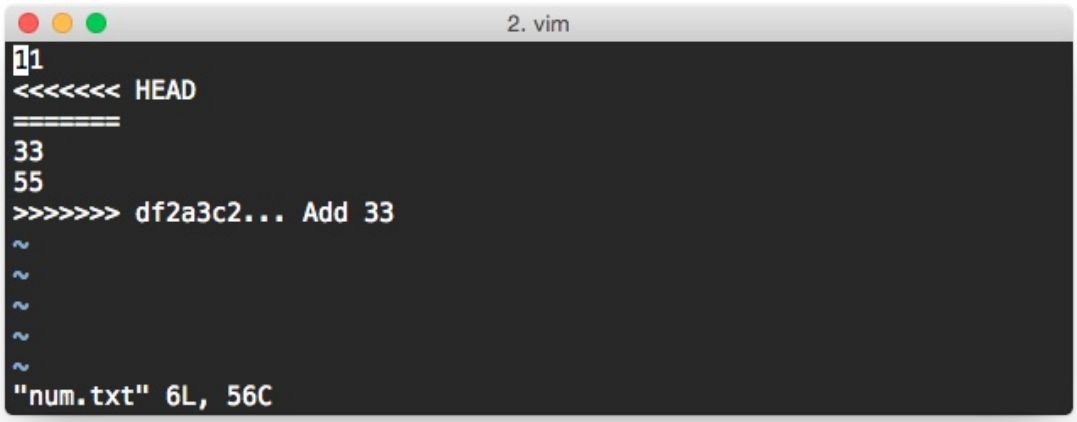
- ***fix conflicts and run "git cherry-pick --continue"***
- ***use "git cherry-pick --abort" to cancel the cherry-pick operation***

> 解完衝突請執行 `git cherry-pick --continue`
> 若要放棄請執行 `git cherry-pick --abort`

git 會將所有發生 ***conflict*** 的檔案，放在 ***Unmerged paths*** 區塊

`num.txt` 的狀態爲 *both modified* 的意思是，兩個 patch 都有同時修改到這個檔案

接著我們使用 vim 來開啓檔案 `num.txt` 來解 ***conflict*** 吧！




```
<<<<<<< patch_1
```

> 這裡是 *patch_1* 對此檔案所修改的內容

```
=======
```

> 這裡是 *patch_2* 對此檔案所修改的內容

```
>>>>>>> patch_2
```

從 `<<<<<<< patch_1` 到 `>>>>>>> patch_2` 是一段發生衝突的區塊

檔案裡可能會有 "很多段" 這樣的標記

用來告訴我們發生衝突的兩個 patch 分別爲這個檔案修改了什麼內容

因此對 `num.txt` 來說，P4 沒有修改東西，但是 P6 除了 "33" 之外，還多了 "55"

```
<<<<<<< HEAD (P4)
```

>

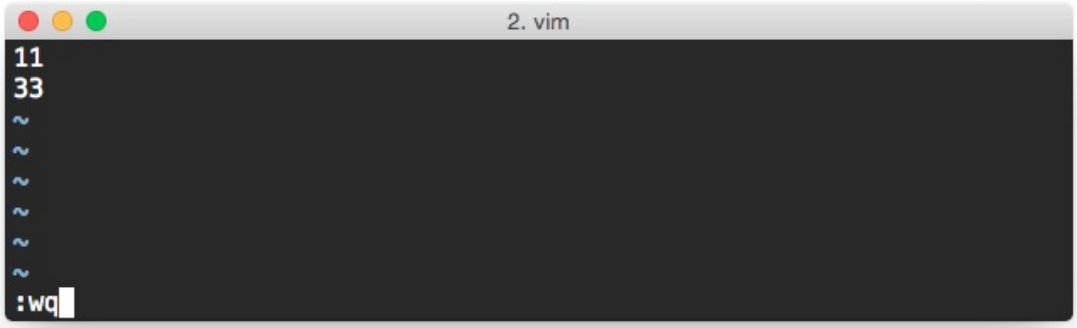
```
=======
```

> *33*
> *55*

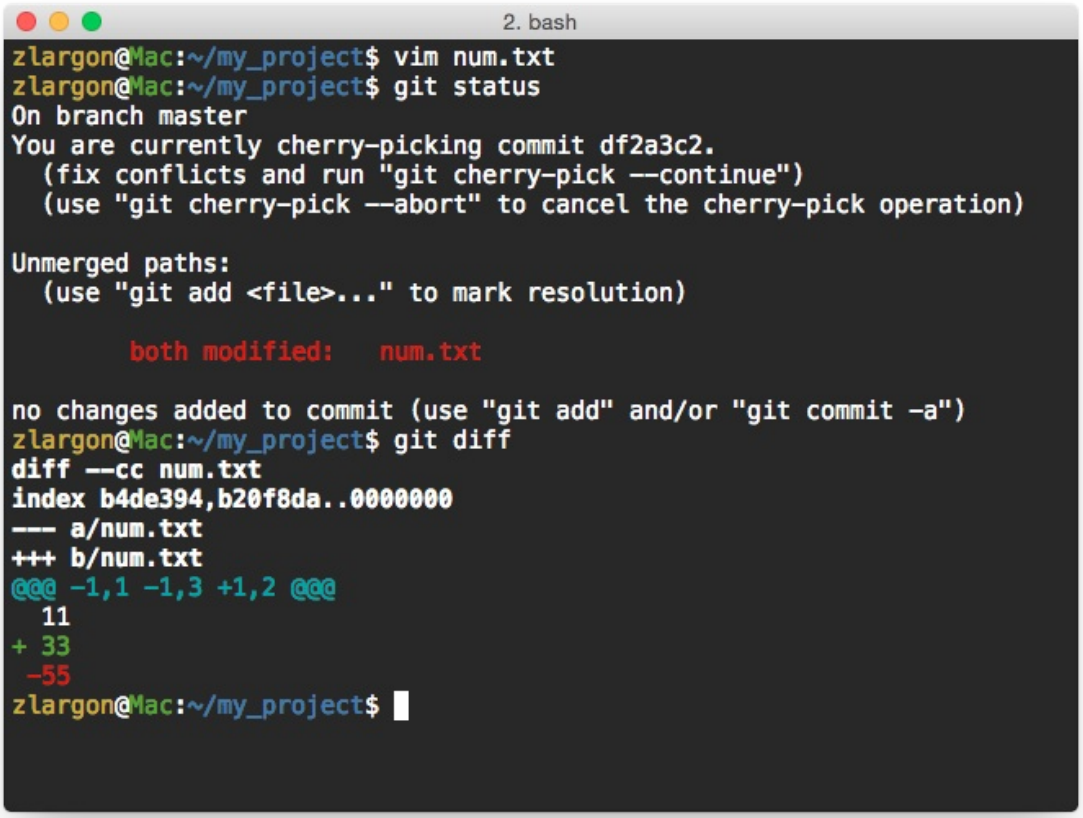
```
>>>>>>> Cherry-pick P6
```

我們在這個 patch 要做的事情只有新增 "33"，我們並不想要提交 "55" 的部分

因此我們就把 "55" 這一行刪掉，以及 `<<<` , `===` , `>>>` 的符號也一併刪掉，只保留 "11" 跟 "33"



存檔離開後，並用 `git diff` 觀察狀態



```
  11
+□33
□-55
```

如果我們仔細的觀察 `git diff`，會發現每一行前面的正負號看起來怪怪的

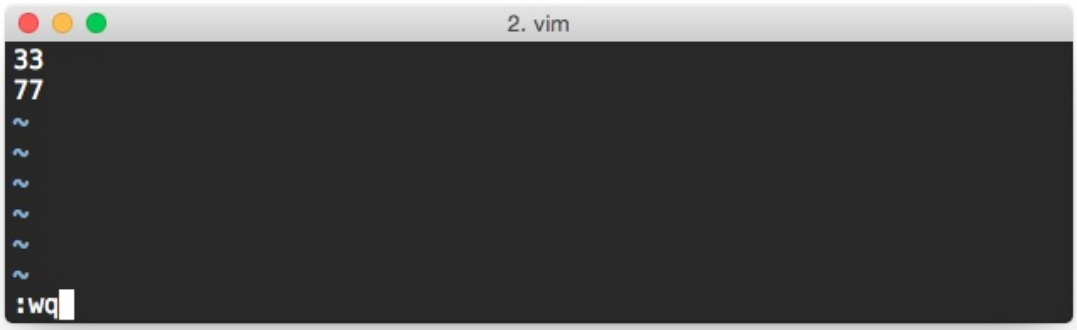
正負號有的在前，有的在後，這是爲什麼呢？

前面的正負號表示，我們解完 ***conflict*** 後的 ***patch*** 與 patch_1 的差別；而這裡的 patch_1 就是 P4（HEAD）

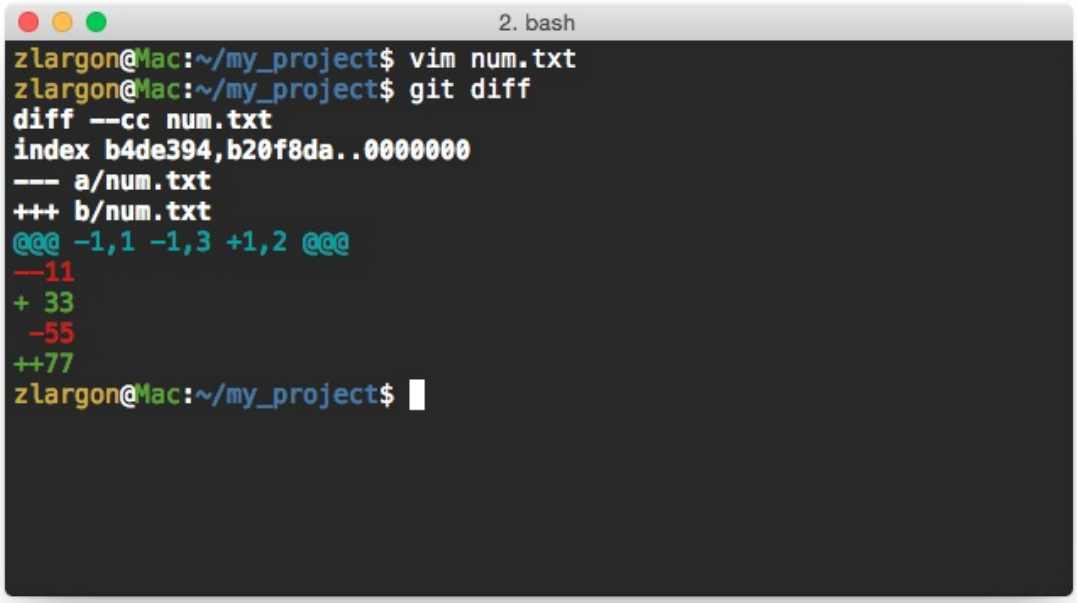
後面的正負號表示，我們解完 ***conflict*** 後的 ***patch*** 與 patch_2 的差別；而這裡的 patch_2 就是 P6

前（**P4**）	後（**P6**）	修改的內容	說明
		`11`	解完 ***conflict*** 後的 ***patch*** 與 P4 相比，沒有變化 解完 ***conflict*** 後的 ***patch*** 與 P6 相比，沒有變化
`+`		`33`	解完 ***conflict*** 後的 ***patch*** 與 P4 相比，多了這行 解完 ***conflict*** 後的 ***patch*** 與 P6 相比，沒有變化
	`-`	`55`	解完 ***conflict*** 後的 ***patch*** 與 P4 相比，沒有變化 解完 ***conflict*** 後的 ***patch*** 與 P6 相比，少了這行

如果我們這時候再用 vim 去修改 `num.txt`，把第一行的 "11" 刪除，並且新增一行 "77" 在最後面，我們來看看會發生什麼變化（稍等會把這個 patch 改回來）


2. vim
33
77
~
~
~
~
~
~
:wq

`git diff` 的結果

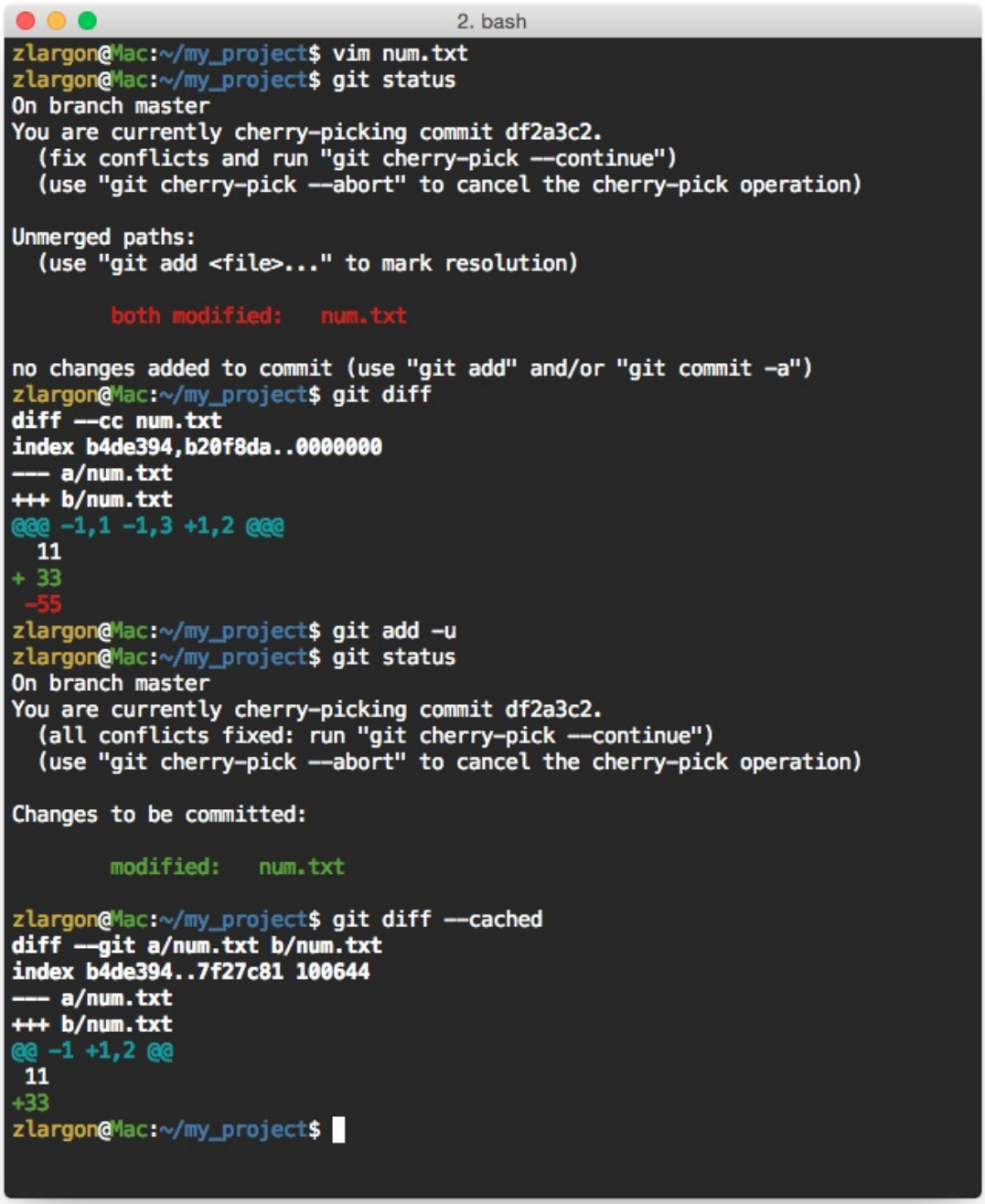

2. bash
zlargon@Mac:~/my_project$ vim num.txt
zlargon@Mac:~/my_project$ git diff
diff --cc num.txt
index b4de394,b20f8da..0000000
--- a/num.txt
+++ b/num.txt
@@@ -1,1 -1,3 +1,2 @@@
--11
+ 33
  -55
++77
zlargon@Mac:~/my_project$

前（**P4**）	後（**P6**）	修改的內容	說明
-	-	11	解完 ***conflict*** 後的 ***patch*** 與 P4 相比，少了這行 解完 ***conflict*** 後的 ***patch*** 與 P6 相比，少了這行
+		33	解完 ***conflict*** 後的 ***patch*** 與 P4 相比，多了這行 解完 ***conflict*** 後的 ***patch*** 與 P6 相比，沒有變化
	-	55	解完 ***conflict*** 後的 ***patch*** 與 P4 相比，沒有變化 解完 ***conflict*** 後的 ***patch*** 與 P6 相比，少了這行
+	+	77	解完 ***conflict*** 後的 ***patch*** 與 P4 相比，多了這行 解完 ***conflict*** 後的 ***patch*** 與 P6 相比，多了這行

這樣應該更清楚 `git diff` 的正負號要怎麼看了吧～

我們現在把內容改回來，並且用 `git add` 加入

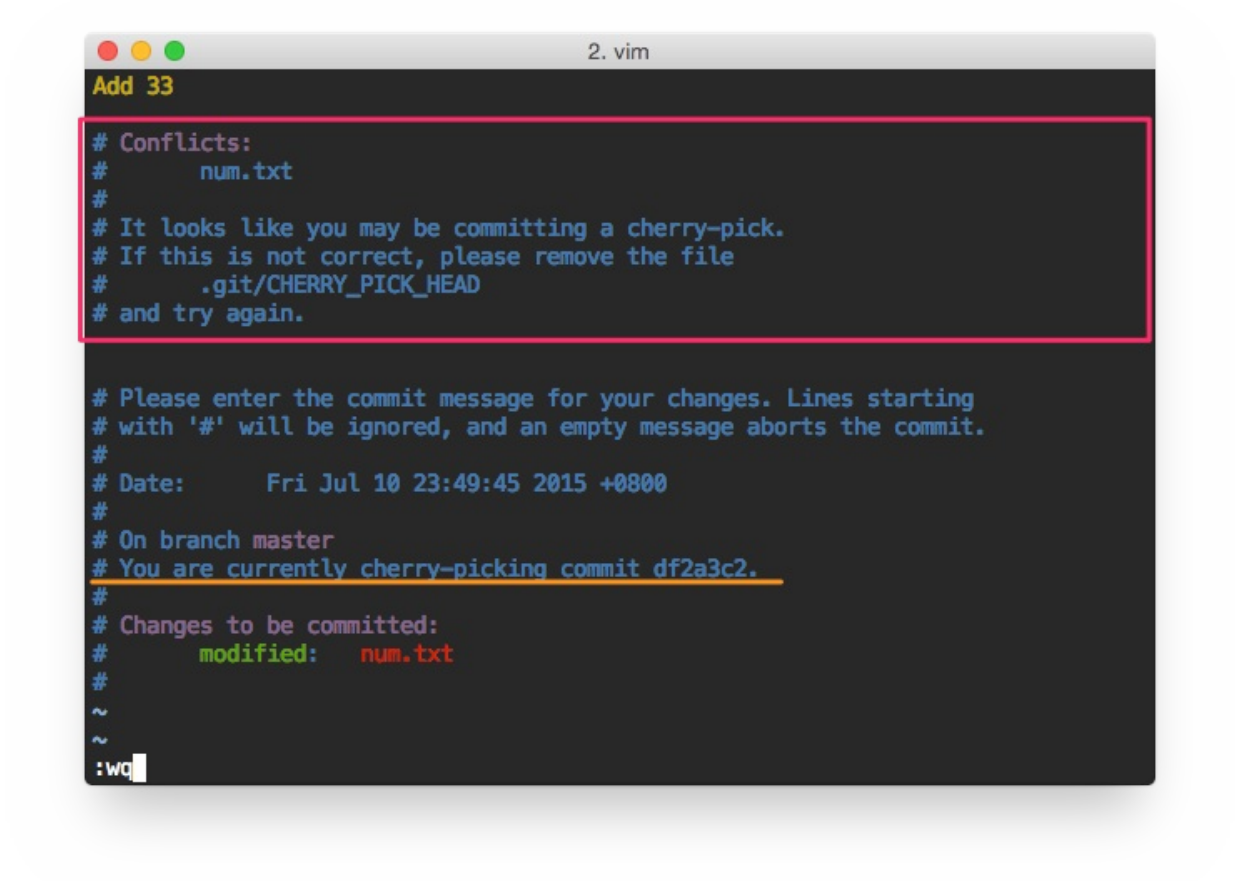
```
$ git status
$ git diff
$ git add -u            # 加入解完 conflict 的部分
$ git status
$ git diff --cached     # 查看這次修改的部分
```



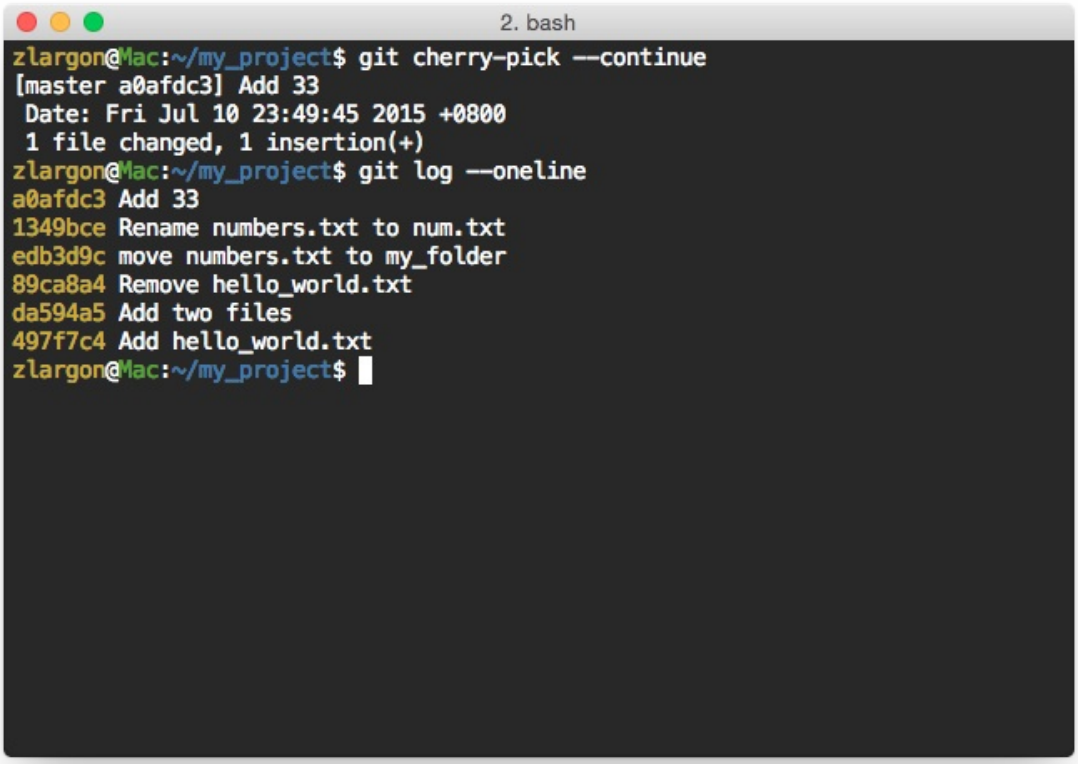
等待一切都準備就緒之後，就可以用 `git cherry-pick --continue` 來告訴 git，我們已經解完了，可以繼續了

```
$ git cherry-pick --continue        # 按下 enter 後會進入 vim 文字編輯模式
```

> 雖然這裡也可以使用指令 `git commit` 來告知 git 我們已經解完了，不過我不建議用這個指令
>
> 因爲除了 `cherry-pick` 之外，`rebase` 也有 `--continue` 的指令
>
> 但是 `git rebase --continue` 不等同於 `git commit`
>
> 因此我建議盡量使用 `git cherry-pick --continue` 比較不會造成混淆
>
> 另外一點是，如果使用 `git commit` 的時候，不小心在後面順手加上參數 `--amend` 的話，就會很悲劇...



存檔離開後，我們使用 `git log` 來查看結果

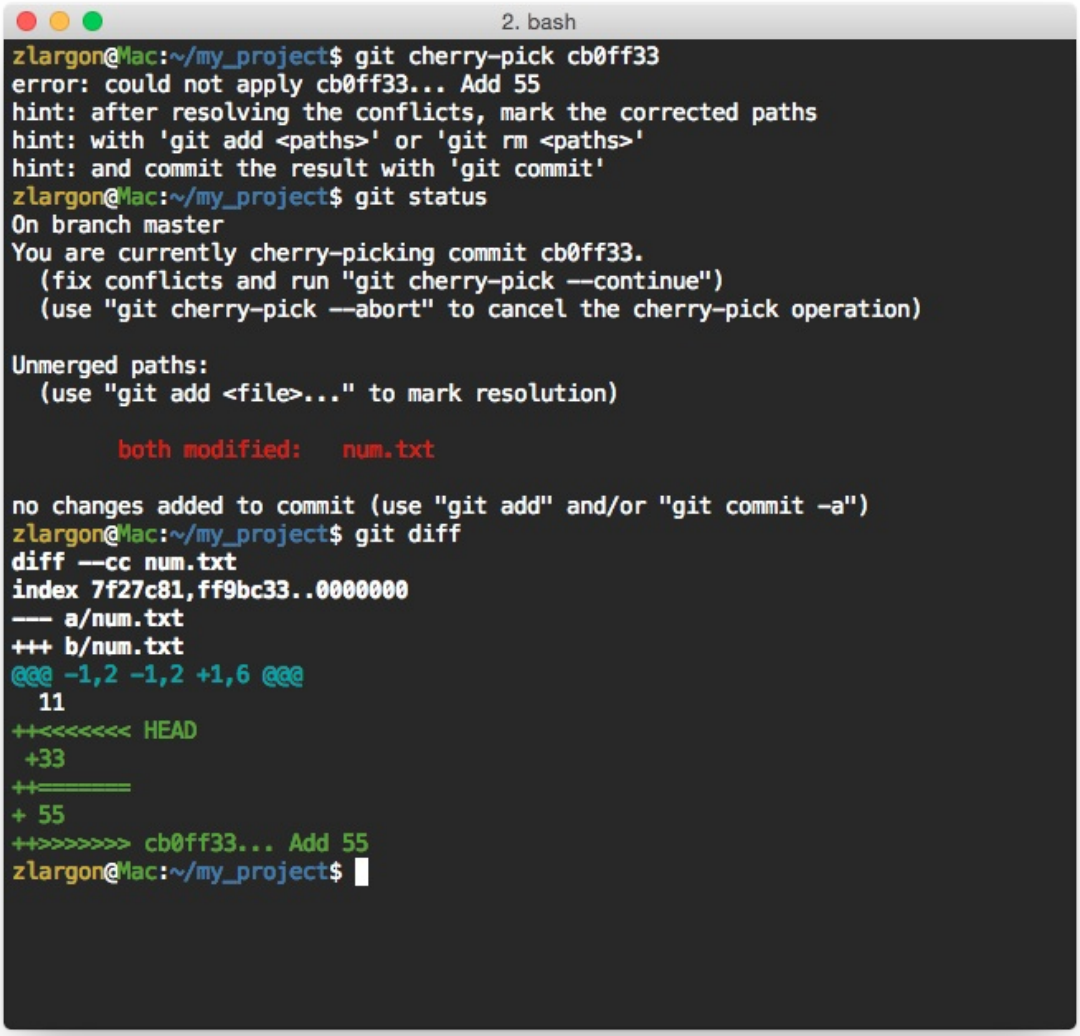



```
P0    →    ...    →    P4    →    P5    →    P6        原本的情況

P0    →    ...    →    P4    →    P6'                  現在的情況
```

我們已經成功把 P6 挑進來了

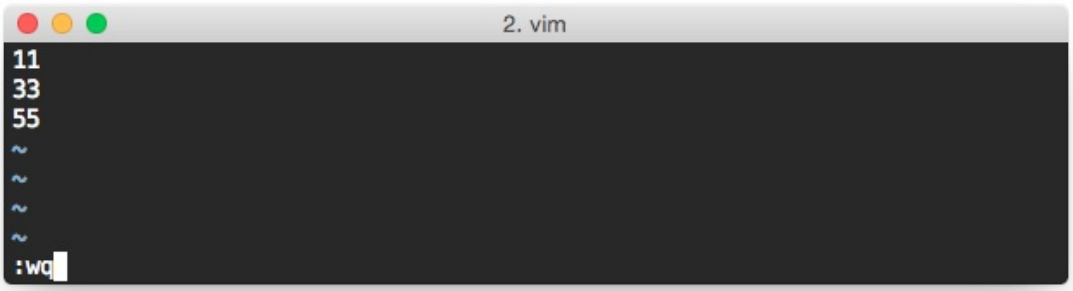
現在我們只要再把 P5 挑進來，就可以完成互換 P5, P6 了

```
$ git cherry-pick cb0ff33       # cherry-pick  P5  →  再度發生 conflict
$ git status                    # 查看檔案狀態
$ git diff                      # 查看檔案內容
```

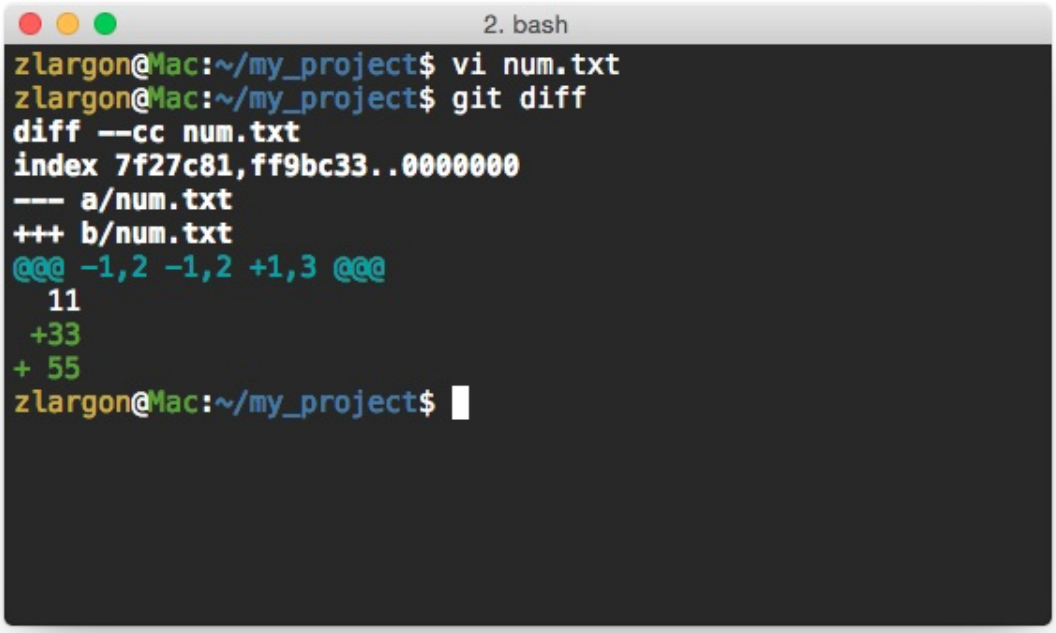


由於 P5 原本的 parent 是 P4，並沒有 "33"；而新的 P6 多了 "33"，因而發生了 ***conflict***

根據剛才解 ***conflict*** 的經驗，只要把 "33" 跟 "55" 都 `add` 進去就可以了



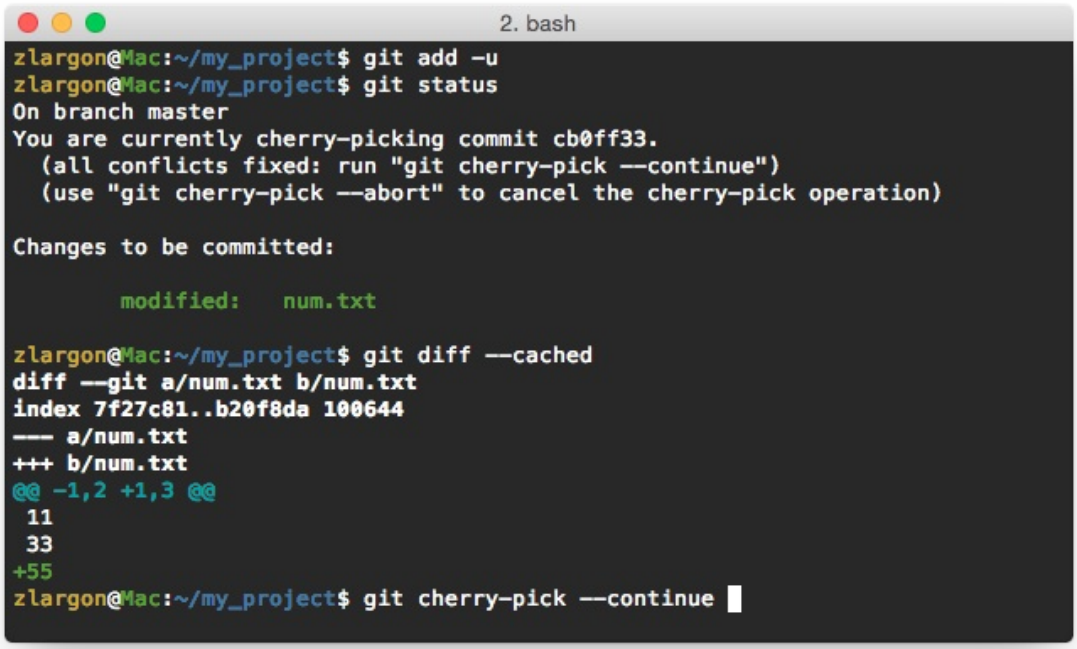
存檔離開，並用 `git diff` 來查看狀態



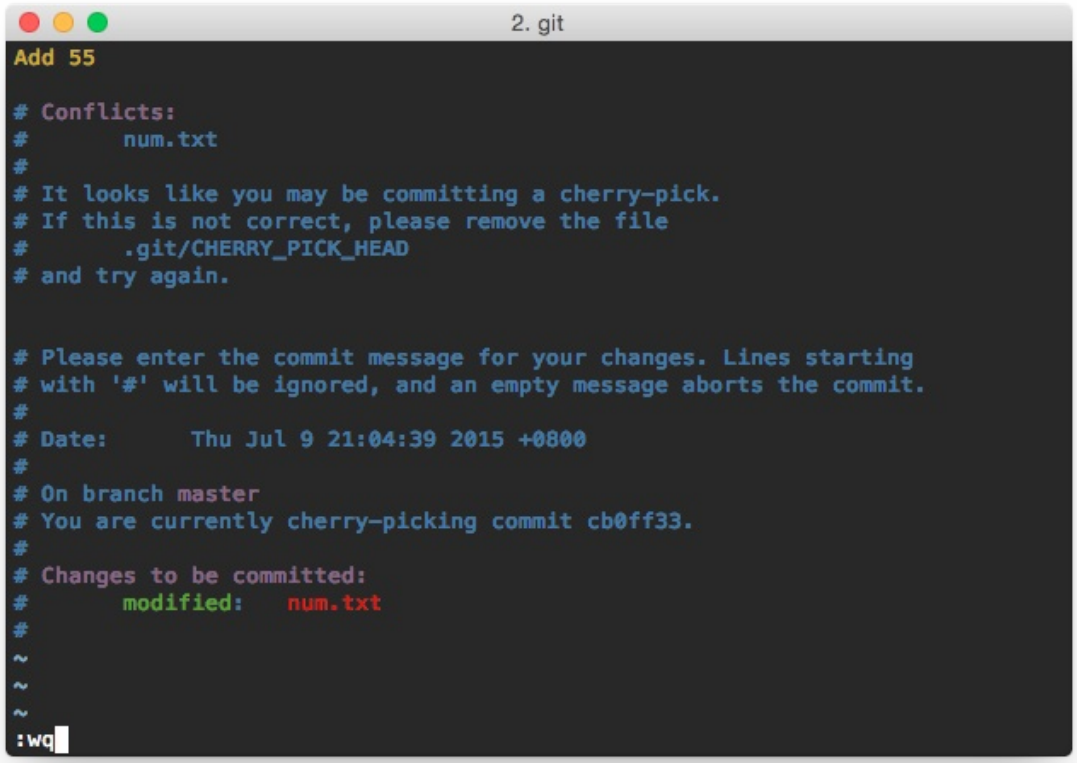
前 （**P6'**）	後 （**P5**）	修改的內容	說明
		`11`	解完 ***conflict*** 後的 ***patch*** 與 P6' 和 P5 都沒有改變
	`+`	`33`	解完 ***conflict*** 後的 ***patch*** 與 P5 相比，多了這行
`+`		`55`	解完 ***conflict*** 後的 ***patch*** 與 P6' 相比，多了這行

我們把解完的部分 `add` 起來

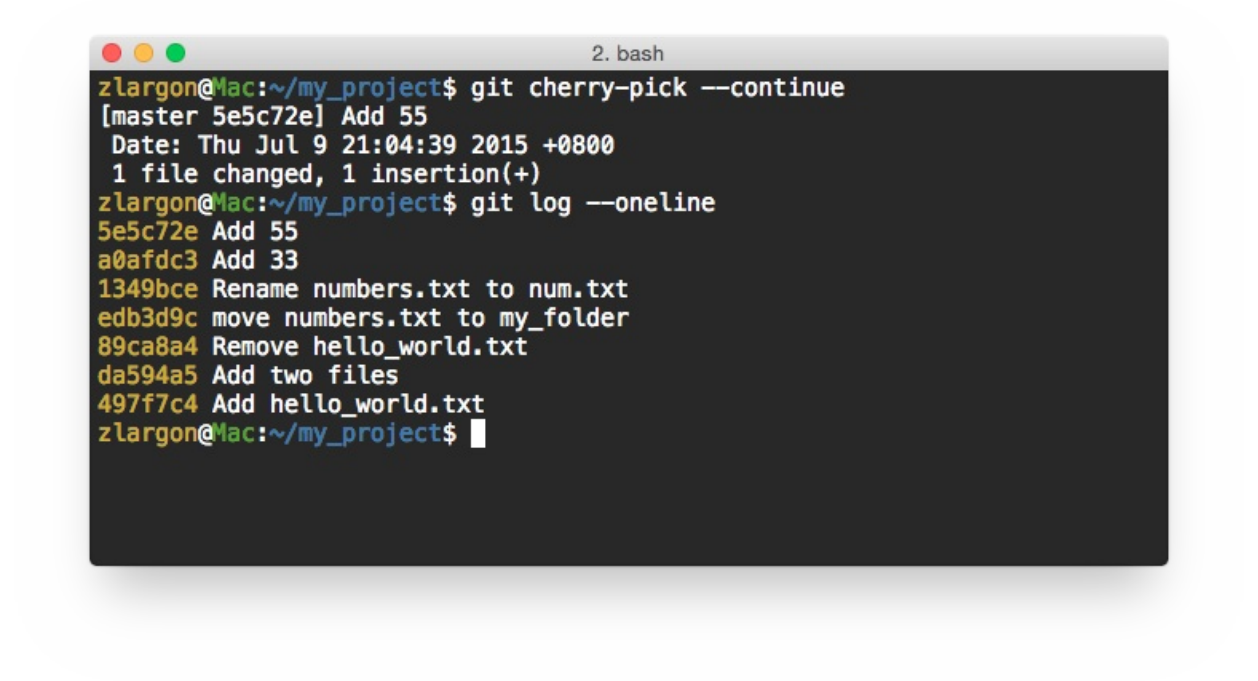
並且下指令 `git cherry-pick --continue` 告知 git 我們已經解完 ***conflict*** 了



進入 vim 文字編輯模式 `:wq` 存檔離開



使用 `git log` 查看，我們已經改好了，大功告成！




```
P0   →   ...   →   P4   →   P5   →   P6          原本的順序

P0   →   ...   →   P4   →   P6'  →   P5'         現在的順序
```

使用 `git cherry-pick --abort` 取消，並且回到 cherry-pick 前的狀態

如果你在解 ***conflict*** 的時候不太順利，或是不小心改錯了，想要全部重來

都可以用 `git cherry-pick --abort` 直接取消

如何降低發生 *Conflict* 的機會

解 ***Conflict*** 真的是一件很累人的事情，萬一沒解好還會後遺症一大堆

所以最好的辦法就是，盡量避免發生衝突的機會

在一開始講解 patch 的基本觀念時，有說到每個 patch 其實就是其相對於 parent 的變化量

因此，我們只要確保變化量越少，就越不容易發生 ***conflict***，也越容易被 git 拆解、合併

一般來說，發生衝突都是發生在多人共同開發的時候

我們自己寫的 code 通常不會跟自己發生衝突吧 XDD

除非像我這樣沒事亂搬 patch 的位置才有可能發生

舉例來說，Developer A 在一個 patch 裡面同時新增了 3 個 featrue，解了 5 個 bug

而 Developer B 也在同時新增了 3 個 feature，解了 4 個 bug，改了 3 個檔案的名稱

A 跟 B 可能同時去修改到同一個檔案，或同一個 function

或是 A 跟 B 剛好都發現了同一個 bug，同時都解了這個 bug 但是改法不同

然後更過分的是 B，還順便修改了檔案名稱和路徑！

這時候 A 跟 B 的 patch 會非常容易發生 ***conflict***

因此比較好的做法是，Developer A 跟 B 都把各自的 patch 拆開

Developer A：`featrue_patch * 3 + bug_fix_patch * 5`

Developer B：`featrue_patch * 3 + bug_fix_patch * 4 + move_file_patch * 3`

當所有的 patch 都拆成這小 patch 之後，要去挑 patch 就會變得很容易

每個 patch 的變化都減少了，即時發生了衝突也會很好解

因此請務必養成良好的上 code 習慣，每改好一個小功能就立即提交一個 patch

patch 多沒有關係，但 patch 改的內容要越細越好

我們之後還會陸續講到關於 ***Coding Style*** 以及上 patch 的「好習慣」

> 之前遇過最難解的 ***conflict*** 無非就是 iOS 的 `story board` 了
>
> `story board` 裡面一推由 Xcode 自動產生的 id，還眞的是不太好解...
>
> 解 ***confilct*** 的時間，都可以重拉一個新的了... Orz
>
> 因此建議還是同一個時間，只要有一個人去改 `story board` 就好了

本章回顧

- 看懂 ***conflict*** 檔案中 `<<<`，`===`，`>>>` 標記的意思
- 學會如何修改 ***conflict*** 的檔案
- 看懂 `git diff` `both modified` 檔案的時候，其每一行前面的正負號所表達的意思
- 使用 `git cherry-pick --continue` 告知 git 已經解完衝突
- 使用 `git cherry-pick --abort` 取消，並且回到 cherry-pick 前的狀態
- 如何降低發生 ***conflict*** 的機會

Rebase 版本衝突

我們在上一章已經教了如何解 `cherr-pick` 的版本衝突

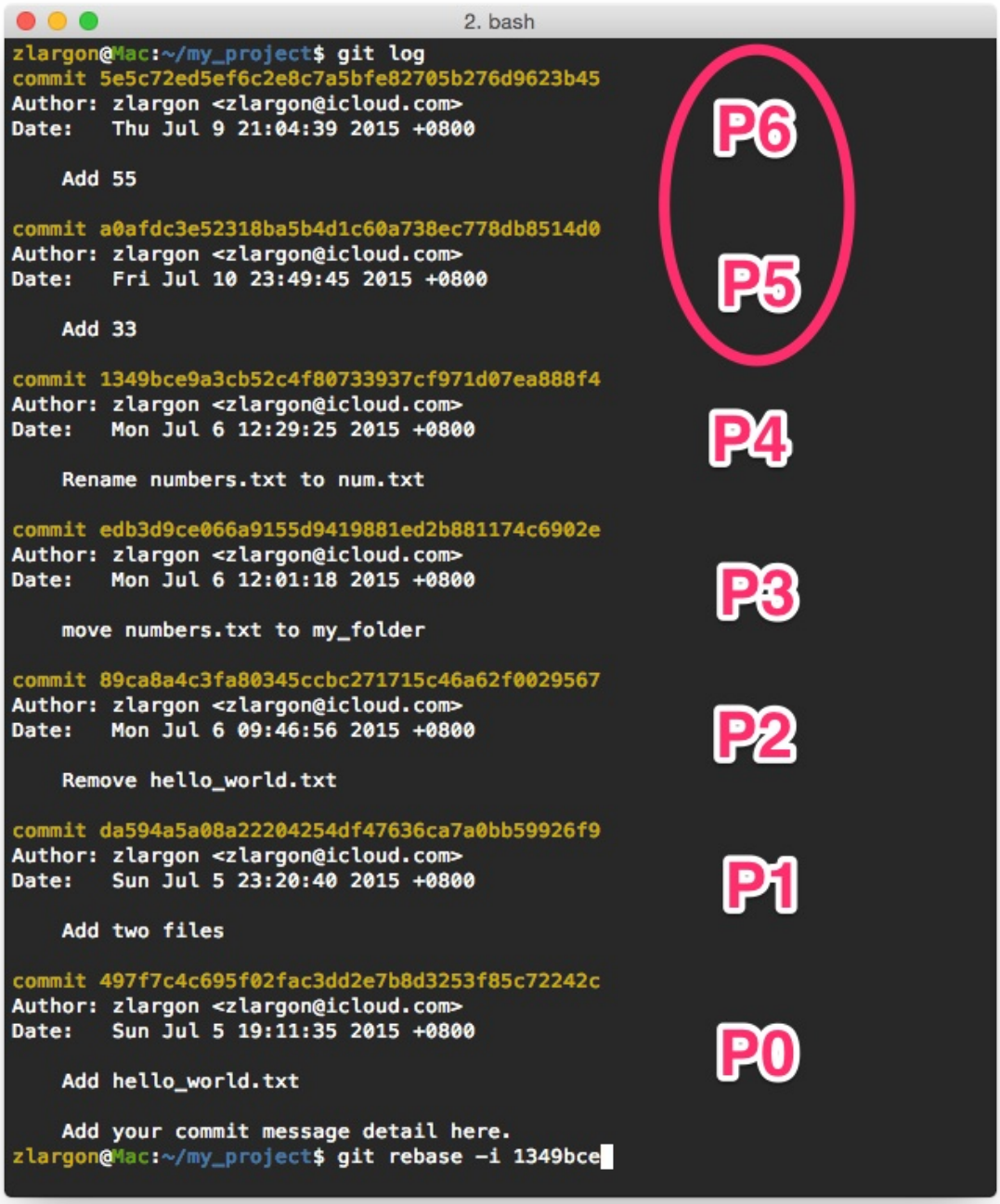
而 Rebase 互動模式其實就只是透過 `Rebase TODO` 依序完成多個 `cherry-pick` 的動作

因此，既然 `cherry-pick` 會發生版本衝突，那麼 `rebase` 必定也會發生版本衝突

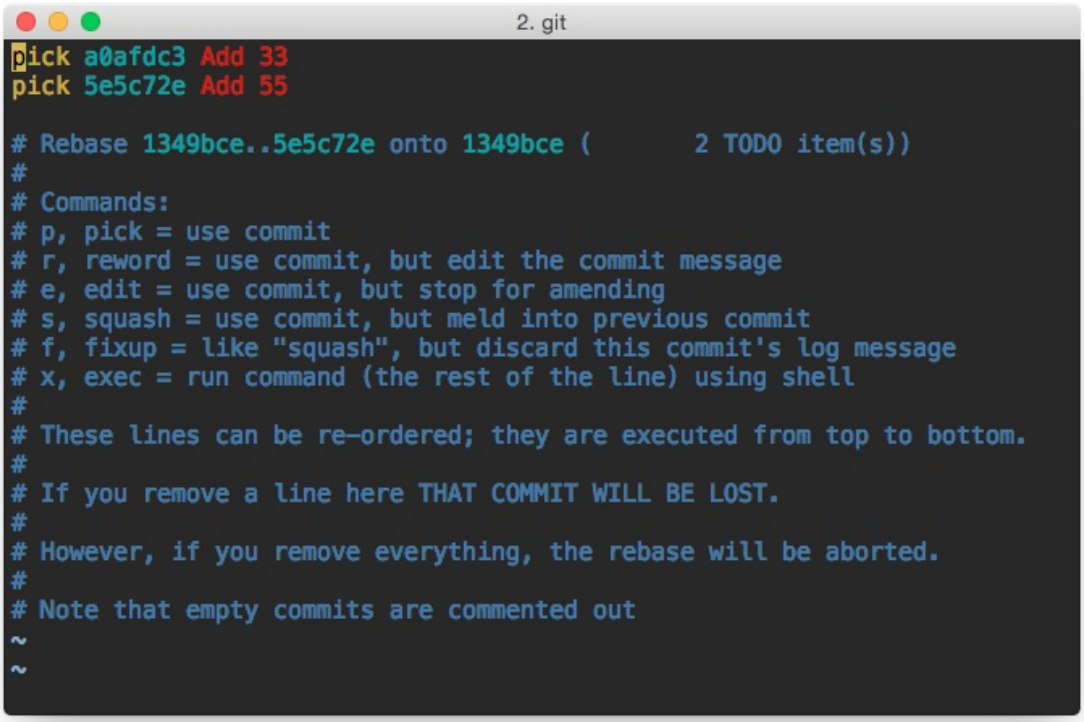
現在我們就把剛才最後的兩個 patch 用 Rebase 互動模式再調換回來吧！

改成先 "Add 55"，再 "Add 33"

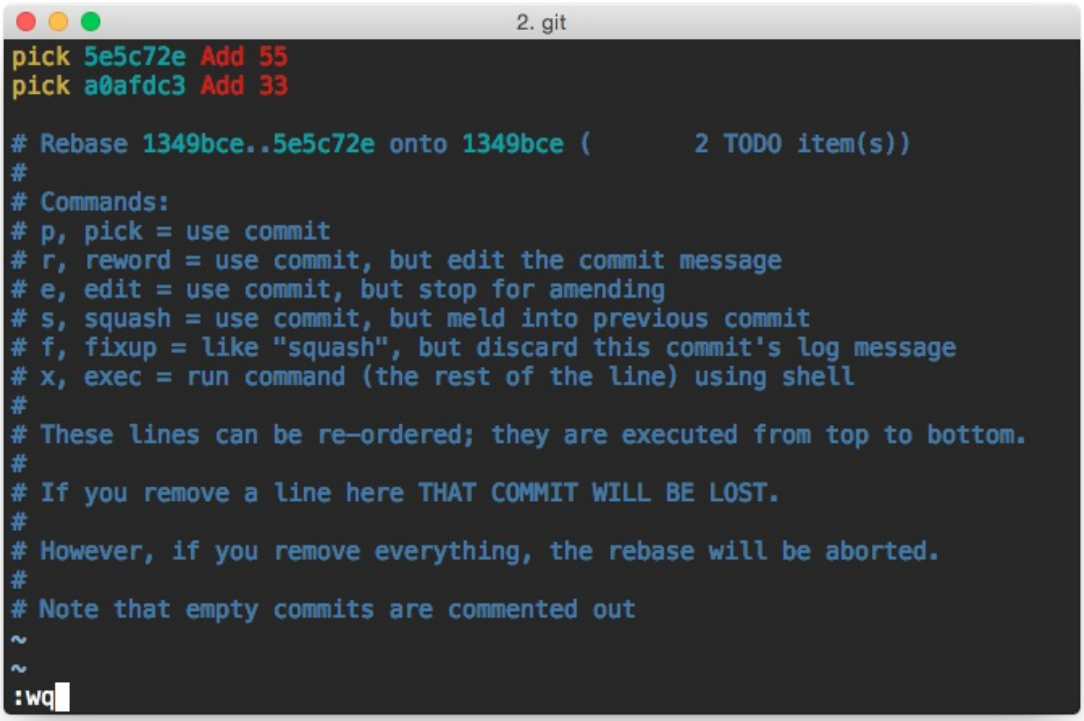
```
P0    →    ...    →    P4     →    P5     →    P6        現在的順序

                     1349bce     a0afdc3     5e5c72e

P0    →    ...    →    P4     →    P6'    →    P5'       我們將要改成的順序
```



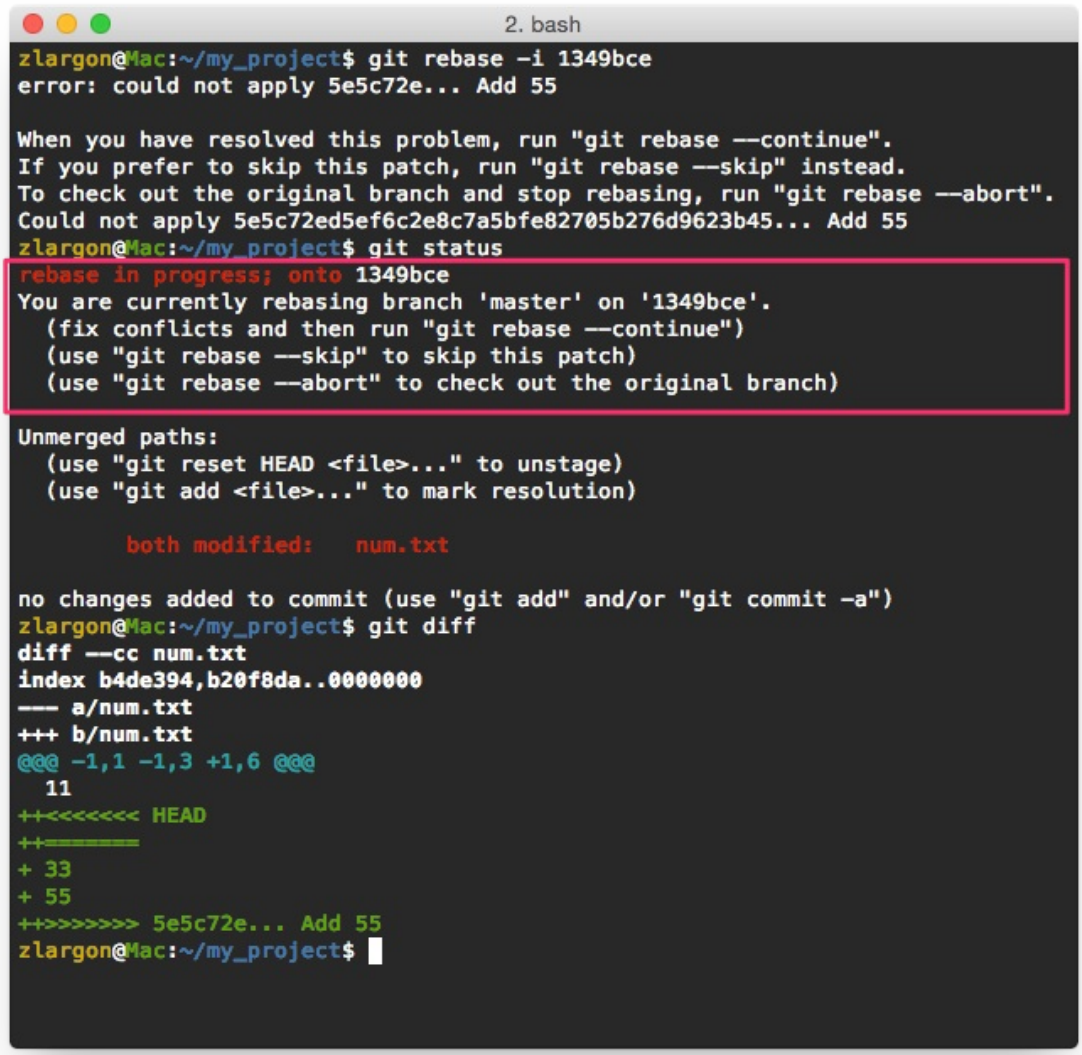

```
$ git rebase -i 1349bce    # 告訴 git 我們要改動的範圍是從 P4 之後的 patch (P5) 到 HEAD (P6)
                           # 並且進入 vim 文字編輯模式來編輯 Rebase TODO
```



我們把兩個 patch 的順序對調，並且存檔離開



git 在 `cherry-pick` P6 的時候發生了衝突



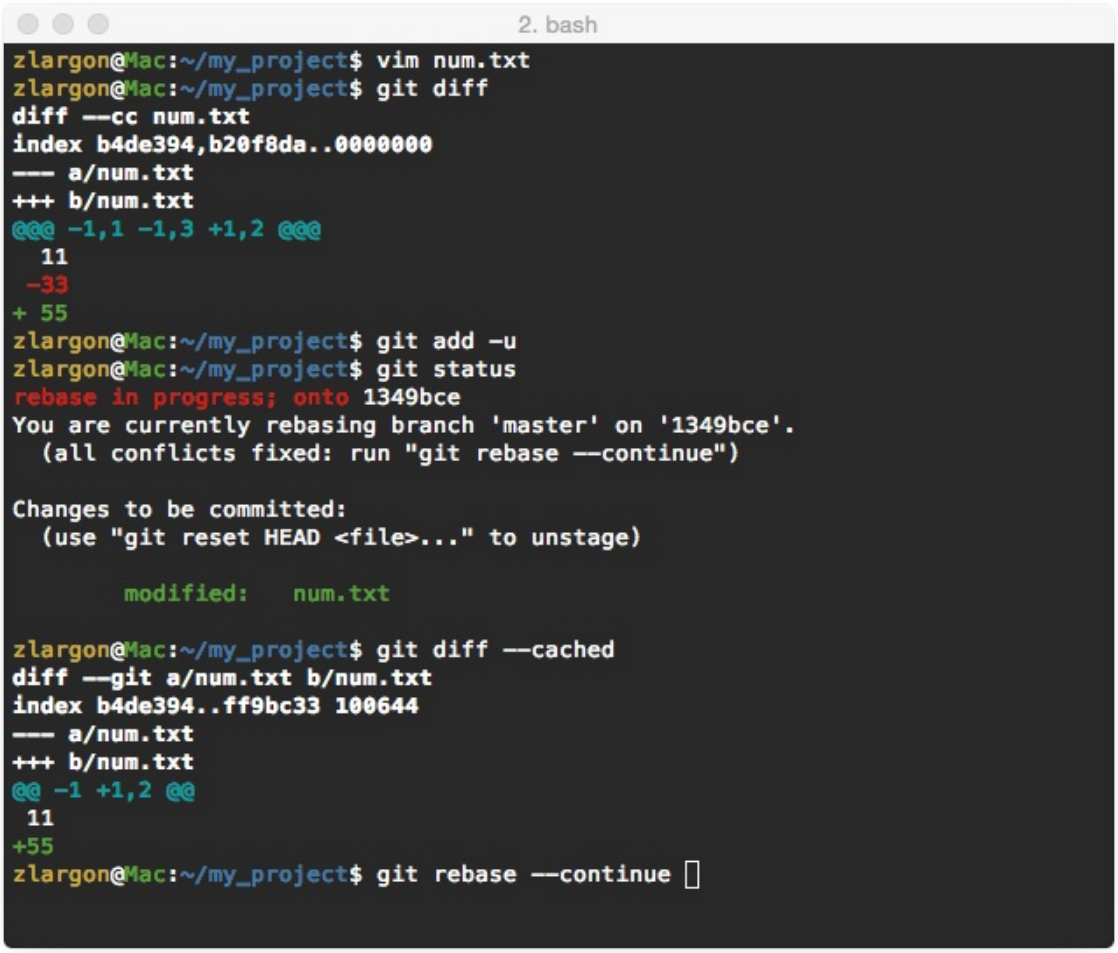
- ***fix conflicts and then run "git rebase --continue"***
- ***use "git rebase --skip" to skip this patch***
- ***use "git rebase --abort" to check out the original branch***

> 請在解完 conflict 後，下指令 `git rebase --continue`
> 使用 `git rebase --skip` 略過這個 patch
> 使用 `git rebase --abort` 全部取消，並且回到 rebase 前的狀態

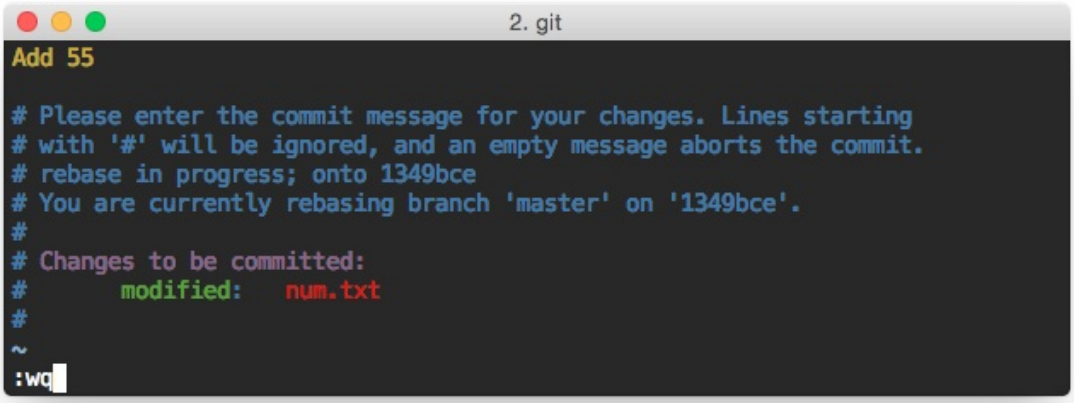
這時候 rebase 的動作會先停下來，並且要求我們解決衝突

我們保留 "55" 刪除 "33"

把 `num.txt` add 起來，下指令 `git rebase --continue` 告知 git 已經解完 ***conflict***，請繼續

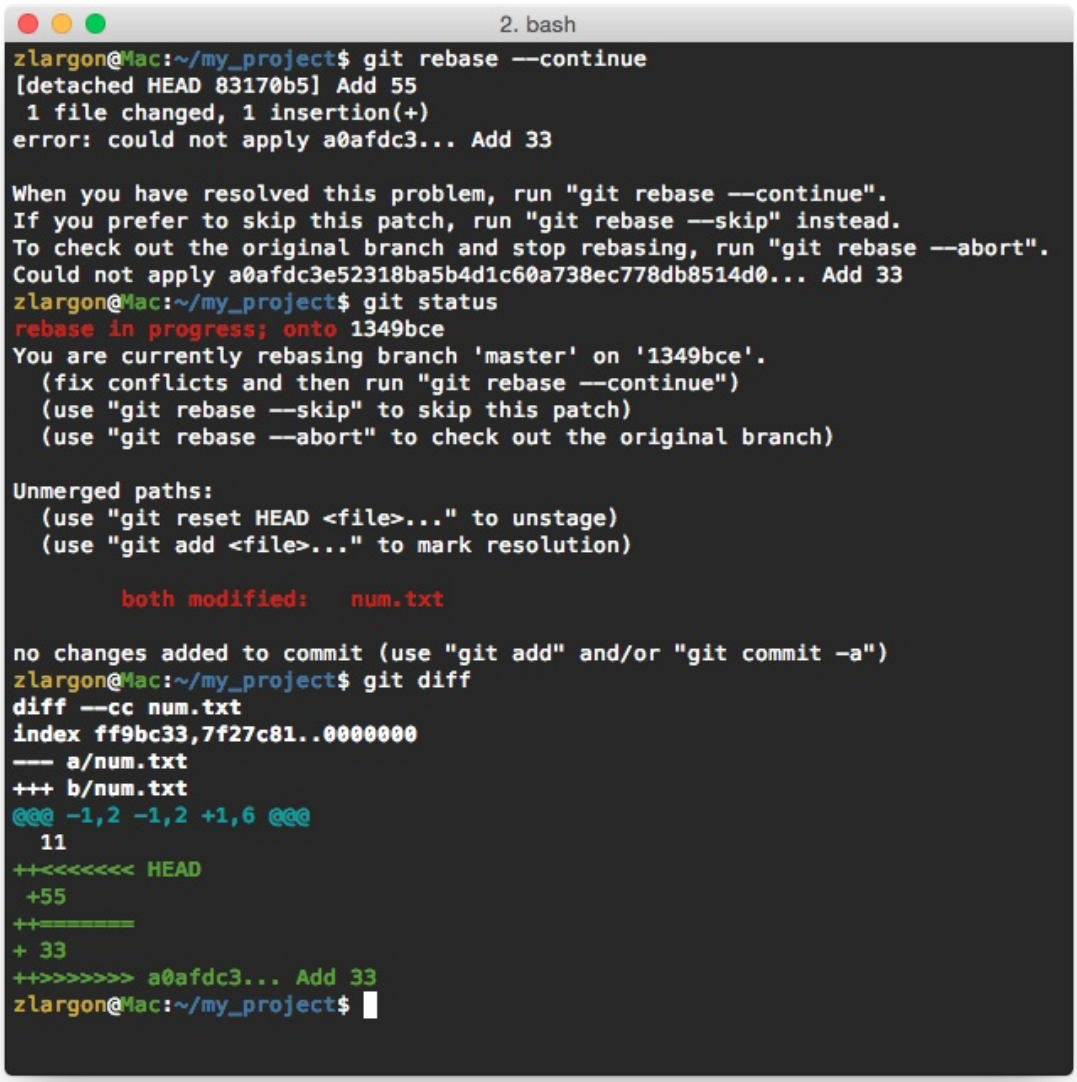



```
$ git rebase --continue      # 按下 enter 後，會立刻進入 vim 文字編輯模式
                             # 就跟 cherry-pick 的時候一樣
```



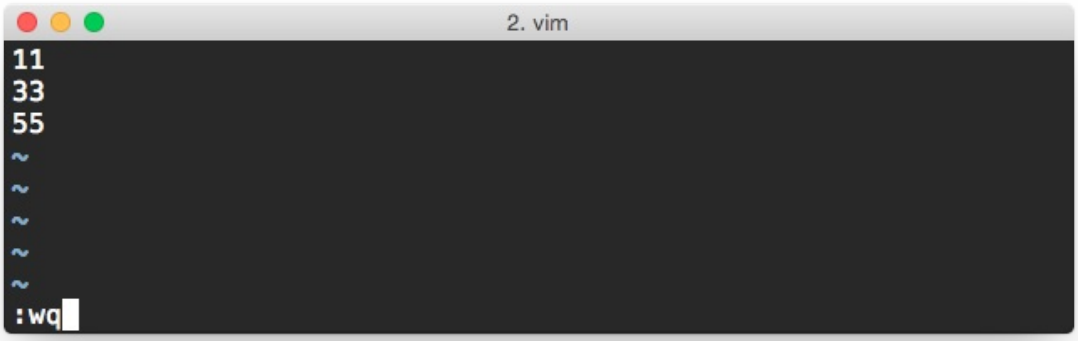
解完 P6 的 ***conflict*** 之後，git 會根據 `Rebase TODO` 繼續做下一件事情

接著會去做 `cherry-pick` P5，又再度發生了 ***conflict***

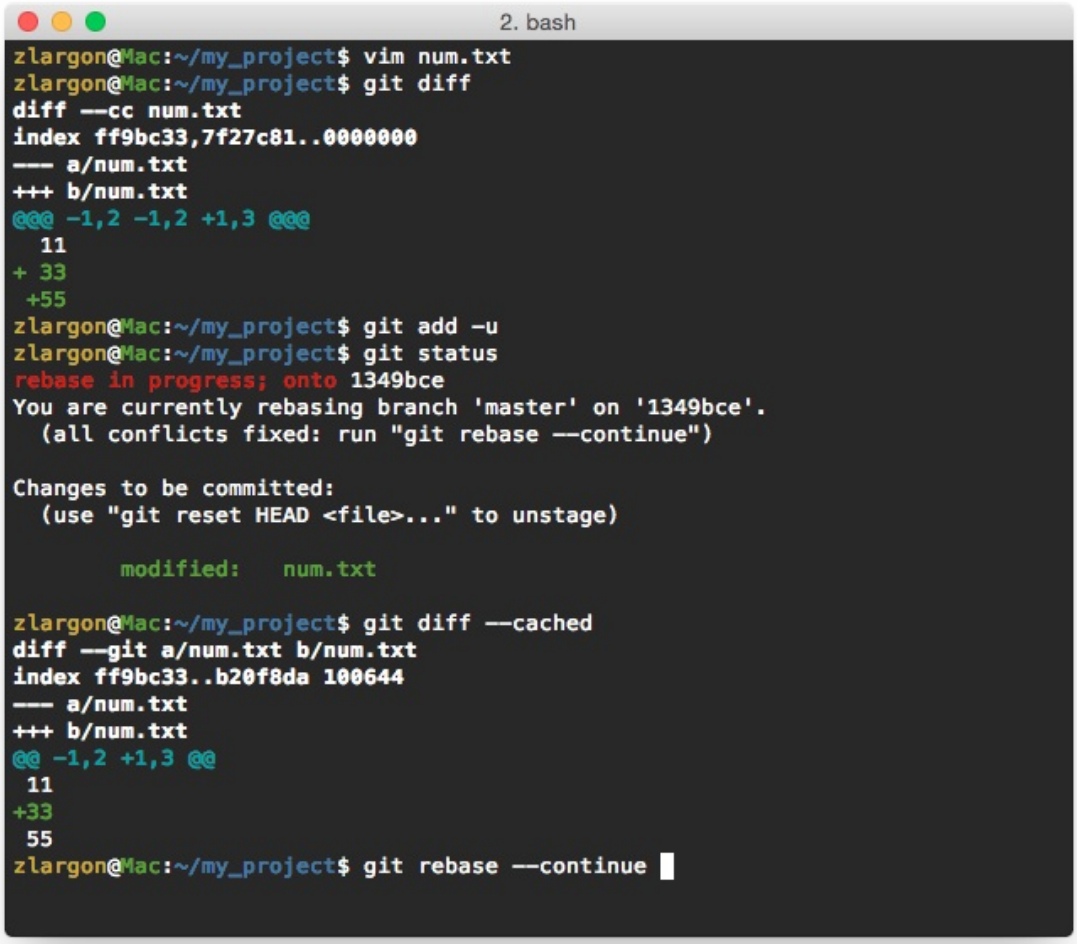


我們把 "33" 跟 "55" 都保留下來

這裡還要特別注意行序，我們想要順序是 11 → 33 → 55，而不是 11 → 55 → 33

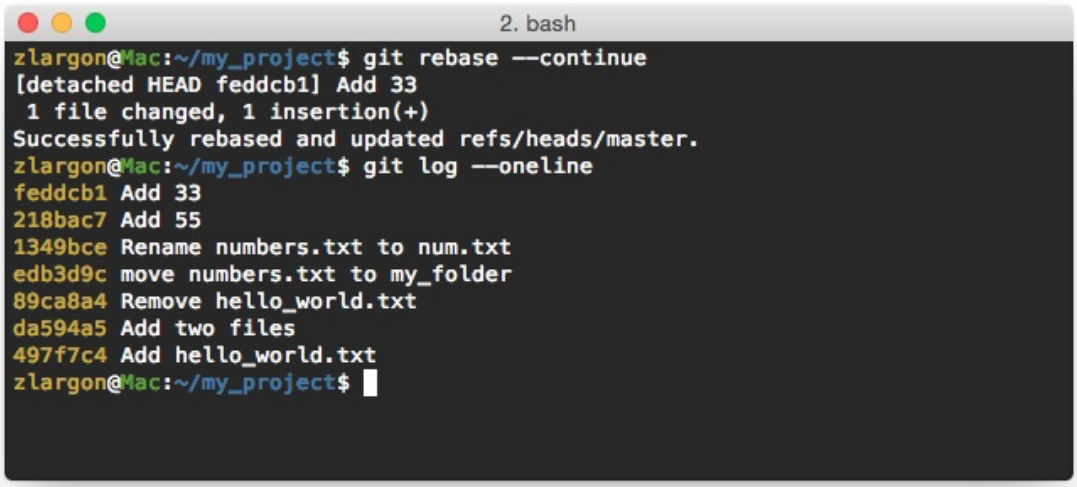


把 `num.txt` add 起來，下指令 `git rebase --continue` 告知 git 已經解完 ***conflict***，請繼續



進入 vim 文字編輯模式，存檔離開

使用 `git log` 查看結果



我們已經成功將 P5 跟 P6 的位置調換過來了

使用 `git rebase --skip` 跳過目前正在執行 `Rebase TODO` 的動作，並執行下一個動作

use "git rebase --skip" to skip this patch

Git 對 `--skip` 的說明爲 "略過這個 patch"

這對於 Rebase 互動模式來說，可以解釋爲跳過目前正在執行 `Rebase TODO` 的動作

因爲 `Rebase TODO` 的每個動作都會對應到一個 patch

git 跳過目前正在執行的動作後，會繼續執行下一個動作

使用 `git rebase --abort` 全部取消，並且回到 rebase 前的狀態

Rebase 互動模式讓我們快速的完成多個 `cherry-pick` 的指令

如果中途輸入 `git rebase --abort` 的話，就會全部取消，回到 rebase 前的狀態

包括中間好不容易解掉的 ***conflict*** 也要全部重來

如果想要保留之前解好的 ***conflict*** 並維持現狀的話，其實可以一直 `--skip` 到 `Rebase TODO` 全部跑完...

不過當遇到會連續發生 ***conflict*** 的狀況時，還是建議用 `cherry-pick` 一個一個挑會比較保險 ^^"

本章回顧

- 使用 `git rebase --continue` 告知 git 已經解完衝突
- 使用 `git rebase --skip` 跳過目前正在執行 `Rebase TODO` 的動作，並執行下一個動作
- 使用 `git rebase --abort` 全部取消，並且回到 rebase 前的狀態

Revert Patch

在 "移除單一個 Patch" 的時候，我們有學到如何移除一個 patch 或是調換 patch 的順序

必須使用 `git reset --hard` 回到特定的點上面，然後再用 `git cherry-pick` 把每個 patch 一個一個挑回來

或是我們也可以用 Rebase 互動模式這樣的高階的指令來達到相同的目的

但是很多時候，當我們發現之前某個 patch 有問題想要把它拿掉時，往往他的後面已經又上了 N 個 patch 了

如果我們只是爲了要把這個 patch 拿掉，卻要重新 `cherry-pick` 後面 N 個 patch

感覺改動會非常大，發生錯誤的風險也很高，不是很符合經濟效益

那有沒有什麼方法是，可以不太動之前的 patch，又可以讓我們把特定的 patch 拿掉的方法呢？

使用 `git revert <commit id>` 還原指定的 patch

這是我們目前的 patch 的情形，而我們想要把 P4 這個有 bug 的 patch 拿掉，這時候我們可以下指令 `git revert P4`

```
P0    →    ...    →    P4    →    P5    →    P6
                      (BUG)                (HEAD)
```

`revert` 的概念是在最後面再上一個 P4' 的 patch

把原本 P4 所做的事情，反向的再做一次

例如原本 P4 新增了一檔案，那個 P4' 就會刪除這個檔案

例如原本 P4 新增了一個 function，那個 P4' 就會刪除這個 function

因此 P4' 可以把 P4 所做的事情剛好抵消掉

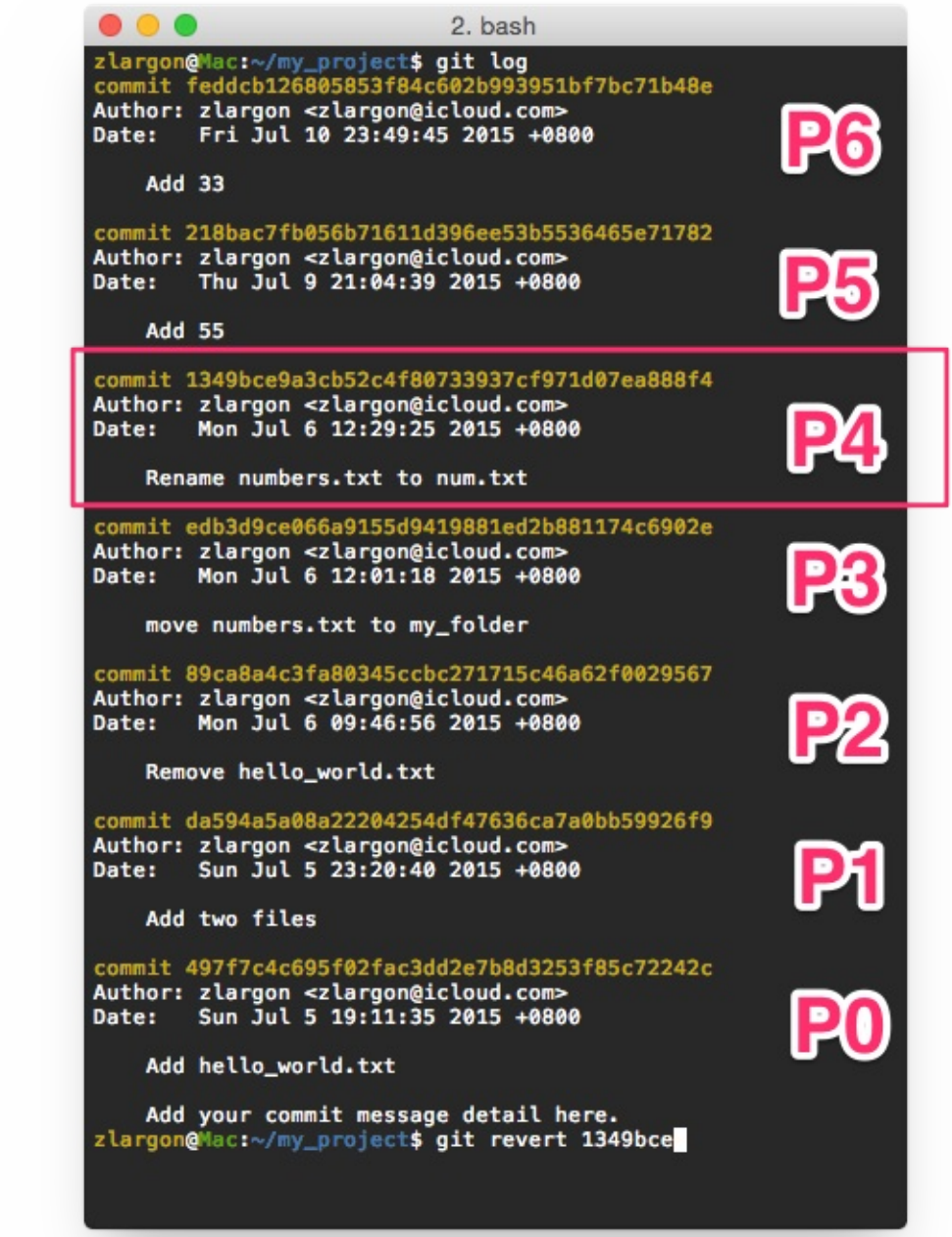
就像數學 `100 + (-100)` 的概念

```
P0    →    ...    →    P4    →    P5    →    P6    →    P4'
                       (+BUG)               (HEAD)      (-BUG)
```

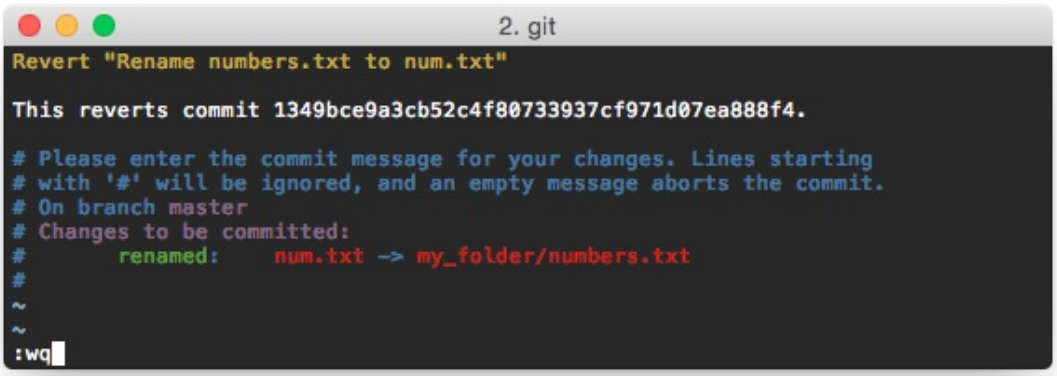
以先前的例子來說，如果我們要 revert P4

這個 patch 我們將 `my_folder/numbers.txt` 檔案改成 `num.txt`

revert 之後，他會把檔案再改回 `my_folder/numbers.txt`

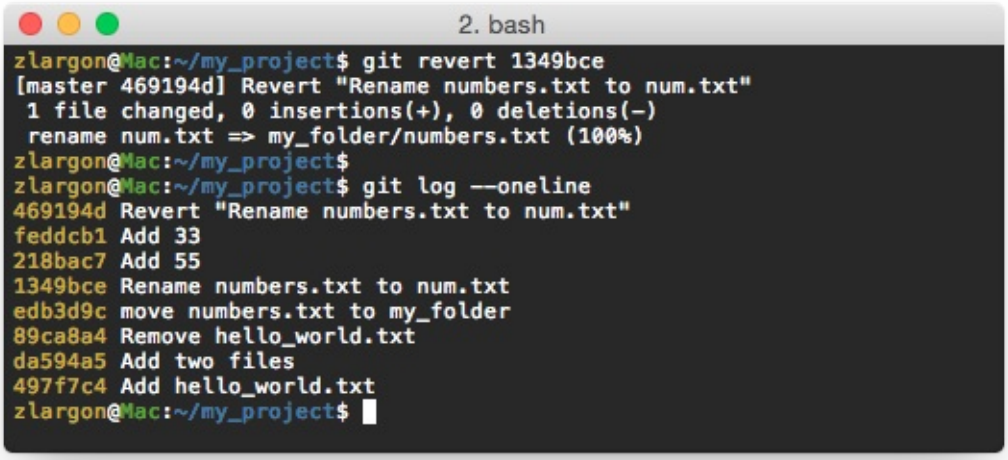


```
$ git revert 1349bce        # 按下 enter 就會進入 vim 文字編輯模式
```



git 會自動幫你把 commit title 跟 message 寫好

你可以在後面補充內容，或是直接存檔離開就ＯＫ了

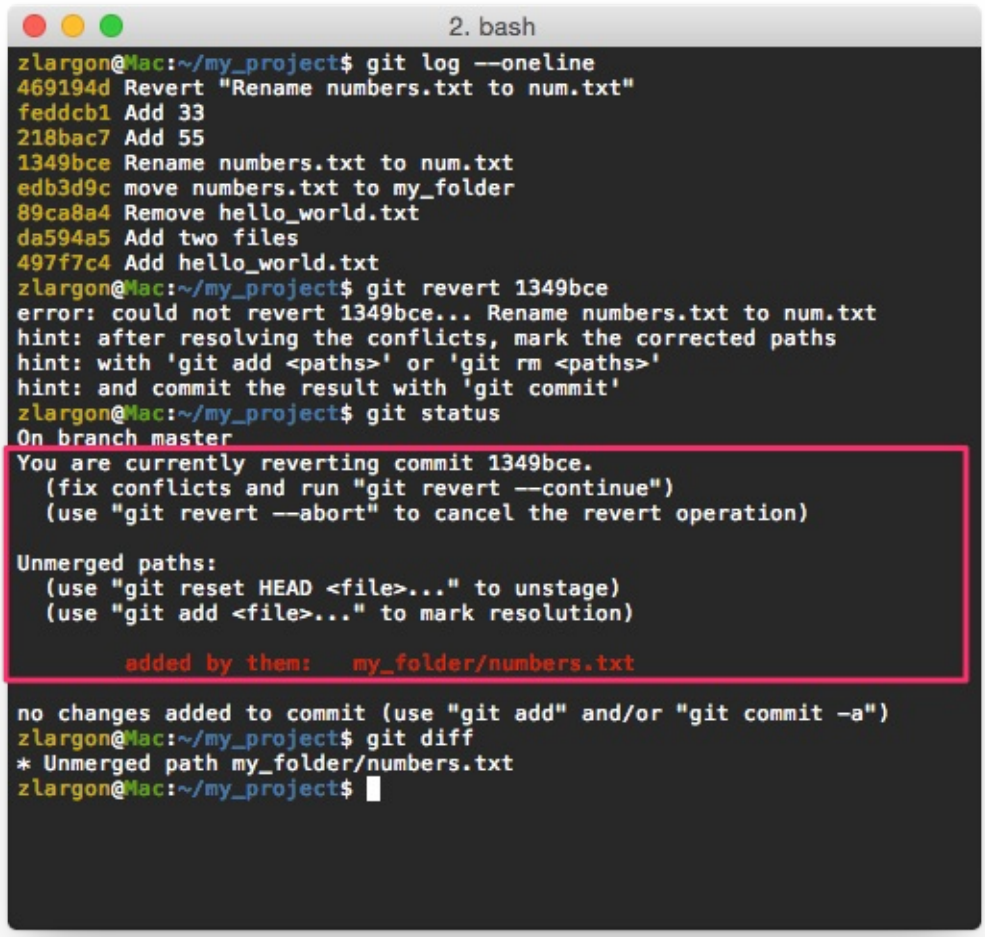


> 由於 revert 之後，git 等於是幫你再 commit 一個 patch
>
> 所以這個新的 patch 也可以被 revert 掉

Revert 版本衝突

`revert` 就跟 `cherry-pick` 和 `rebase` 一樣，都會遇到版本衝突的問題

例如，如果我已經把 P4 revert 掉了，現在我們又再對他做一次 revert，那會發生什麼事呢？



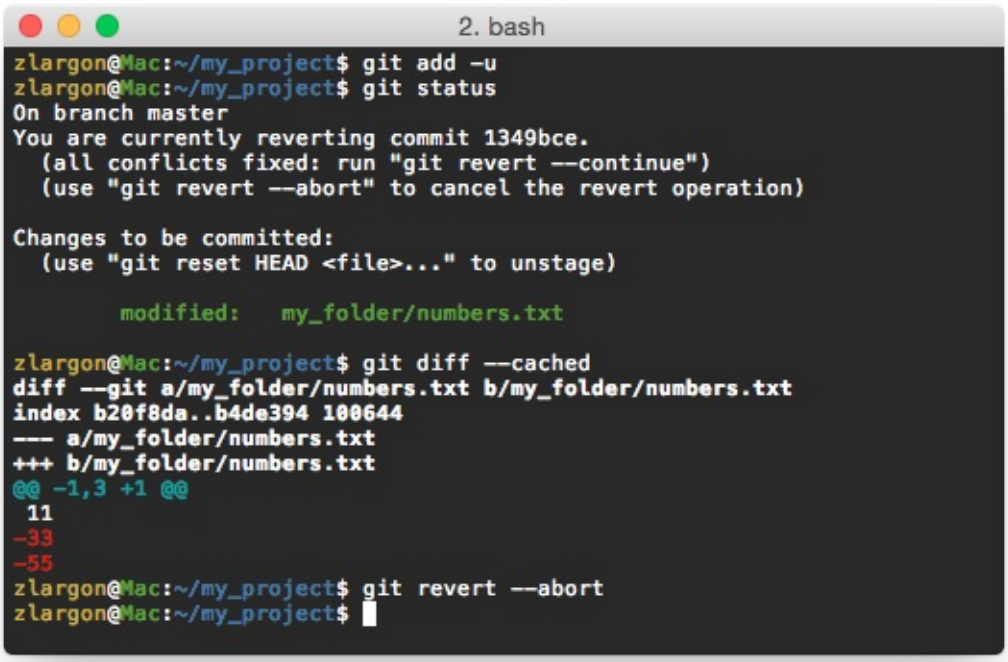
由於我們已經把 `num.txt` 檔案改回 `my_folder/numbers.txt`

而且我們在接下來的兩個 patch（P5, P6）又爲檔案新增了 "33" 跟 "55"

因此就發生 ***conflict*** 了

- ***fix conflicts and run "git revert --continue"***
- ***use "git revert --abort" to cancel the revert operation***

> 解完 ***conflict*** 之後，請下指令 `git revert --continue` 告知 git 已經解完衝突
> 如果想要放棄這次 revert，請下指令 `git revert --abort`



如果我們使用 `git add` 把他加進去的話，那麼 "33" 跟 "55" 就會被刪除

顯然這並不是我們想要的結果，因此我們使用 `--abort` 來放棄這次的 revert

本章回顧

- 使用 `git revert <commit id>` 還原指定的 patch
- 使用 `git revert --continue` 告知 git 已經解完衝突
- 使用 `git revert --abort` 來要放棄這次 revert

分支管理

查看分支

```
git branch
gitk --all &
```

建立 / 刪除分支

```
git branch <new branch name>
git checkout <branch name>
git checkout -b <new branch name>
git branch -f <branch name> <commit id>
git branch -D <branch name>
git log <branch name>
```

Commit Tree

```
git checkout <commit id>
git checkout -b <new branch name> <commit id>
```

Rebase 合併分支

```
git cherry-pick <commit 1> <commit 2> ...
git rebase <new base>
git rebase <new base> <branch name>
git reset --hard ORIG_HEAD
```

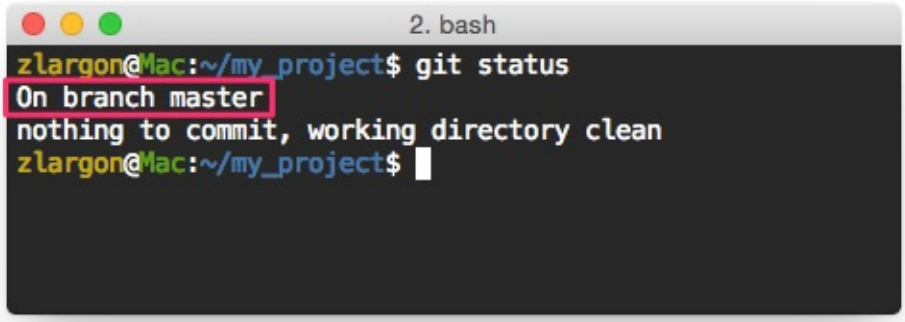
Merge 合併分支

```
git merge <branch name>
git merge --no-ff <branch name>
```

查看分支

每次我們下指令 `git status` 的時候，第一行總是會出現 `On branch master`

這是什麼意思呢？



其實一直以來，我們都是在一個叫 `master` 的分支上提交 patch

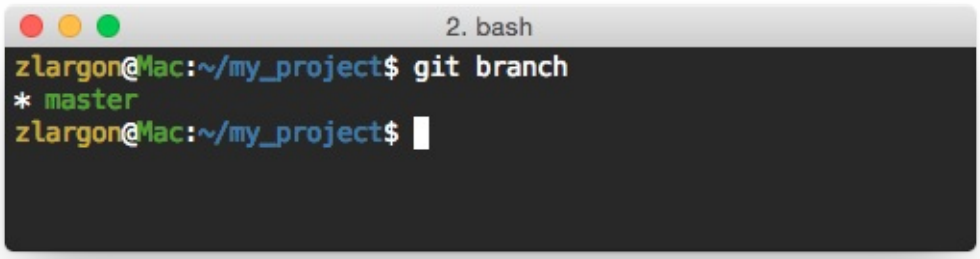
```
P0  →  P1  →  P2  →  P3  →  P4  →  P5          (master branch)
                                   (HEAD)
```

Git 習慣把 "主分支" 稱爲 `master`

當我們一開始使用 `git init` 來建立專案時，分支 `master` 就一起被創出來了

使用 `git branch` 查看所有的分支

```
$ git branch
```



這表示目前只有 `master` 一個分支

而 `master` 前面的 `*` 符號表示，我們現在就在這個分支上

使用 Gik 圖形化介面查看所有的分支

從這一章開始，我們會搭配 `gitk` 來輔助說明

`gitk` 是一個 git 圖形化介面的工具，可以讓我們清楚了解所有 patch 和分支的情況

他內建於 git 的安裝程式，且支援所有平台

所以當你安裝好 git 之後，`gitk` 也就安裝好了

在 git 的專案底下，輸入 `gitk --all` 即可啓動

後面帶 `--all` 才會顯示所有的分支，否則只會顯示目前的分支

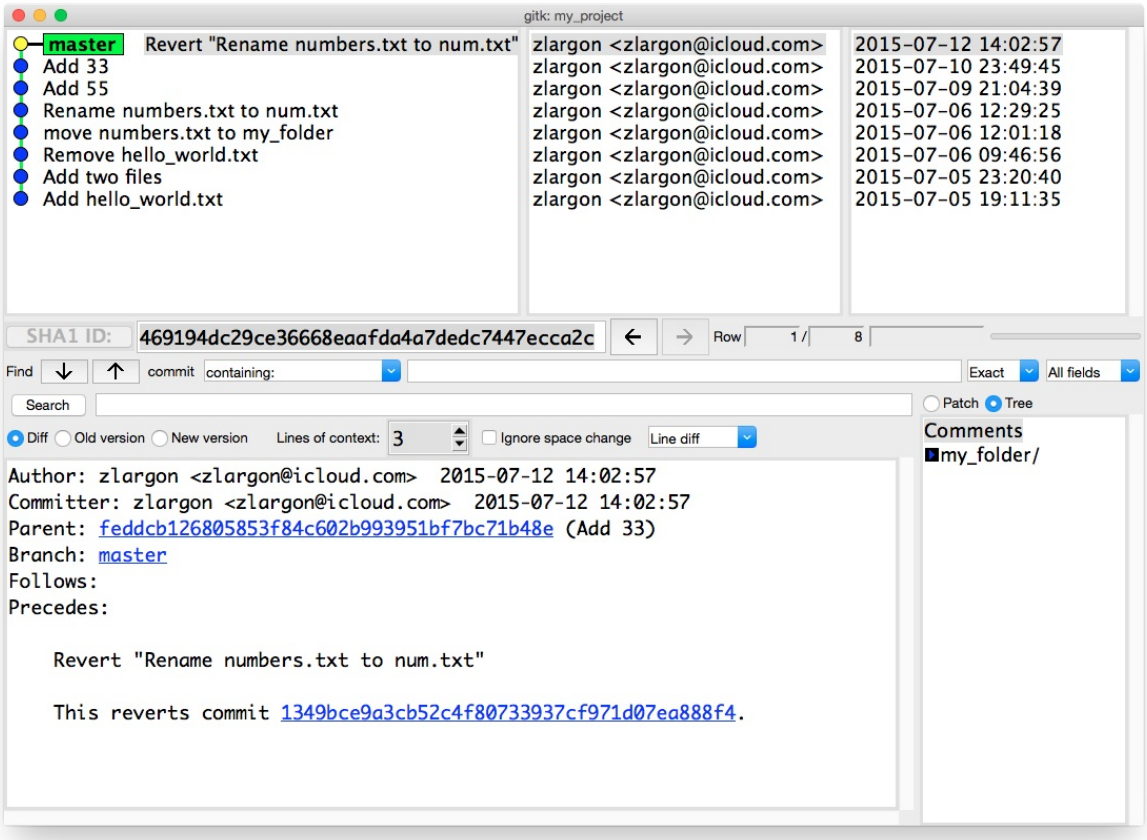
如果想要使 `gitk` 在背景執行的話，請記得在最後面輸入 `&`，這樣 `gitk` 才不會佔用終端機

```
$ gitk --all &      # 在背景執行 gitk
```

從 `gitk` 可以一目瞭然的看到所有 patch

- 黃色點點：
  表示我們目前所在的位置，也就是 `HEAD` 的位置
- 綠色框框：
  分支的名稱

> 由於黃點與 `master` 重疊，這表示我們目前位在 `master` 這個分支上



`gitk` 是一個非常陽春的圖形化介面，只能單純觀看所有分支的情況

而且每當我們下了 git 指令去修改後，`gitk` 也不會自動刷新，必須手動按 `F5` 更新

大家也可以上網找自己喜歡的 GUI 軟體，只要可以清楚觀看所有 patch 和分支情況就行了 ^_^

分支的用途

在 Git 的世界裡，我們可以把每個 patch 都看成是一個小節點，每個節點都可以另外長出自己分支

```
                              o---o---o feature
                             /
o---o---o---o---o---o---o---o---o---o---o  master
         \            /
          o---o---o---o  bugFix
                  \
                   o---o---o  testing
```

這種分支的模型，其實就是一個樹狀的結構，git 稱之爲 `Commit Tree`

Git 的分支模型非常的輕巧，可以讓我們很快速的建立 / 刪除分支

我們可以開一個叫做 `bugFix` 的分支，去修正某個 bug

這時 `master` 跟 `bugFix` 功能可以平行開發，不會互相影響

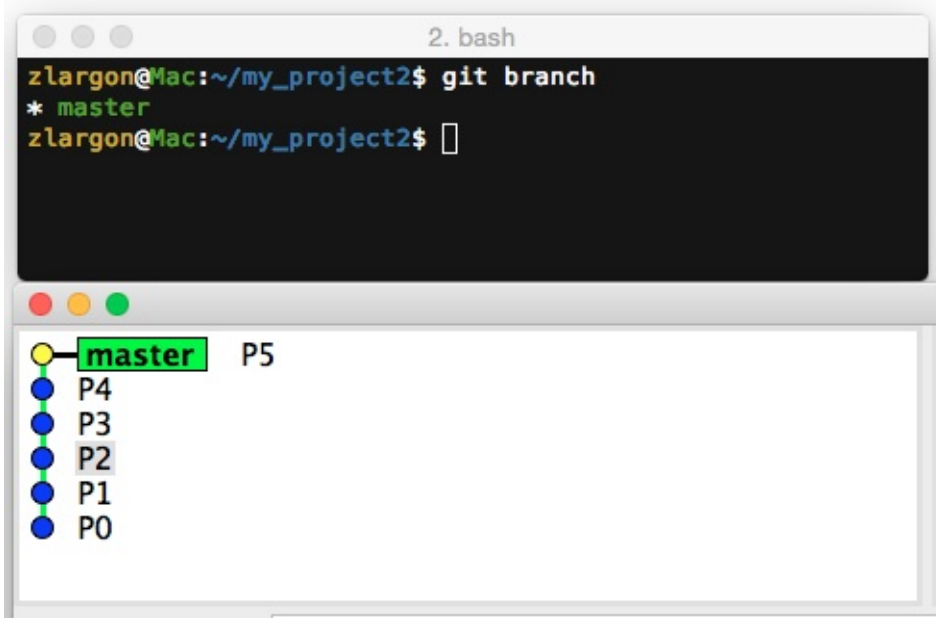
等 `bugFix` 修完測過沒問題之後，再把他合併回 `master` branch

當我們想要做一些簡單的測試時，也可以快速的建立一個測試用的分支，等測完之後再把 branch 砍掉

本章回顧

- 使用 `git branch` 查看所有的分支
- 使用 `gik --all &` 啓動圖形化介面

建立 / 刪除分支

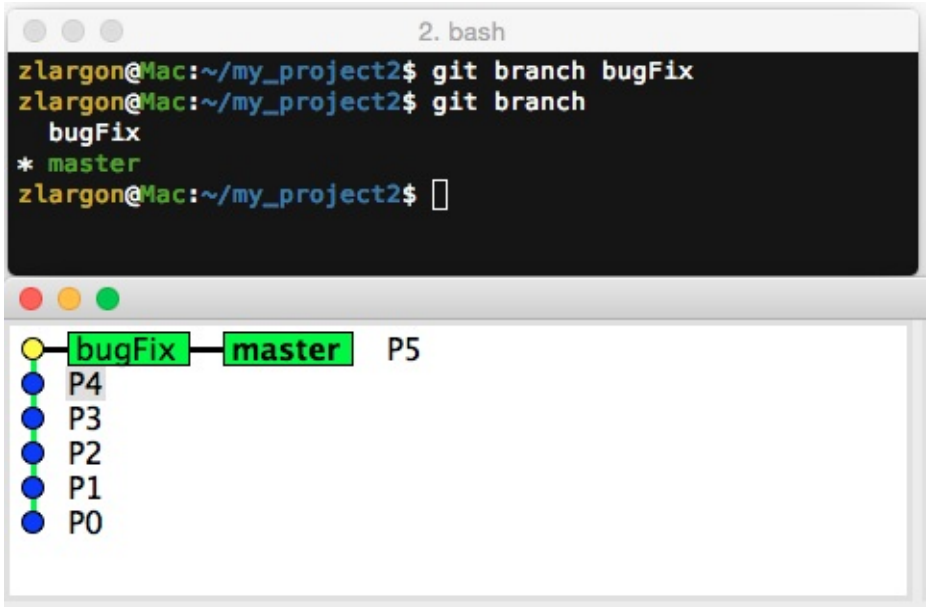


黃色點我們目前 patch 的位置，也就是 `HEAD`

而我們目前在 master 上

使用 `git branch <new branch name>` 建立新的分支

```
$ git branch bugFix         # 建立名爲 "bugFix" 的分支
$ git branch                # 查看所有的分支
```

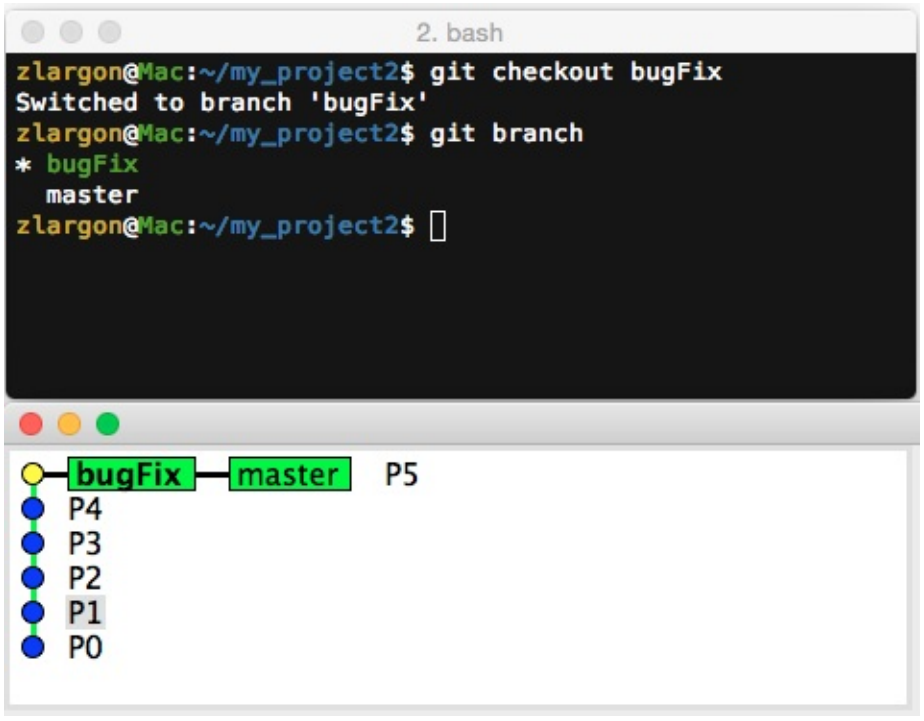


`git branch <new branch name>` 會根據目前所有的位置來建立分支

因此現在 `master` 和 `bugFix` 內容是完全一樣的

`gitk` 中綠色框框的 `master` 爲粗體，表示我們目前在 `master` 分支上

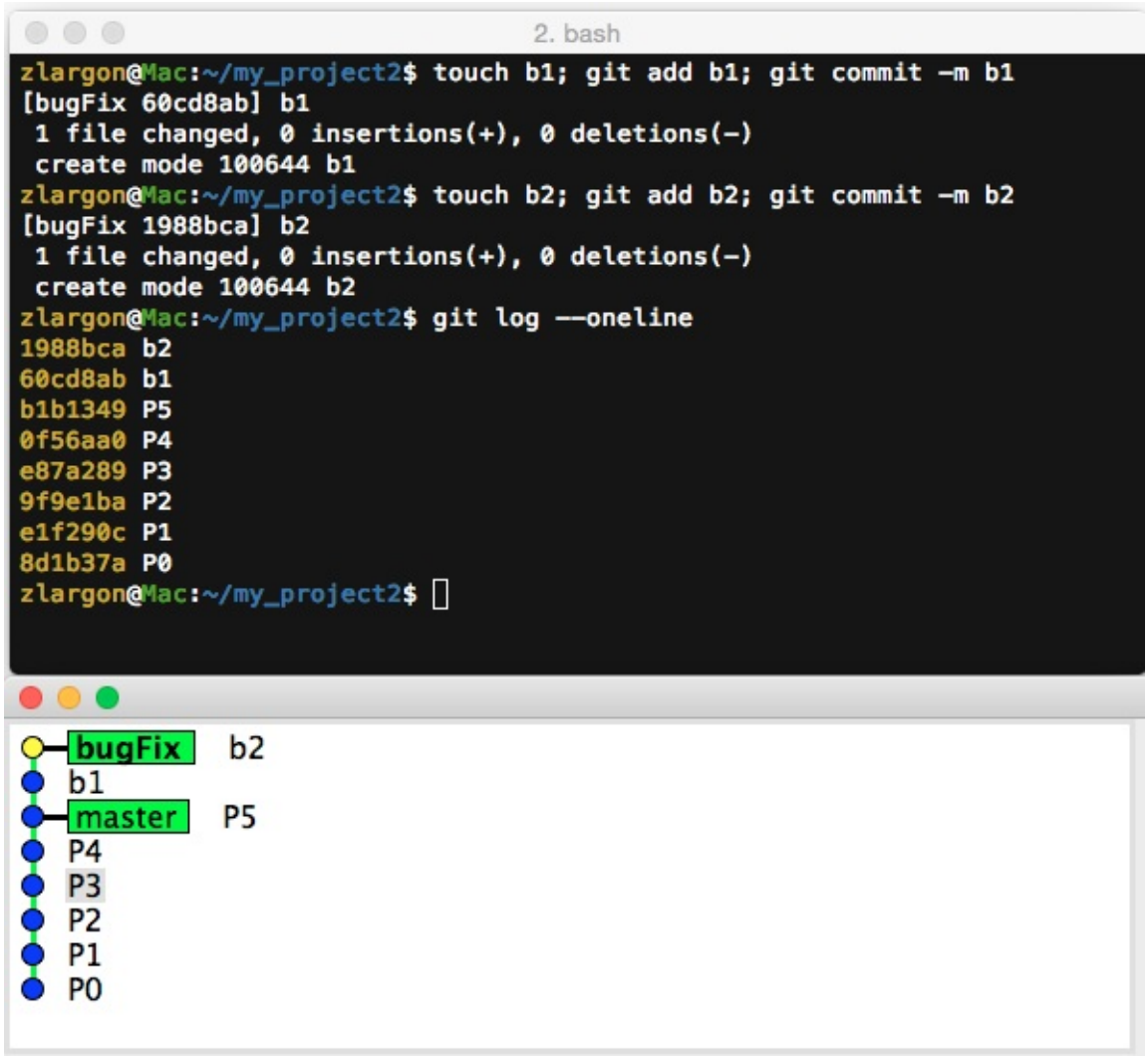
使用 `git checkout <branch name>` 切換分支



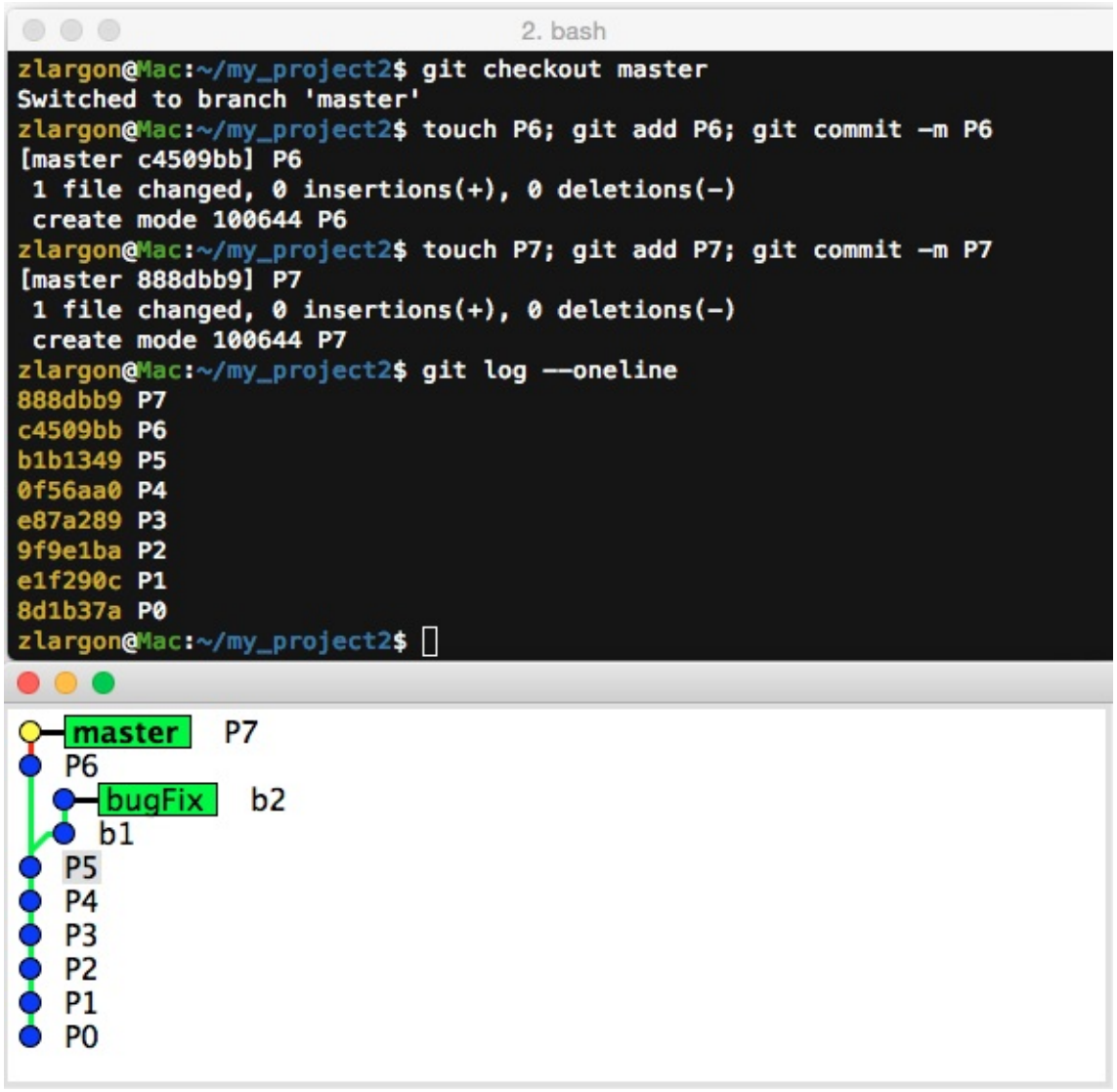
`gitk` 中綠色框框的 `bugFix` 爲粗體，表示我們目前在 `bugFix` 分支上

> `gitk` 這裡要按 shift + F5 來 reload

我們現在在 `bugFix` 分支提交新的 patch b1, b2



我們再切回到 `master` 分支提交 patch P6, P7



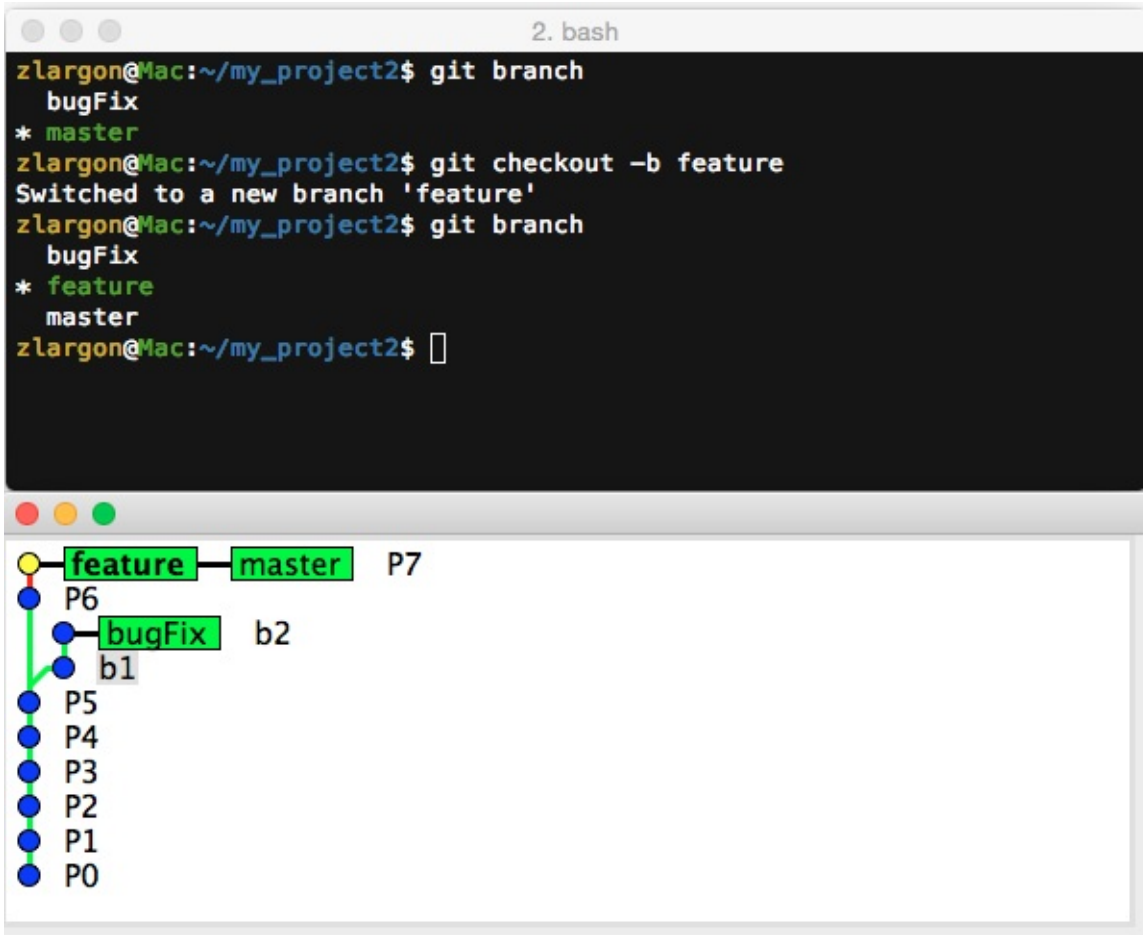
這時候從 `gitk` 上看，分支的感覺就出來了

使用 `git checkout -b <new branch name>` 建立新的 **branch** 並且切換過去

這個指令等同於是以下兩個指令合體

```
$ git branch <new branch name>
$ git checkout <new branch name>
```

`-b` 就是 branch 的意思

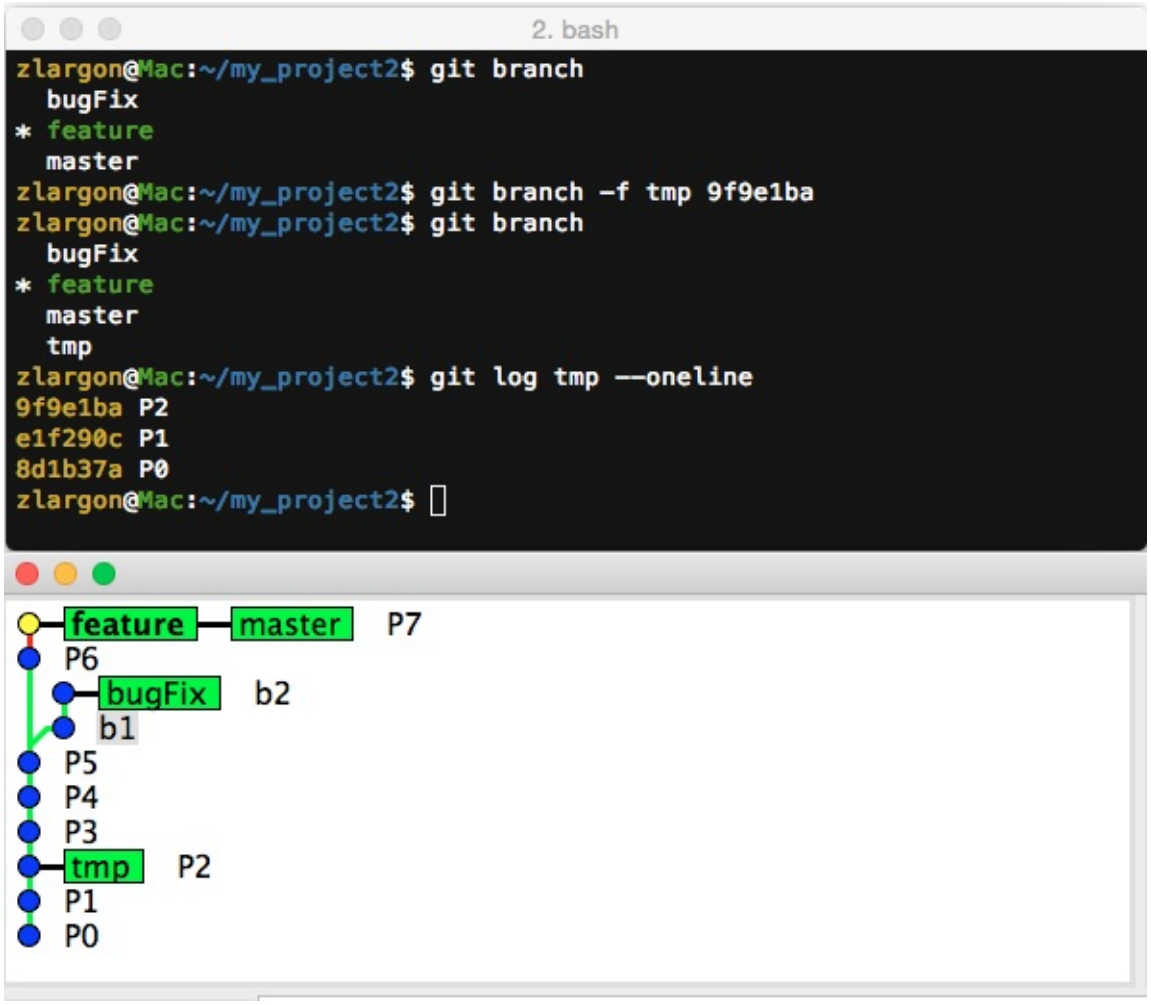


使用 `git branch -f <branch name> <commit id>` 在指定的 patch 上建立 branch（若 branch 已經存在，就切過去）

我們現在希望在 P2 上，建立一個 `tmp branch`

並且用 `git log <branch name>` 來查看此 branch 下有哪些 patch

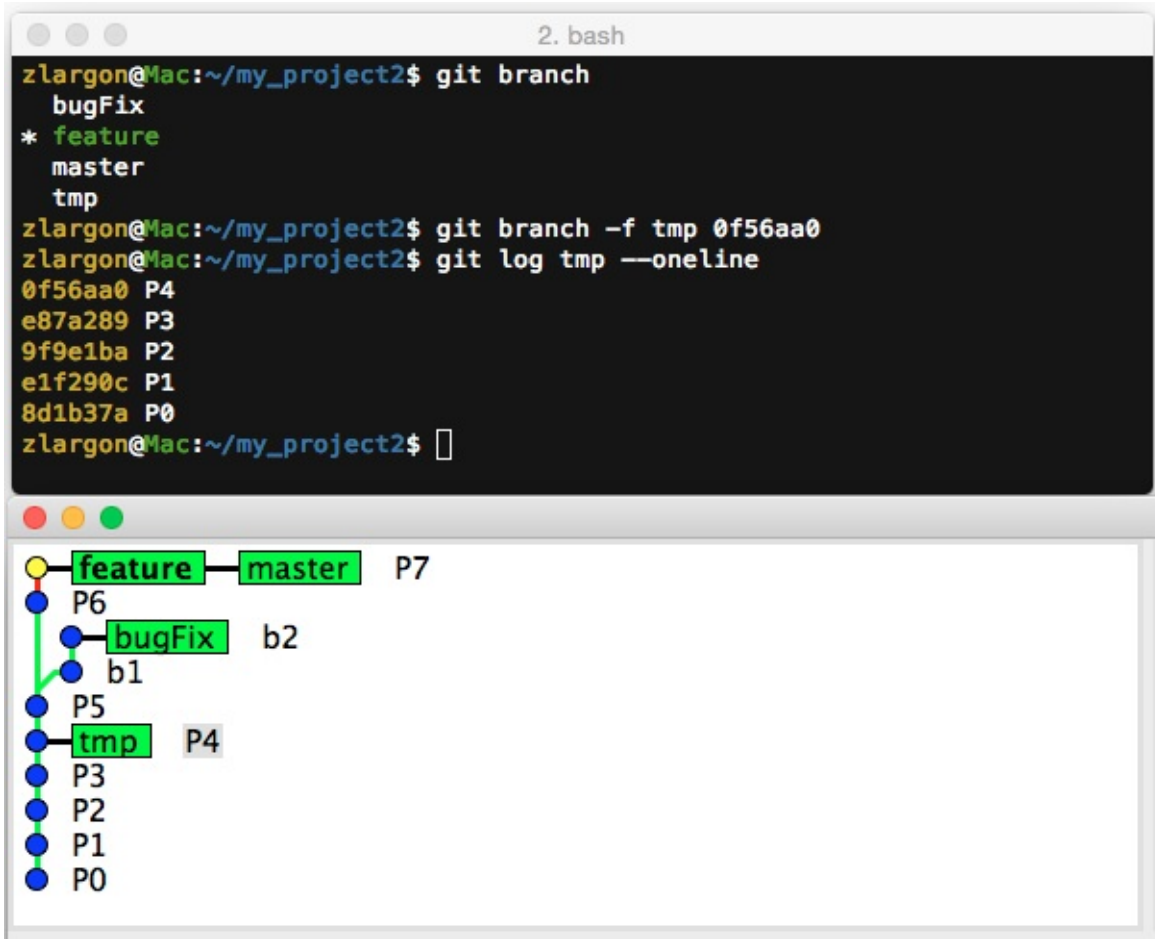
```
$ git branch -f tmp 9f9e1ba     # P2 = 9f9e1ba
$ git log tmp --oneline
```



`-f` 就是 force，有強制的意思

我們也可以透過這個指令，強制把 `tmp branch` 切到 P4

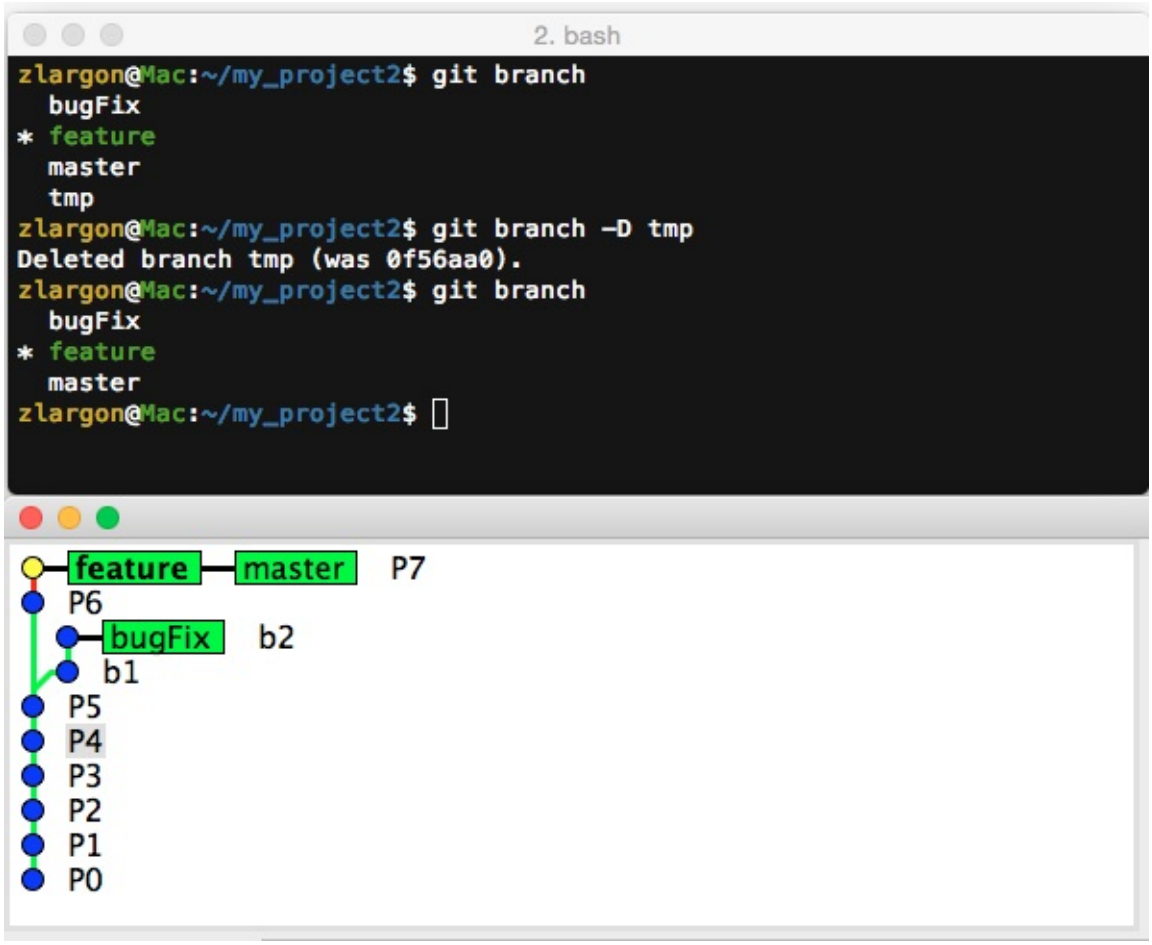
```
$ git branch -f tmp 0f56aa0     # P4 = 0f56aa0
$ git log tmp --oneline
```



使用 `git branch -D <branch name>` 刪除分支

刪除 `tmp` 分支

```
$ git branch -D tmp
```



`-D` 的意思表示 delete 刪除，用大寫 D 的原因是，git 怕我們不小心誤刪分支

> 不過我覺得這樣的設計好像沒什麼太大的意義 XDD
>
> 在某些 git 版本開始，已經可以用小寫 `-d` 直接刪除分支

我們不能刪除目前所在的分支，git 會顯示錯誤的訊息

```
$ git branch -D featrue         # 失敗
```

我們必須切到其它分支上，才能把 `feature` 分支刪除

本章回顧

- 使用 `git branch <new branch name>` 建立新的分支
- 使用 `git checkout <branch name>` 切換分支
- 使用 `git checkout -b <new branch name>` 建立新的 branch 並且切換過去
- 使用 `git branch -f <branch name> <commit id>` 在指定的 patch 上建立 branch（若 branch 已經存在，就切過去）
- 使用 `git branch -D <branch name>` 刪除分支

Commit Tree

在 Git 的世界裡，我們可以把每個 patch 都看成是一個小節點

這些 patch 一個一個的串連起來，最後組成了 ***Commit Tree***

```
                                  o---o---o feature
                                 /
    o---o---o---o---o---o---o---o---o---o---o  master
root       \             /
            o---o---o---o  bugFix
                     \
                      o---o---o  testing
```

不論我們在哪一個節點上面，都可以藉由 `parent` 一路往回走，最終都會回到 `root`

`root` 也就是我們的 first commit

因此每一條從 patch 回到 root 的路徑，其實都是一個的分支

每個 commit id 都是獨一無二的，絕對不會重複，因爲每一條路徑都是唯一的

每個 commit id 都可以視爲是一個版本分支

然而分支其實就只是爲 patch 貼上一個 `branch name` 的標籤而已

這也是爲什麼對 git 來說，建立分支是這麼輕鬆的事情

而當你把某個分支刪除，就像是把標籤撕掉，絲毫不費力

原本分支底下的 patch 不會因而消失，他們都會一直存在於 Git 的 ***Commit Tree***

都可以從 `reflog` 裡把對應的 `commit id` 找回來

Patch 的各種表示法

到目前爲止，我們學過很多方法可以來表示 patch

- ***Commit Id***
- ***HEAD*** 代表目前所在的 ***patch***
- ***Branch Name***

- 在 ***HEAD*** 或 ***Branch Name*** 後面加上 `^` 或 `~n` 來表示相對位置

使用 `git checkout <commit id>` 移動 HEAD 的位置

`HEAD` 代表我們目前所在 patch 的位置

大部份的時間 `HEAD` 都會和某個 branch 重疊在一起（例如 `master` 分支）

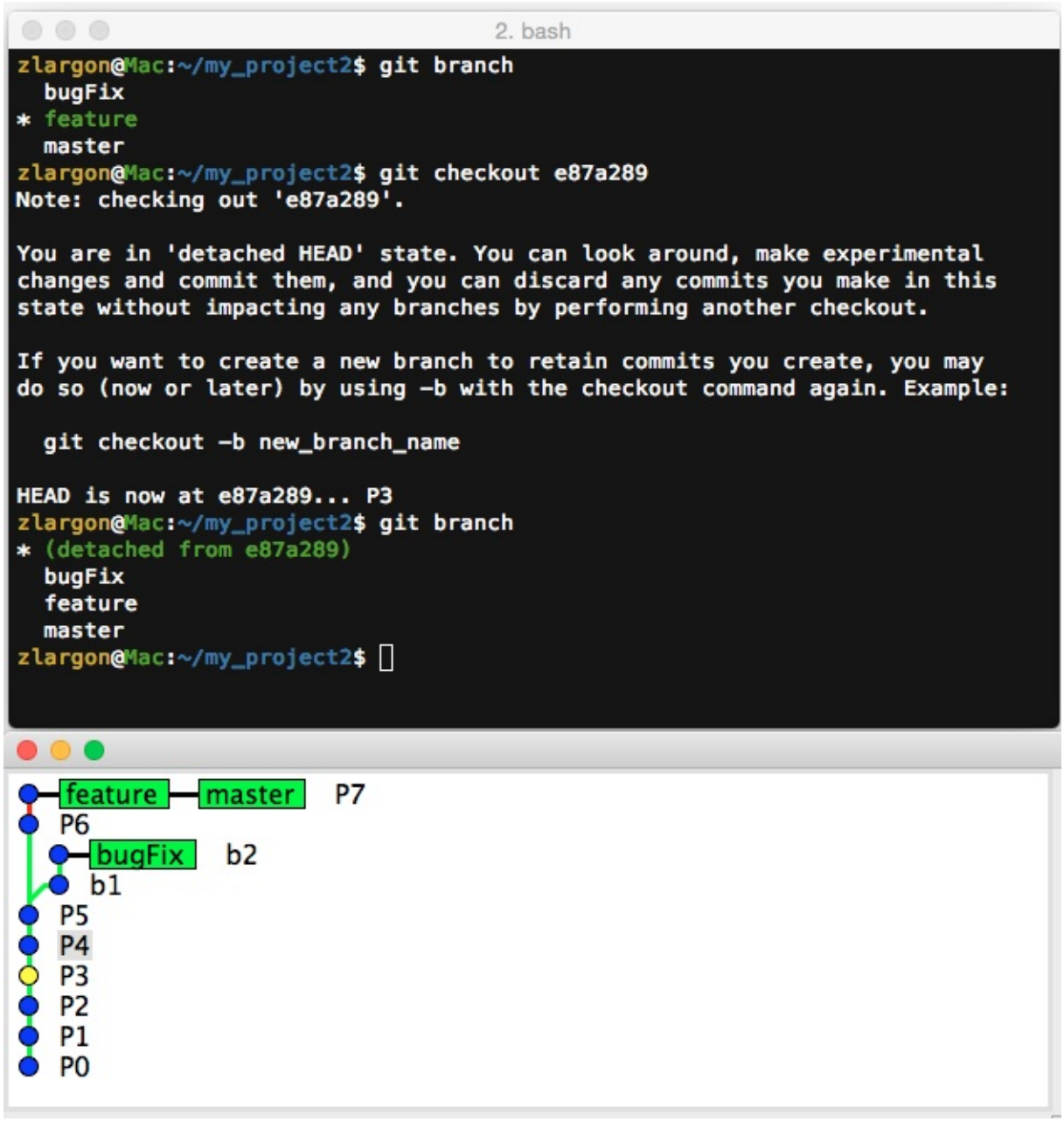
我們在 "Reset Patch" 有學到使用 `git reset --hard <commit id>` 來 reset 到指定的 patch

所以當我們在做 reset 的時候，其實是改變了 `master` 分支的位置

事實上，`HEAD` 可以脫離 `master` 並且遊走 ***Commit Tree*** 中的任何一個 patch，甚至還可以提交 patch

例如說，我們現在想要直接移動到 P3 的位置

```
$ git checkout e87a289      # P3 = e87a289
```



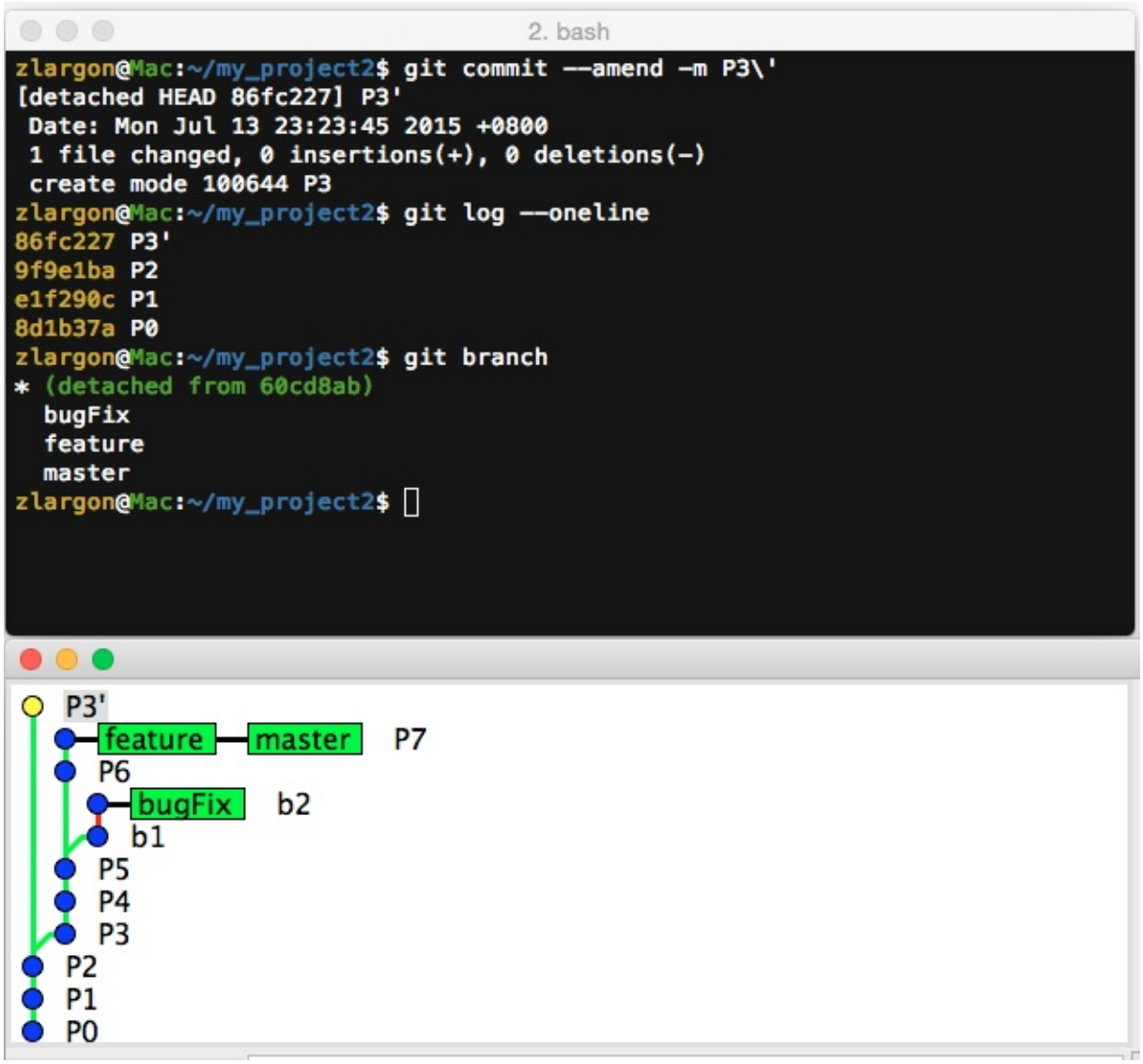
從 `gitk` 可以看到，黃點現在在 P3 的位置

而 git 也會提醒我們，現在的 `HEAD` 是分離狀態，我們可以在這個狀態下繼續提交 patch

可以用指令 `git checkout -b <new branch name>` 把這個 patch 指定爲某個分支

我們現在修改 P3 的 commit message，改成 P3' 看會發生什麼事情

```
git commit --amend -m P3\'
```

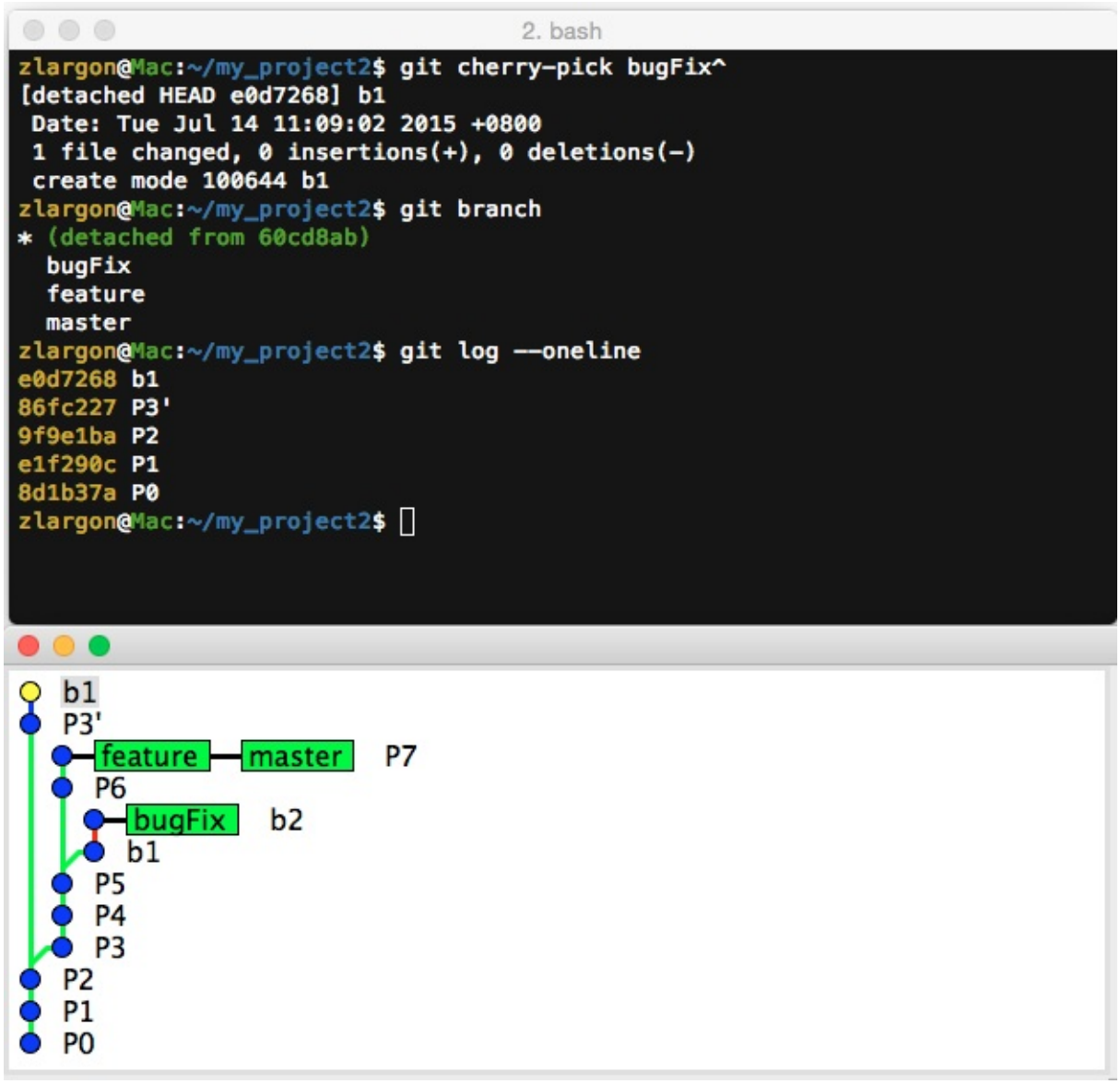


我們修改的 P3 變成了一個新的 patch（commit id 不同）

我們也可以 cherry-pick 某個 patch 進來

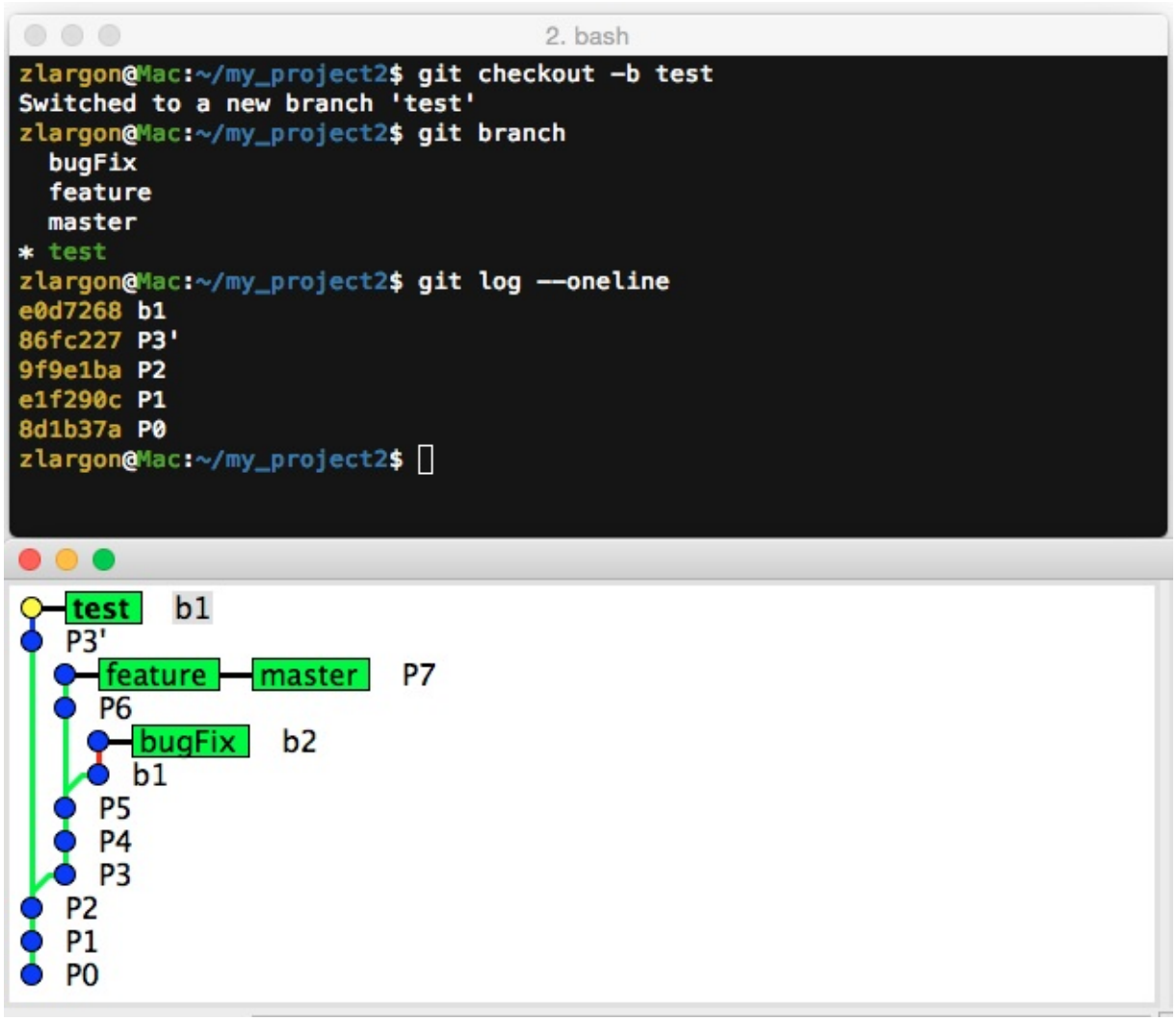
例如我們想要挑 b1 進來

```
$ git cherry-pick 60cd8ab       # b1 = 60cd8ab
$ git cherry-pick bugFix^       # 使用 bugFix 的相對位置來表示
```



最後再用 `git checkout -b <new branch name>` 來產生新的分支

```
$ git checkout -b test_head
```



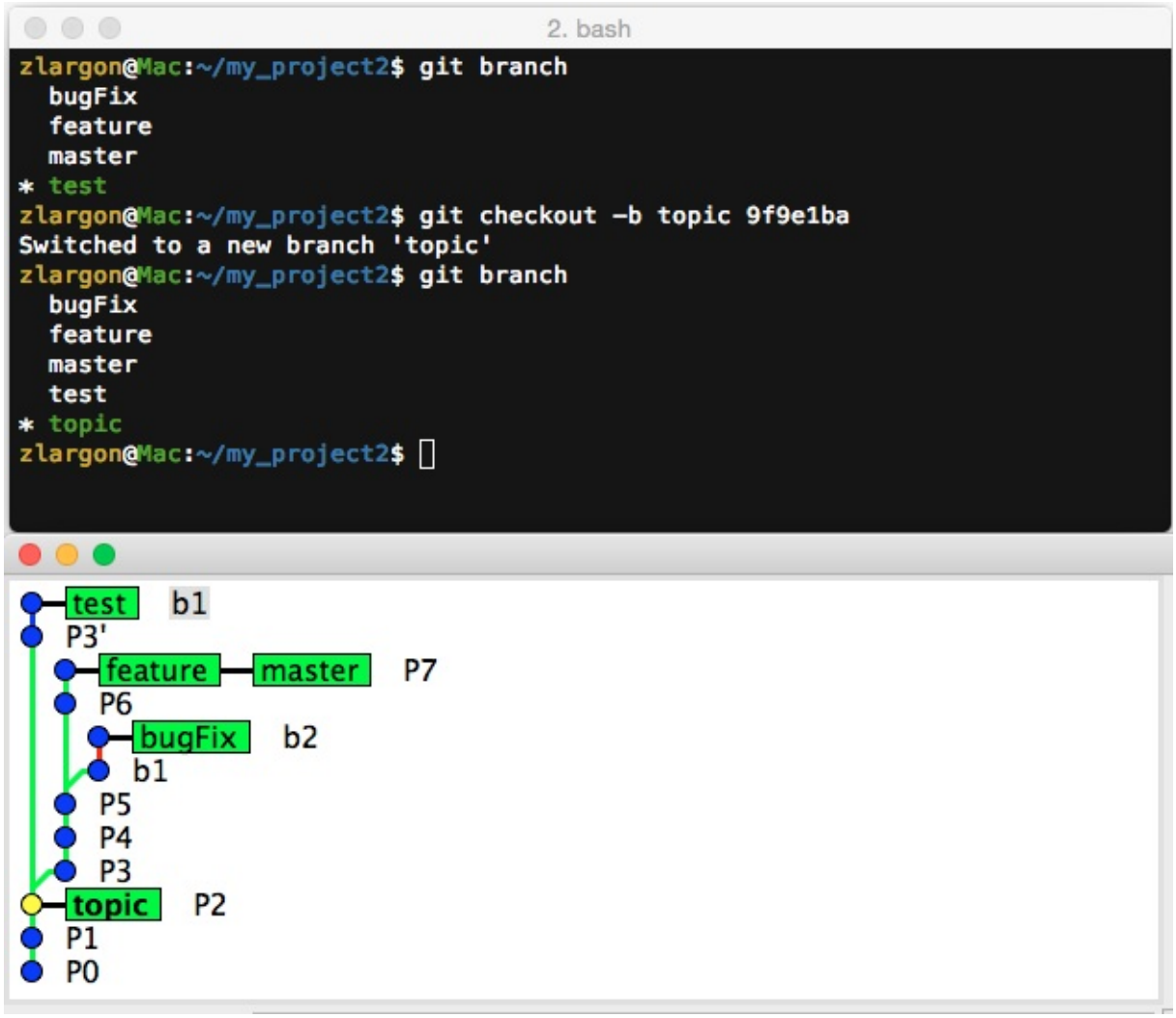
使用 `git checkout -b <new branch name> <commit id>` 在指定的 **patch** 上建立新的 **branch** 並且切換過去

這個指令等同於是以下兩個指令合體

```
$ git checkout <commit id>
$ git checkout -b <new branch name>
```

例如我們現在想要在 P2 建立一個叫做 topic 的分支，並且切換過去

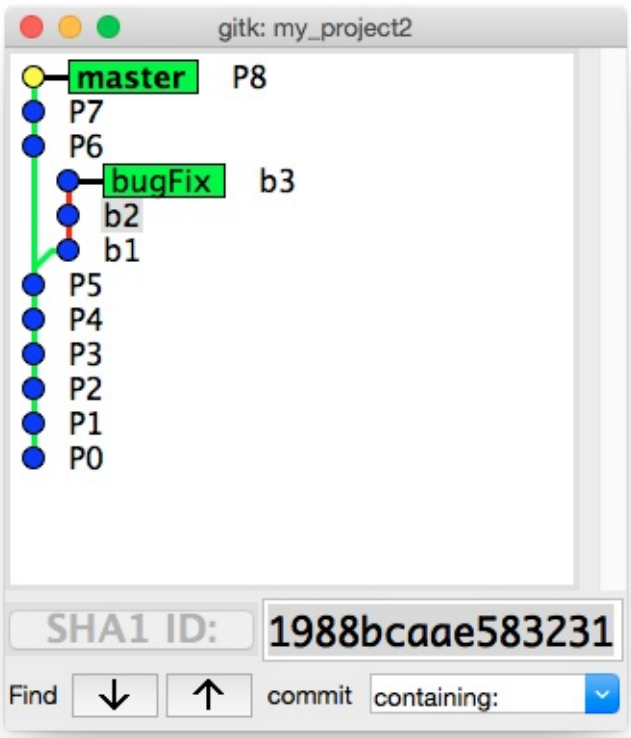
```
$ git checkout -b topic 9f9e1ba         # P2 = 9f9e1ba
```



本章回顧

- ***Commit Tree*** 的概念
- Patch 的各種表示法
- 使用 `git checkout <commit id>` 移動 `HEAD` 的位置
- 使用 `git checkout -b <new branch name> <commit id>` 在指定的 patch 上建立新的 branch 並且切換過去

Rebase 合併分支



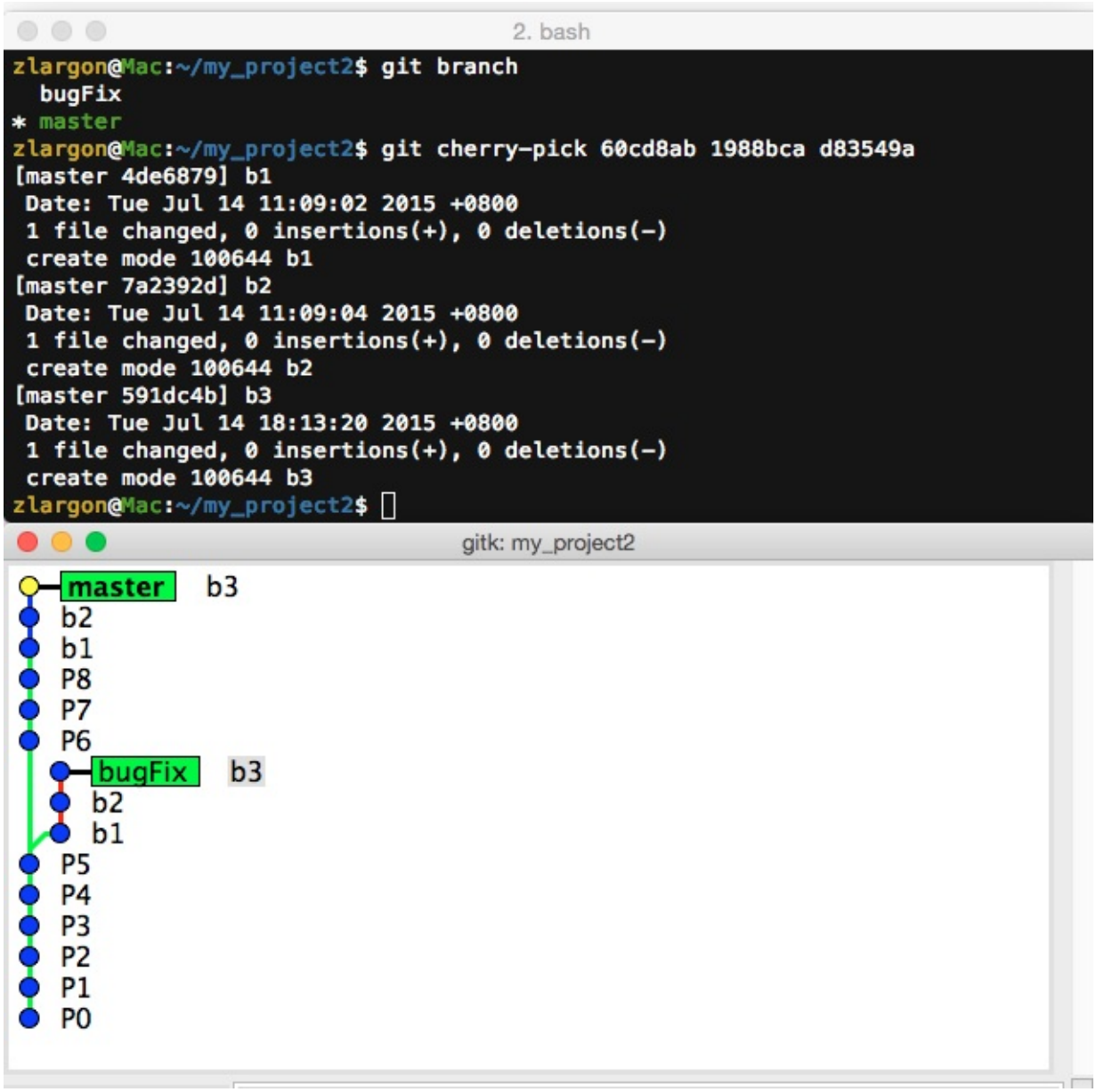
當我們想要將 `bugFix` 分支的 b1, b2, b3 合併到 `master` 分支

最簡單的做法，就是用 `cherry-pick` 一個一個挑

使用 `git cherry-pick <commit 1> <commit 2> ...` 一次接多個 patch

`git cherry-pick` 後面可以一次接多個 commit id

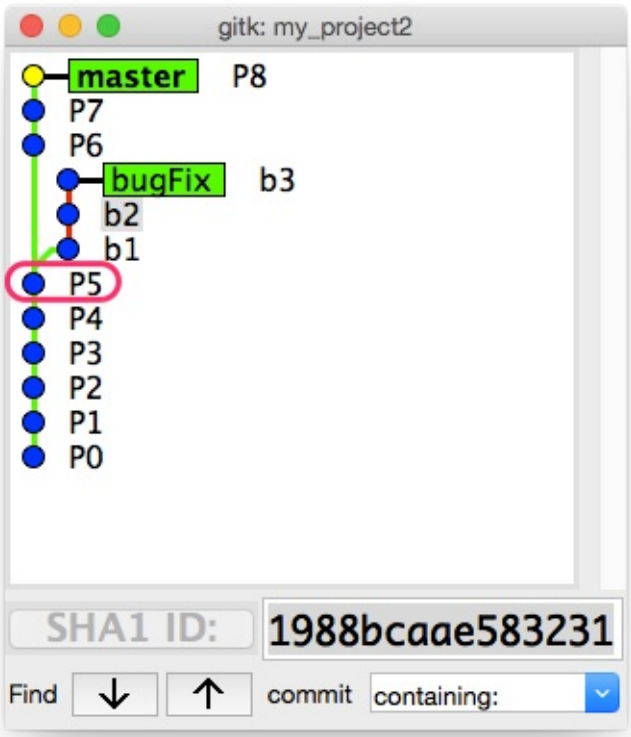
```
$ git checkout master
$ git cherry-pick 60cd8ab 1988bca d83549a   # b1 = 60cd8ab
                                            # b2 = 1988bca
                                            # b3 = d83549a
```



如果分支很長的時候，用 `cherry-pick` 一個一個挑感覺還是滿累的

那有沒有其他做法呢？

使用 `git rebase <new base>` 重新定義分支的基準點



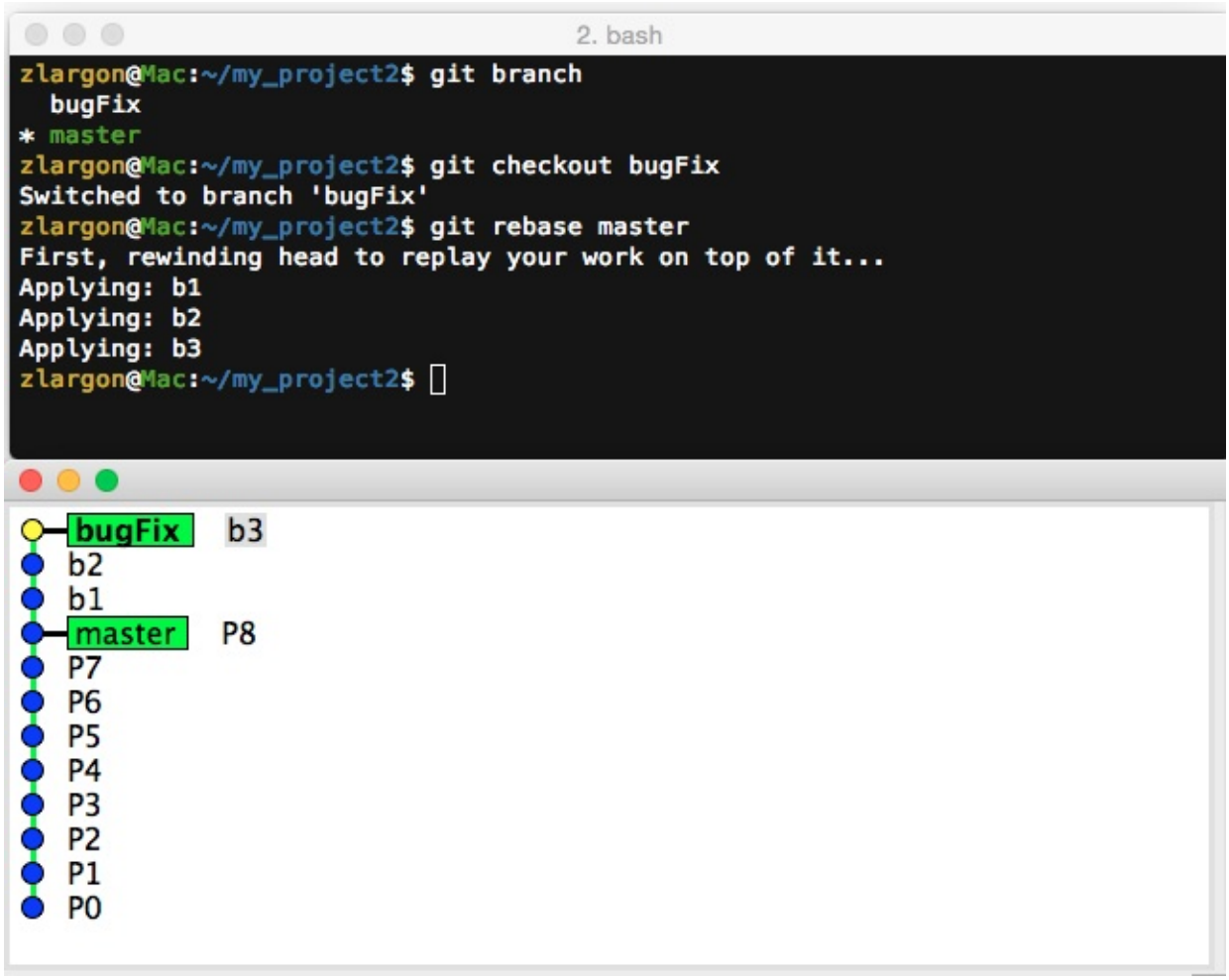
如果我們今天想要把 `bugFix` 接到 `master` 的後面

那麼就要先切到 `bugFix` 分支，然後以 `master` 作爲爲新的基準點進行 rebase

```
$ git checkout bugFix
$ git rebase master
```

因此 git 會根據 `bugFix` 一路往上去找與 `master` 的交會點，於是找到了 P5

接著就將 `bugFix` 之後的 patch 一個一個接到 `master` 的後面



這裡要注意是誰 ***rebase*** 誰！

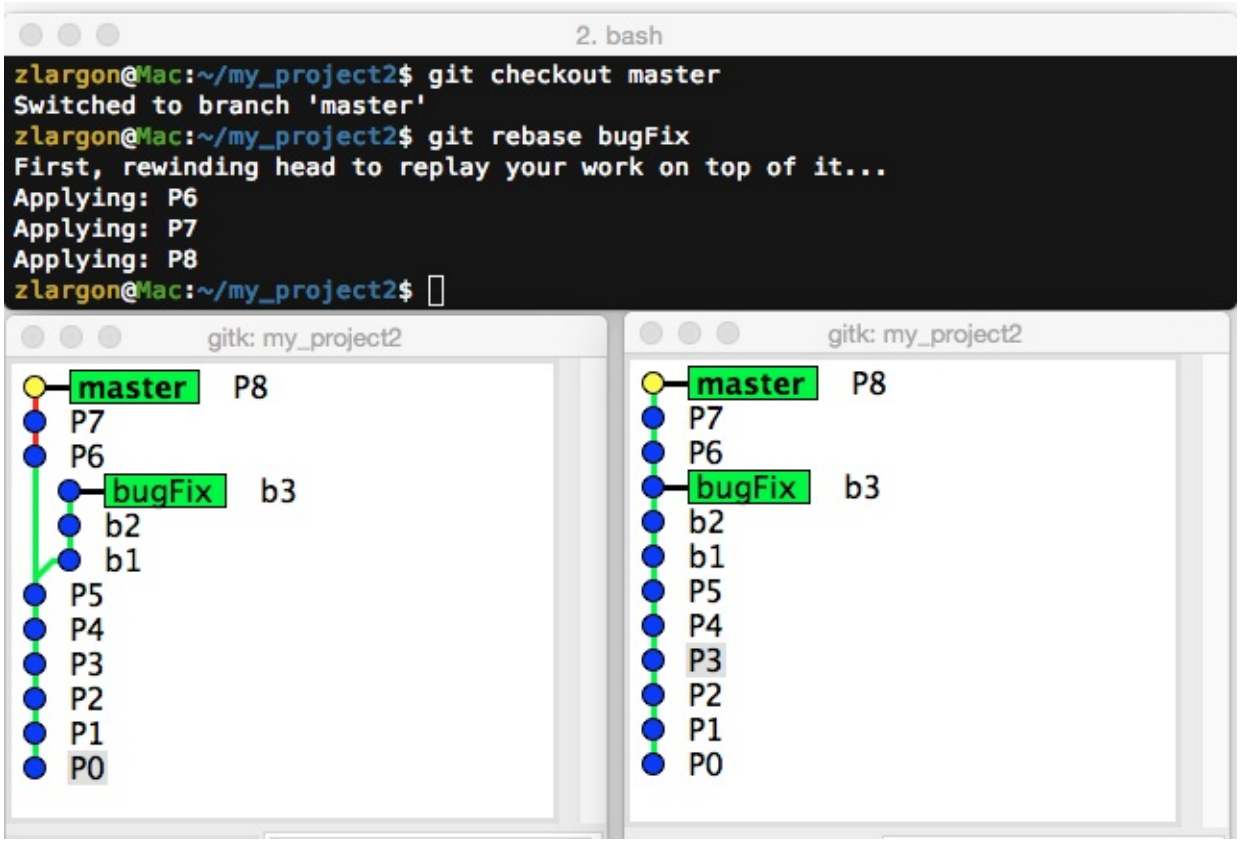
`bugFix` 分支接到 `master` 的後面

```
$ git checkout bugFix
$ git rebase master
```

另外，以上兩行可以寫成一行 `git rebase <new base> <branch name>`

```
$ git rebase master bugFix
```

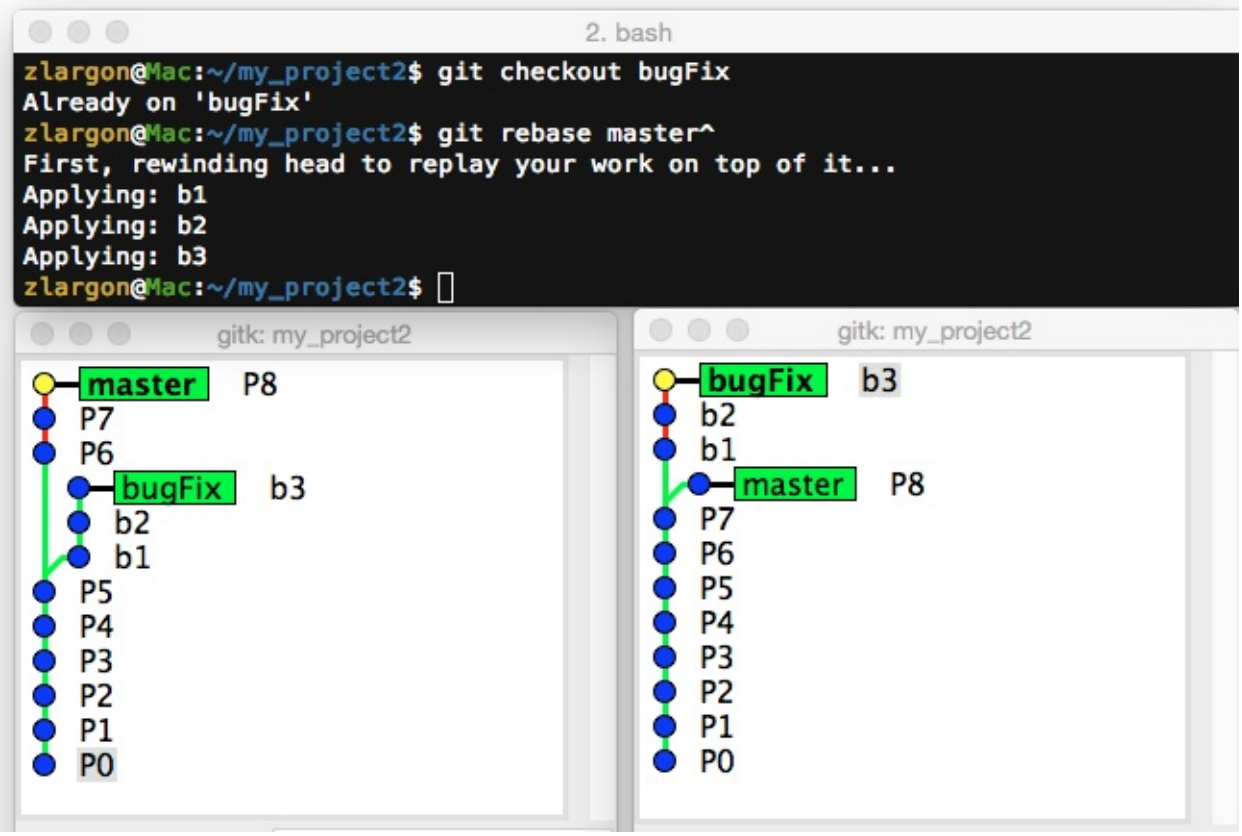
如果兩個分支弄相反的話，就會變成這樣把 `master` 的 P6, P7, P8 接到 `bugFix` 的後面



在我們有 ***Commit Tree*** 的概念之後，其實 `git rebase master` 跟 `git rebase P8` 兩個意思是一樣的

所以，假如我們希望把 `bugFix` 接到 P7 後面的話，就可以這樣做

```
$ git checkout bugFix
$ git rebase master^        # P7 = master^
```



rebase 的注意事項

1. ***rebase*** 指令不直覺，容易造成誤用（誰 ***rebase*** 誰？）

   rebase 還有另外一個常用的指令 `git rebase --onto <new base> <after this commit> <to this commit>`

   我覺得這個指令不好理解，而且實用性低，所以我們這裡不打算教這個指令

2. ***rebase*** 其實就是 ***cherry-pick*** 的高階指令

   當一次 rebase 多個 patch 的時候，其實就是連續做 cherry-pick 的動作

   所以有可能會連續發生 ***conflict***

   當這種情況發生的時候，我寧可手動 cherry-pick

   每挑一個 patch 就跑一次測試程式，確保程式可以正常運作再挑下一個 patch

3. ***rebase*** 會去修改 **"被 *rebase* 的分支"** 的內容

   當下完 rebase 之後，才發現自己接錯了！

這時候就必須去 reflog 去找 rebase 前的 commit id

或者可以用 `git reset --hard ORIG_HEAD` 回到 rebase 前的 `HEAD`

雖然都有補救方案，不過還是建議使用 rebase 前先開一個 `backup` 的分支以免發生意外

本章回顧

- 使用 `git cherry-pick <commit 1> <commit 2> ...` 一次接多個 patch
- 使用 `git rebase <new base>` 重新定義分支的基準點
- 使用 `git reset --hard ORIG_HEAD` 回到 rebase 前的 `HEAD`

Merge 合併分支

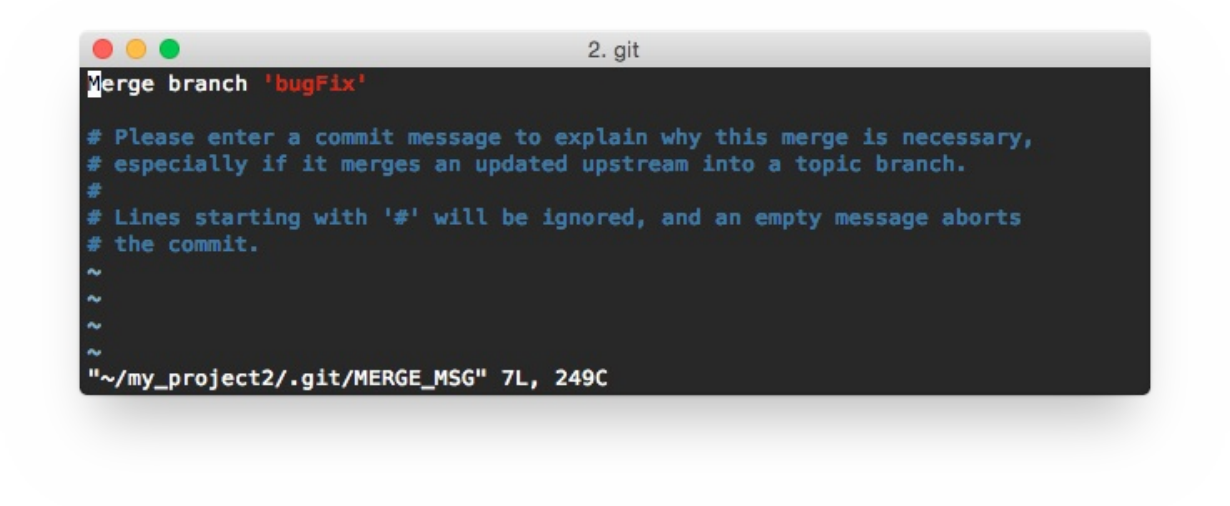
Git 除了可以用 `cherry-pick` 和 `rebase` 的方式來合併分支之外

還可以用 `merge` 指令來合併分支

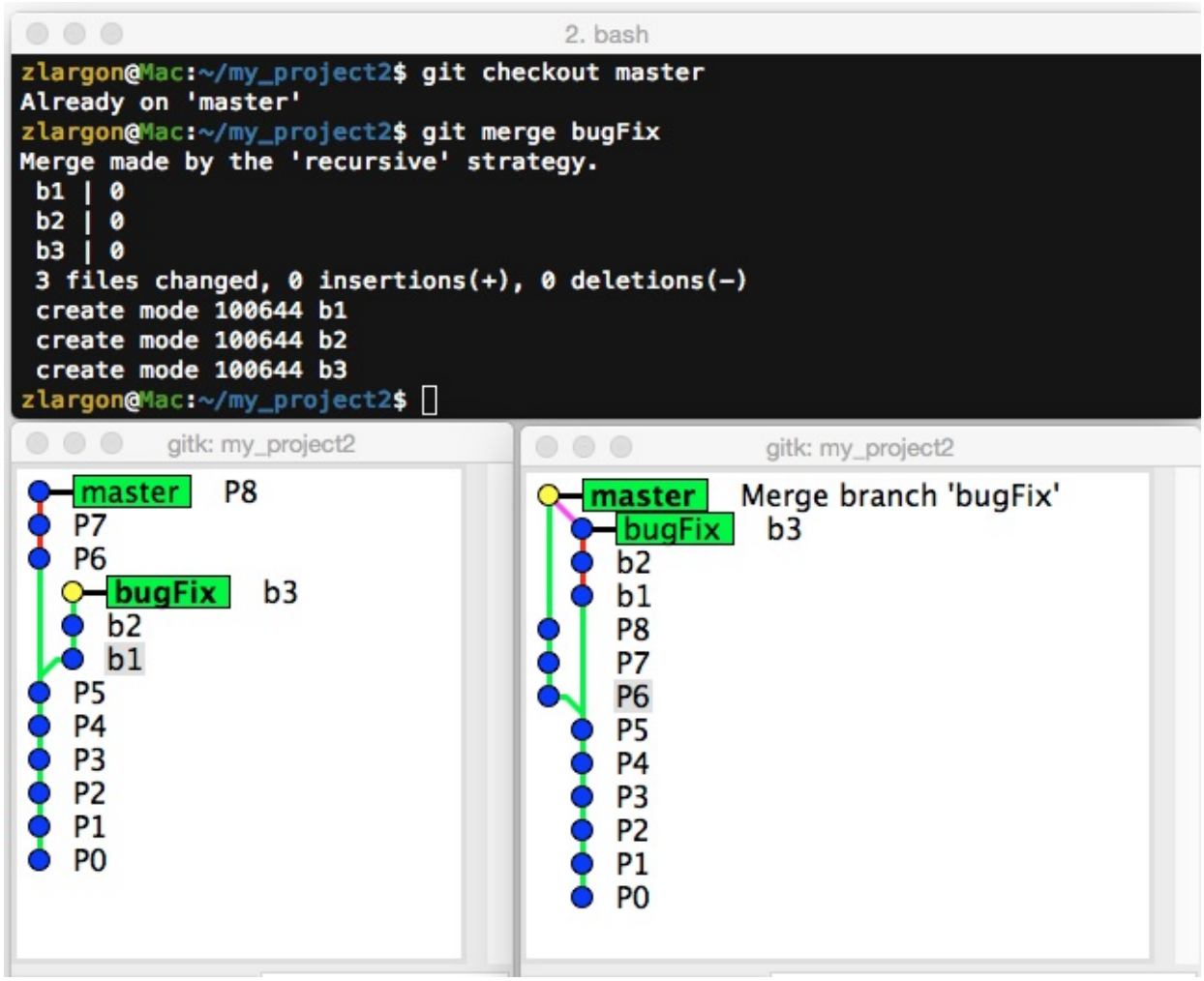
使用 `git merge <branch name>` 來合併分支

例如我們的 `master` 要去合併 `bugFix` 分支

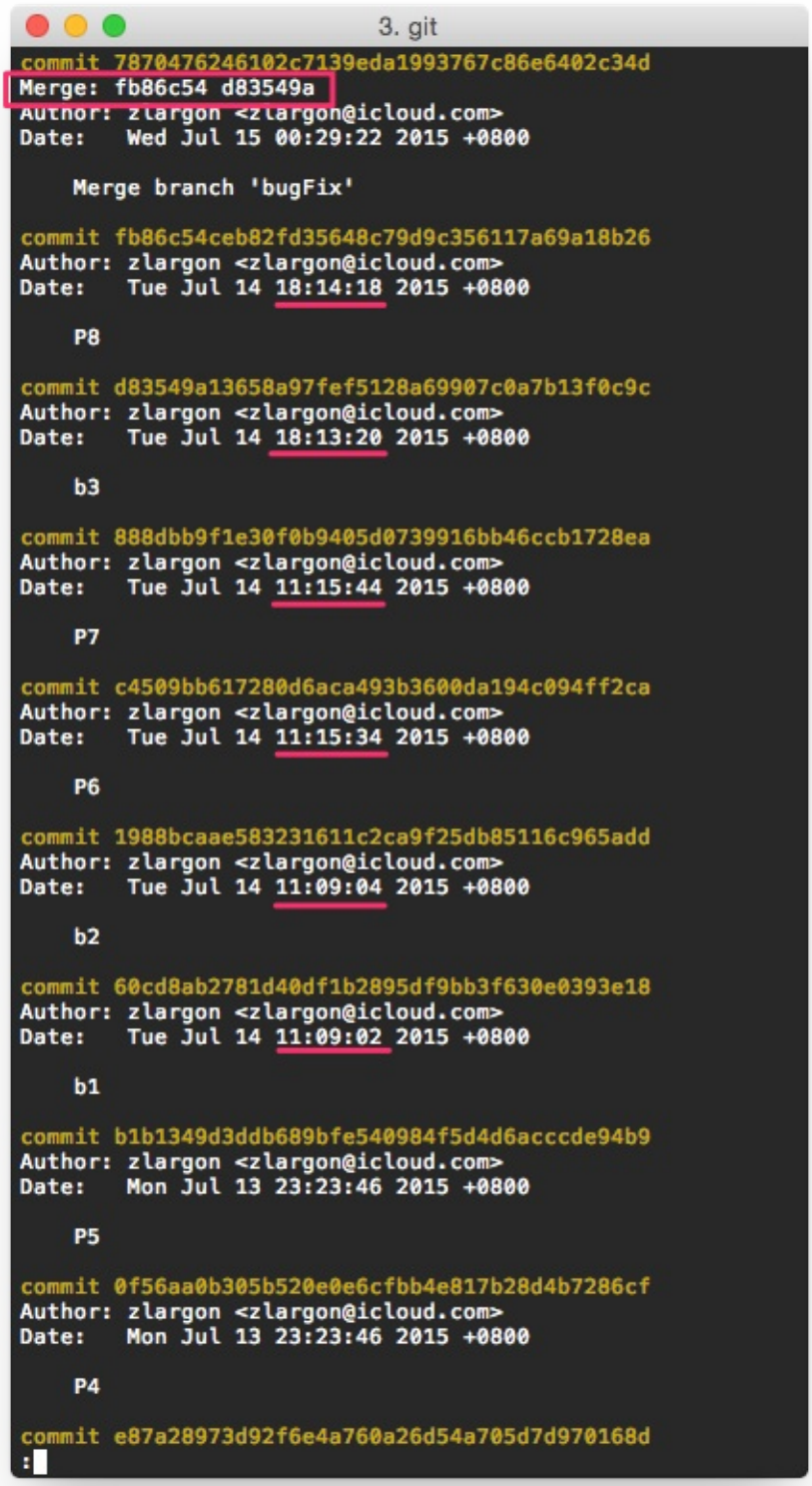
```
$ git checkout master
$ git merge bugFix      # 按下 enter 之後，會進入 vim 文字編輯模式，要求你提交 merge patch
```



合併完成後，他會多出一個 merge patch



接著我們用 `git log` 來查看



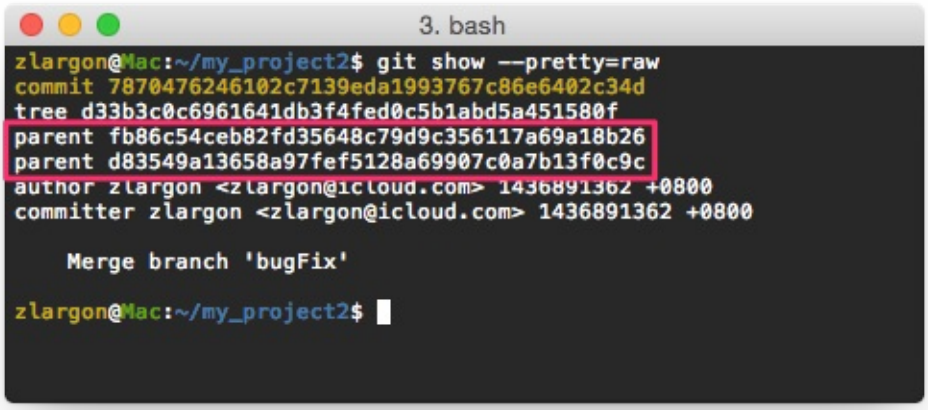
我們可以看到 merge patch 有多一個 Merge 的欄位，他紀錄的是 P8 跟 b3 的 commit id

也就是原始兩個 branch 的頭

另外我們可以看到 `git log` 把 `master` 的 P6, P7, P8 跟 `bugFix` 的 b1, b2, b3 順序打散了

並且會被重新按照 提交時間 來排序，每個 patch 都會保持原本的 commit id

我們用 `git show --pretty=raw` 來查看這個 merge patch，可以看到他有兩個 parents，也就是 P8 跟 b3



Git Merge 的特性

- 合併完成後會有 Merge Patch，用來記錄原本兩個分支的 commit id
- 從 git log 看不出原本兩個分支的順序，會被混在一起並用提交時間來排序
- 不會修改原始的 commit id

Fast-Forward

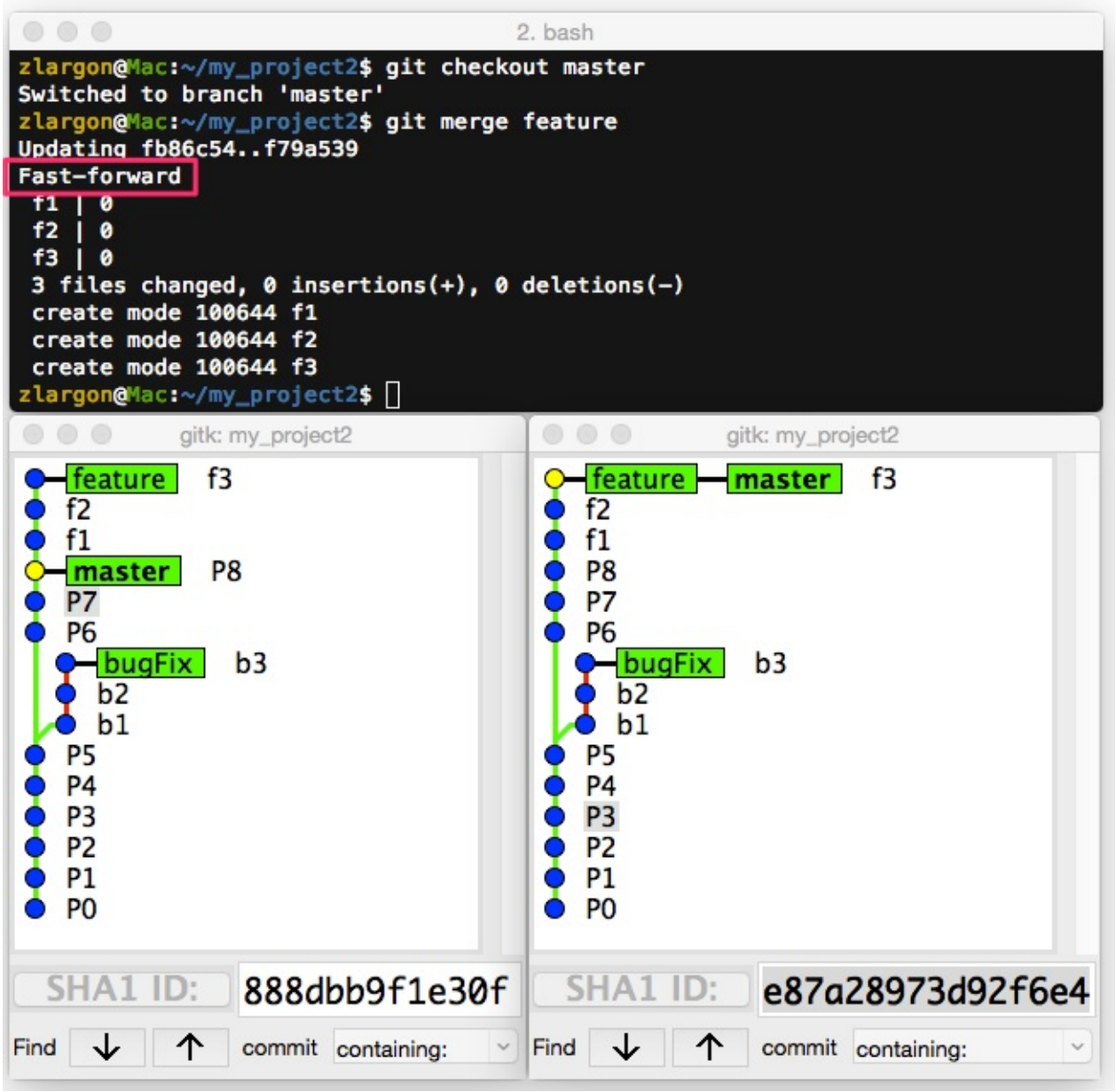
有一種情況下，不會出現 Merge Patch

那就是其中一個分支爲另一個分支的子集

以這張圖來說，`master` 完全被包含於 `feature` 分支

所以 `master` 是 `feature` 的子集

```
$ git checkout master
$ git merge feature
```



因爲 `feature` 跟 `master` 根本就在同一條分支上

所以也沒有並要用 Merge Patch 來記錄兩個分支的 commit id

因此 git 就會直接把 `master` 移到 `feature` 的位置

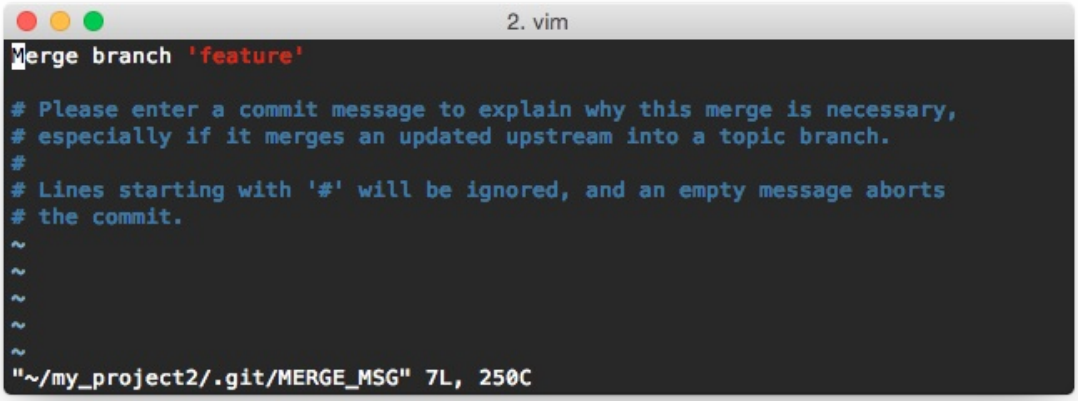
使用 `git merge --no-ff <branch name>` 強制產生 Merge Patch

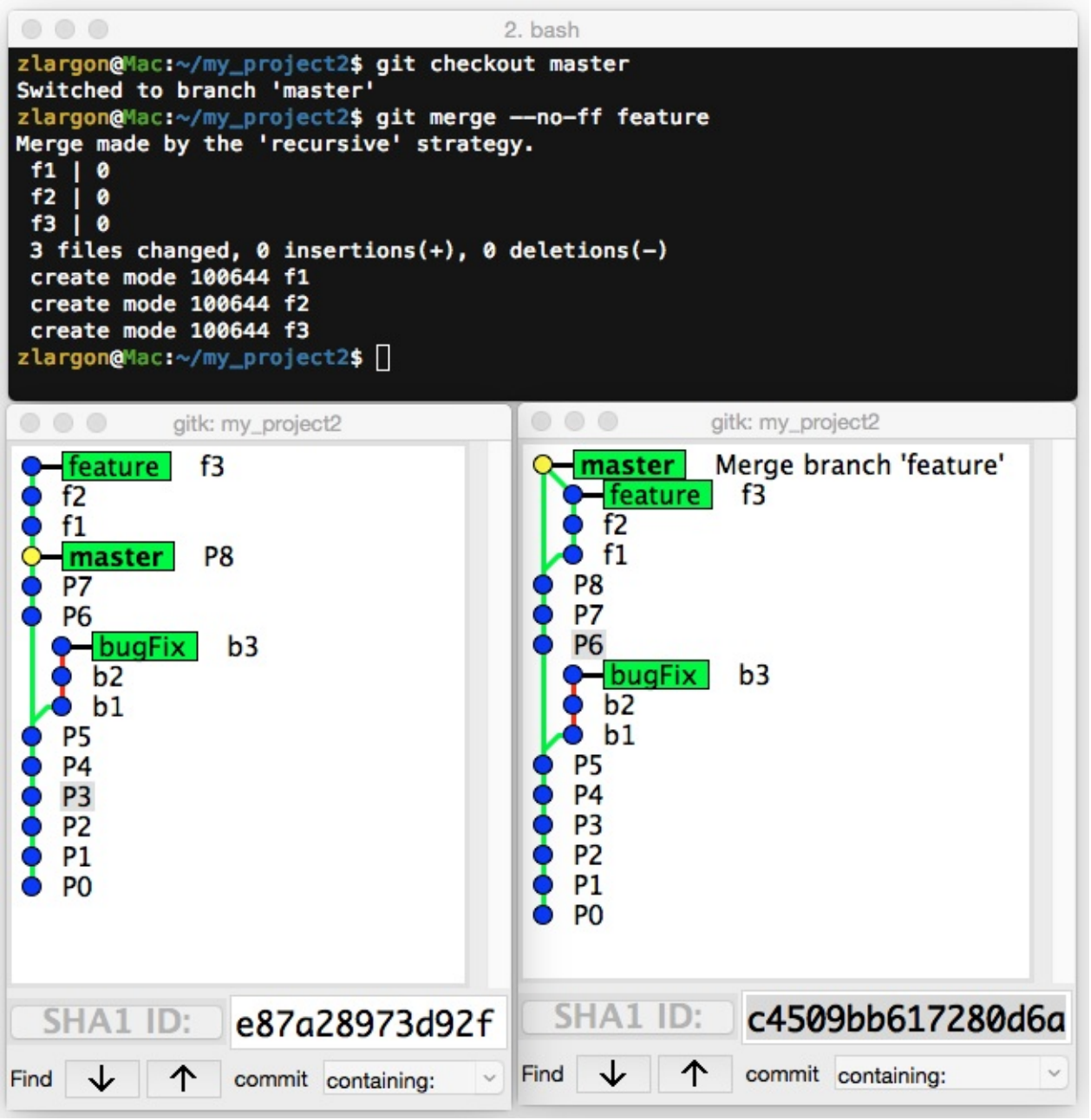
Fast-Forward 是 `git merge` 預設的行爲，如果不想要 Fast-Forward 就要加上參數 `--no-ff`

意思就是 No Fast-Forward

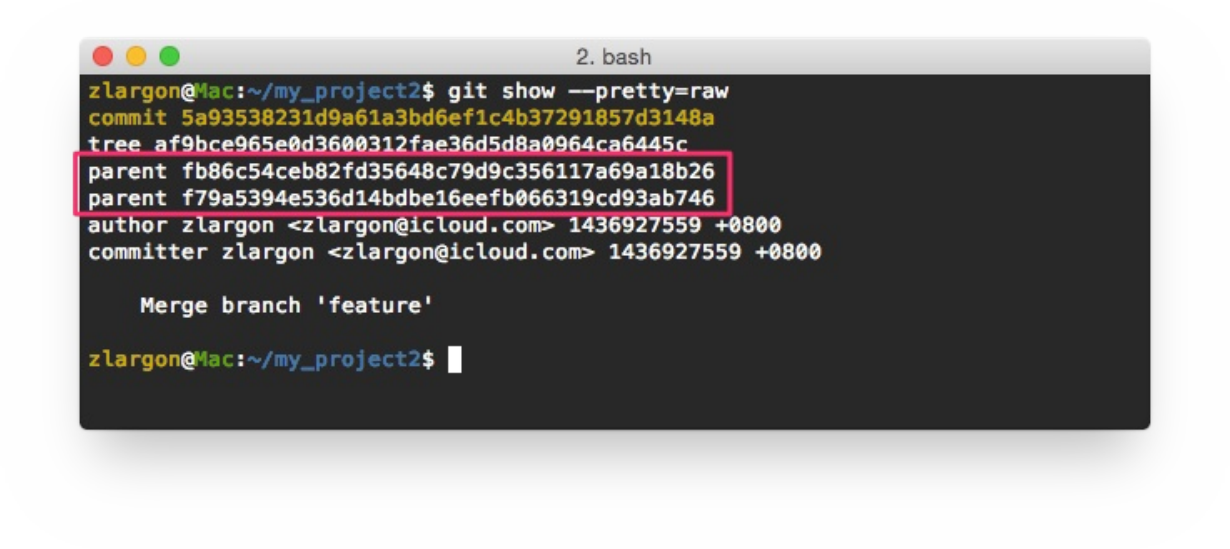
以剛才合併 `featrue` 的例子來說，我們加上參數 `-no-ff` 就會強制產生 Merge Patch

```
$ git checkout master
$ git merge --no-ff feature     # 按下 enter 之後，會進入 vim 文字編輯模式，要求你提交 merge patch
```





使用 `git show --pretty=raw` 可以看出，Merge Patch 有兩個 parents，分別是 P8 跟 f3



這樣的好處是，我們可以清楚的從 `gitk` 看出 f1, f2, f3 原本是隸屬於 `feature`，後來才被 merge 到 `master`

Merge 與 Rebase 比較

Merge	Rebase
容易理解	不容易理解
會產生 Merge Patch，保存原始的 commit id	重新 cherry-pick，重新產生 commit id
當發生 ***conflict*** 的時候，全部在 Merge Patch 一次解完	Rebase 的過程中，每次 cherry-pick 都可能會發生 ***conflict***
commit tree 會有兩個 parents，不容易 trace code	commit 路徑單純，容易 trace code，或回到指定版本
git log 無法呈現合併前分支順序性	git log 可以呈現正確的提交順序
過多的 Merge Patch 會看起來很亂，沒有意義	分支乾淨一致

- merge 或 rebase 皆可用 `git reset --hard ORIG_HEAD` 回到分支合併前的情況

我個人建議是盡量避免使用 `merge`，除非兩隻 branch 的差異過大，逼不得已的時候才會用 `merge`

應盡量確保分支的簡單性，讓每個 patch 容易拆解或重組

所以結論是... `cherry-pick` 才是王道啊 XDDD

本章回顧

- 使用 `git merge <branch name>` 來合併分支
- 使用 `git merge --no-ff <branch name>` 強制產生 Merge Patch

遠端篇

新增專案

```
ssh-keygen -t rsa
ssh -T git@github.com
```

設定 Repo URL

```
git remote -v
git remote add <short name> <repo url>
git remote rm <short name>
git remote rename <short name> <new name>
```

上傳分支

```
git branch -a
git push <remote name> <branch name>
```

設定 Upstream

```
git push -u <remote name> <branch name>
git branch -u <remote>/<remote branch>
git branch --unset-upstream
```

複製 / 下載專案

```
git clone <repo URL>
git clone <repo URL> -b <branch name>
git clone <repo URL> <folder name/path>
git clone <local project>
```

同步遠端分支

```
git fetch <remote name>
git fetch --all
git remote update
git pull
git pull --rebase
```

強制更新遠端分支

```
git push -f
```

刪除遠端分支

```
git push <remote name> :<branch name>
git remote show <remote name>
git remote prune <remote name>
git remote update -p
git fetch --all -p
git fetch -p
```

新增專案

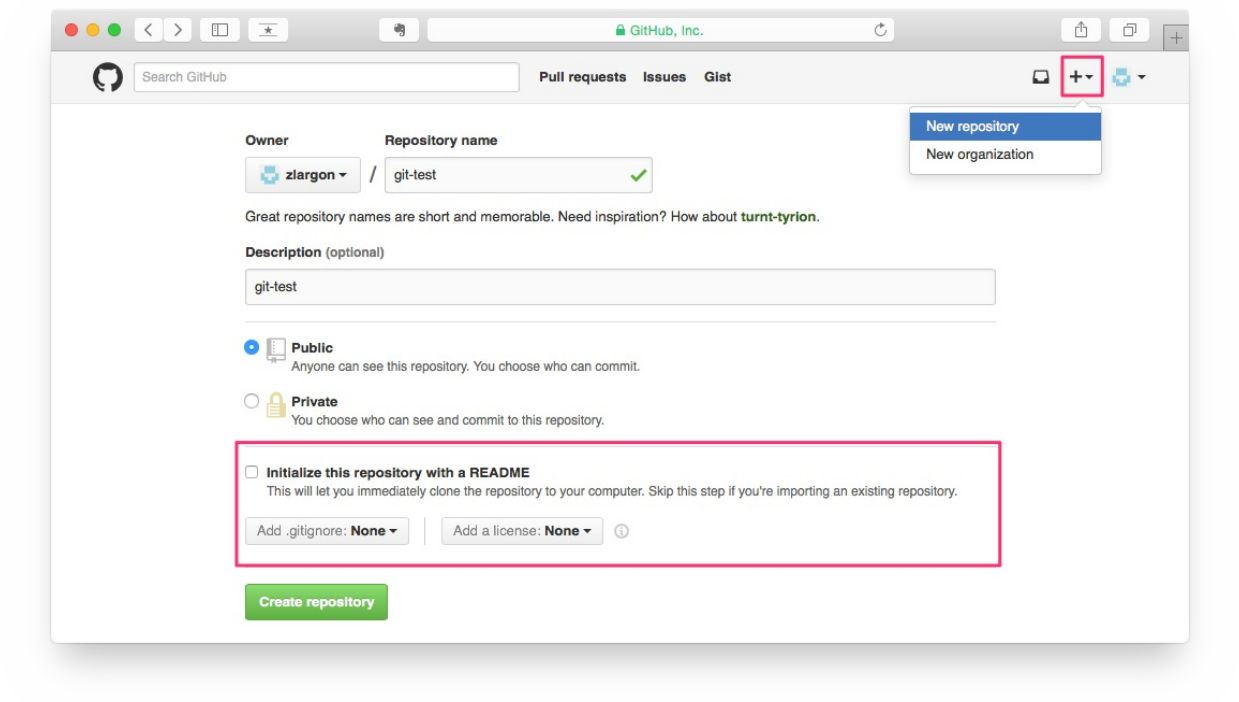
我們這裡將以 Github 作爲示範，大部份的平台做法都大同小異

這裡必須要先申請好 Github 的帳號

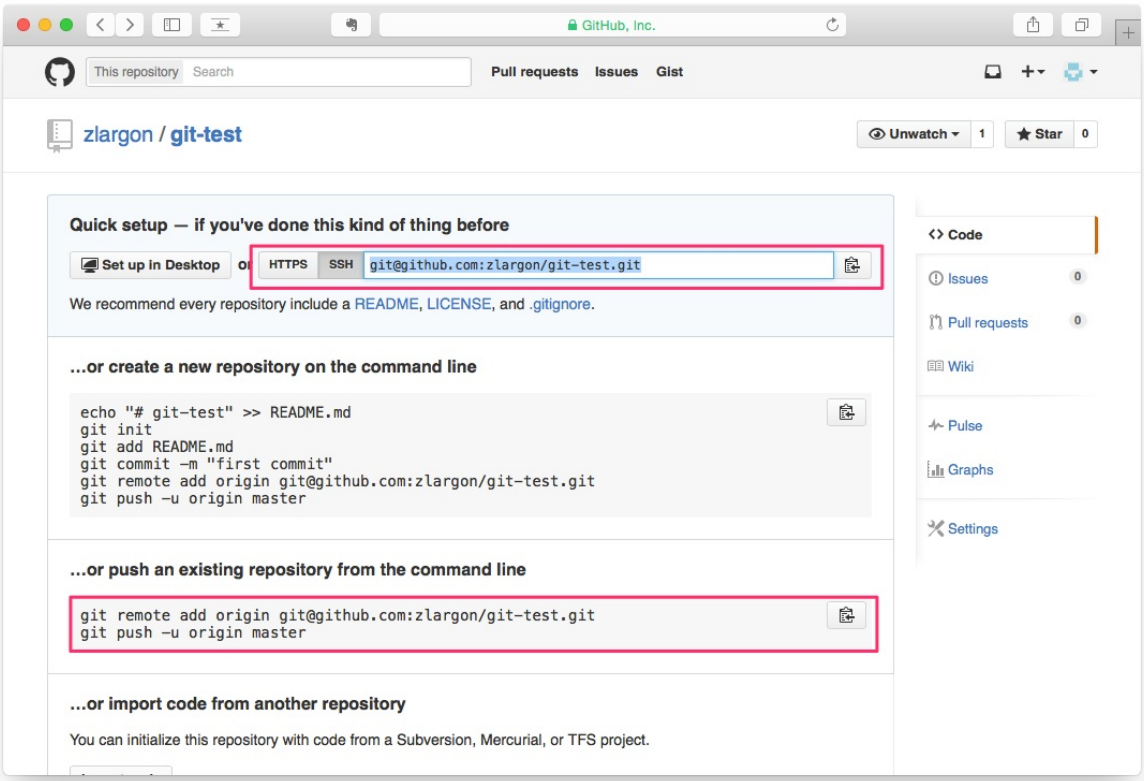
https://github.com/

在 Github 新增專案

1. 按右上角的 `+`，選擇 `New repository`
2. 輸入 Repository name
3. 這裡先不要勾選 **"Initialize this repository with a README"**
4. 按 `Create repository`



新增完成後，Github 會提供你 Repository 的 URL



Repository 的 URL，分成 ***HTTPS*** 和 ***SSH*** 兩種

HTTPS：`https://github.com/zlargon/git-test.git`
SSH：`git@github.com:zlargon/git-test.git`

這兩者的差別在於，若是使用 ***HTTPS*** 的話，每次上傳 code 到 Github 的時候，都要輸入一次 username、password

而使用 ***SSH*** 的話，只要設定好一次之後，就不用再輸入帳號/密碼了

設定 *SSH*

當我們透過 ***SSH*** 跟 Github 連線的時候，會使用 **RSA** 加密演算法

RSA 金鑰是一對的，一把私鑰 + 一把公鑰

我們要先在自己的電腦產生 ***RSA*** 的金鑰，並且保存在電腦裡

然後把 **"公鑰"** 提供給 Github Server

之後使用 ***SSH*** 與 Github Server 連線的時候，他會用 **"公鑰"** 跟我們的電腦的 **"私鑰"** 進行加密以及身份認證

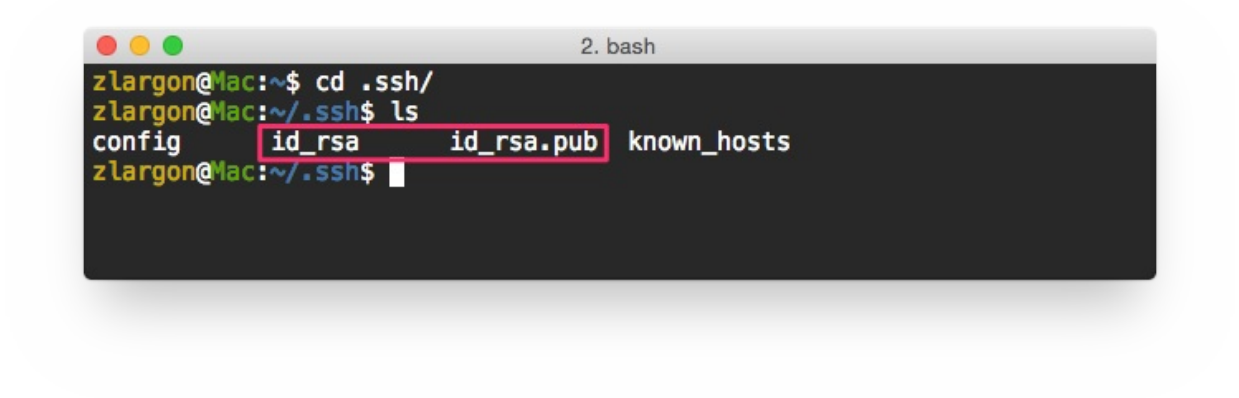
因此設定了 ***SSH*** 之後，我們連線時就不要再輸入 username 跟 password

1. 使用 `ssh-keygen` 產生 ***RSA*** 金鑰

首先我們要先檢查我們的電腦有沒有產生過 ***RSA*** 金鑰

通常產生過的金鑰會被放在 `~/.ssh` 資料夾底下

- ***id_rsa***：私鑰
- ***id_rsa.pub***：公鑰（pub = public）

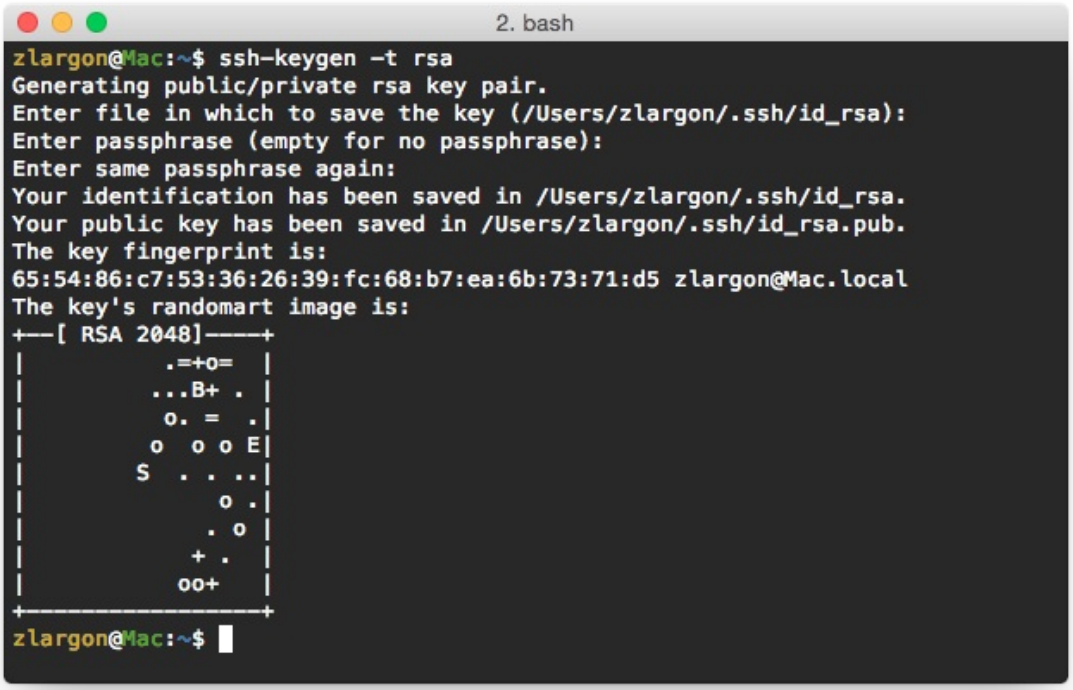


如果已經有了，就不需要再重新產生了

如果沒有的話，就要使用 `ssh-keygen` 來產生

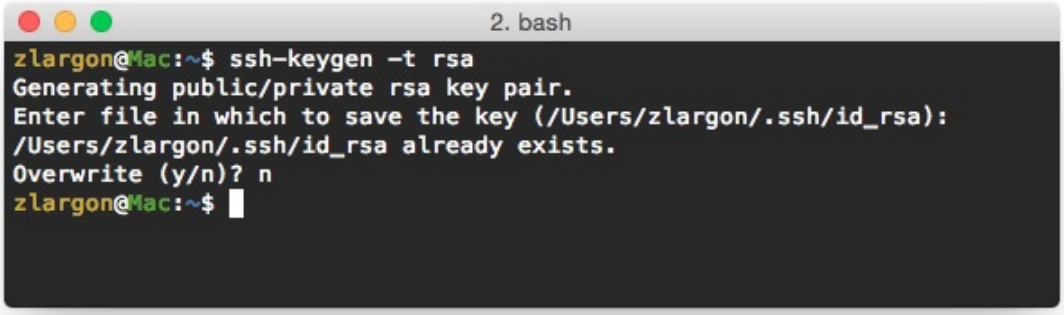
```
$ ssh-keygen -t rsa
```

這裡只要一直按 enter 就行了



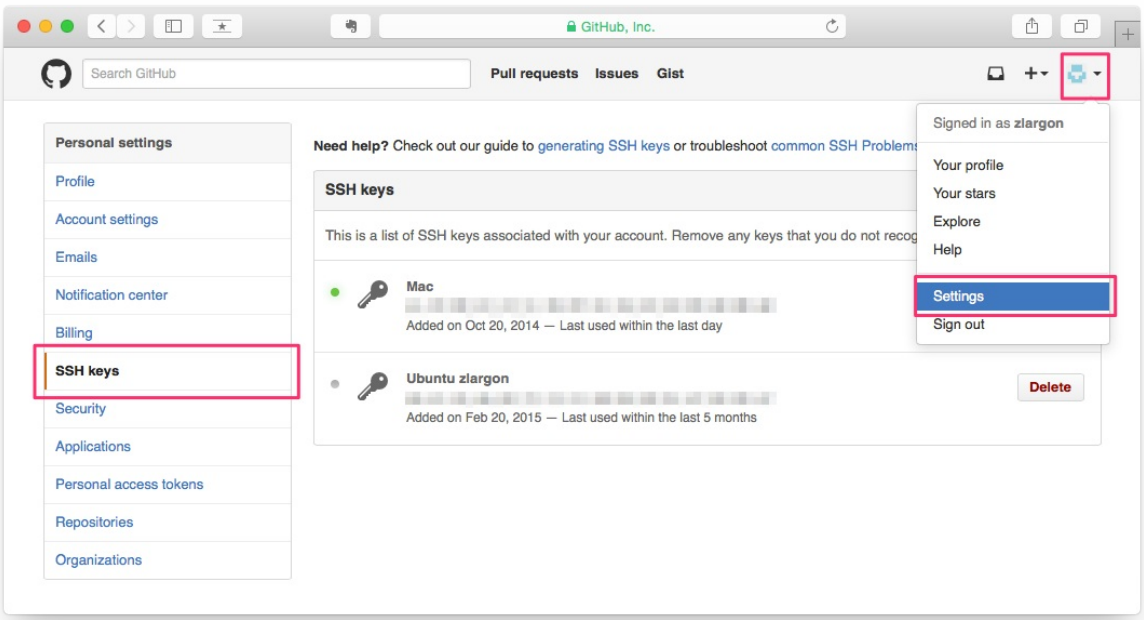
如果出現 ***id_rsa already exists*** 的敘述，要按 ***"n"*** 取消，不然原本的 ***id_rsa*** 會被覆蓋

```
/xxx/xxx/.ssh/id_rsa already exists.
Overwrite (y/n)? n
```



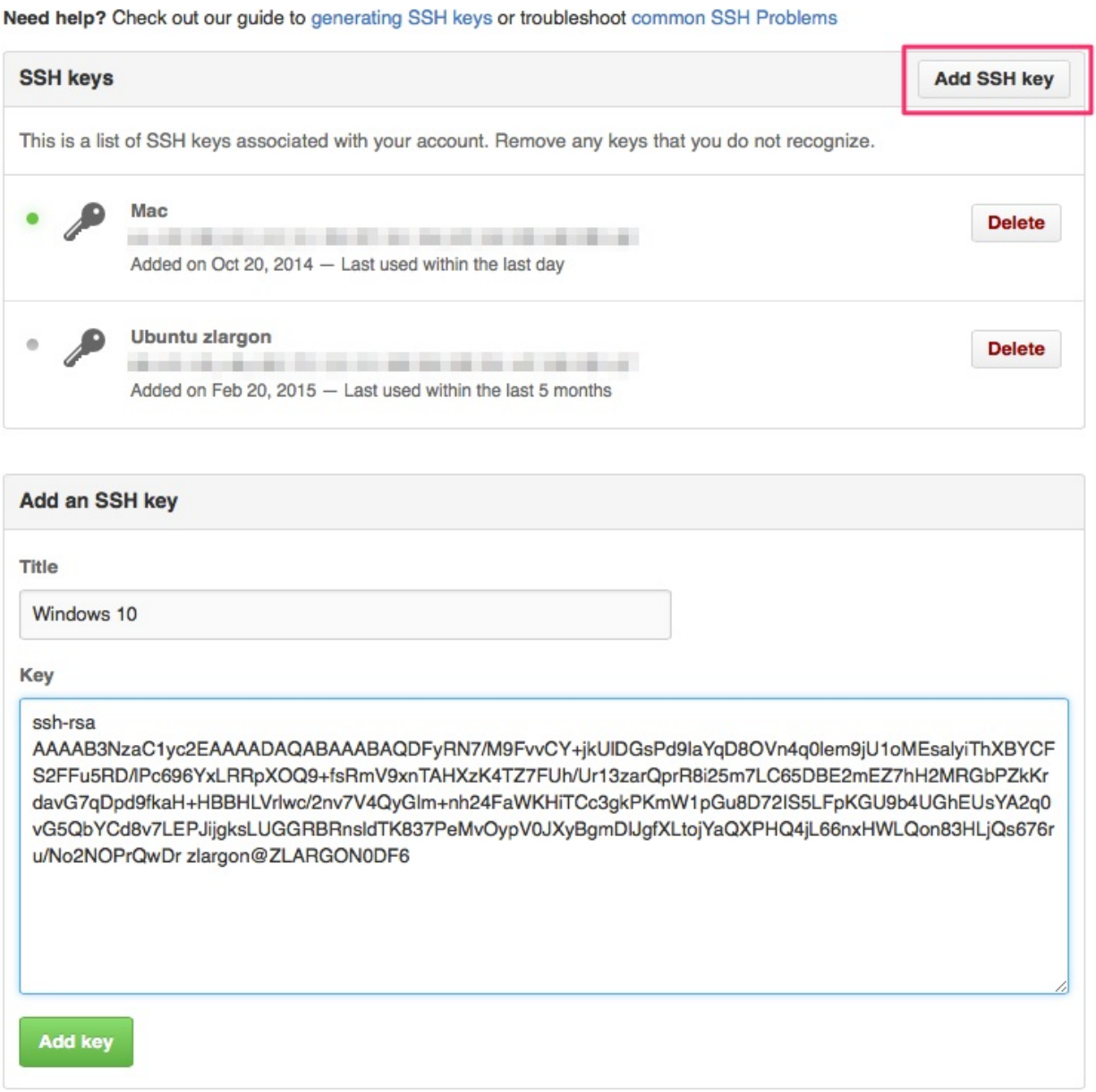
2. 將公鑰（id_rsa.pub）上傳到 Github Server

點選 `Settings` > `SSH Keys`



點右上角的 **"Add SSH key"**

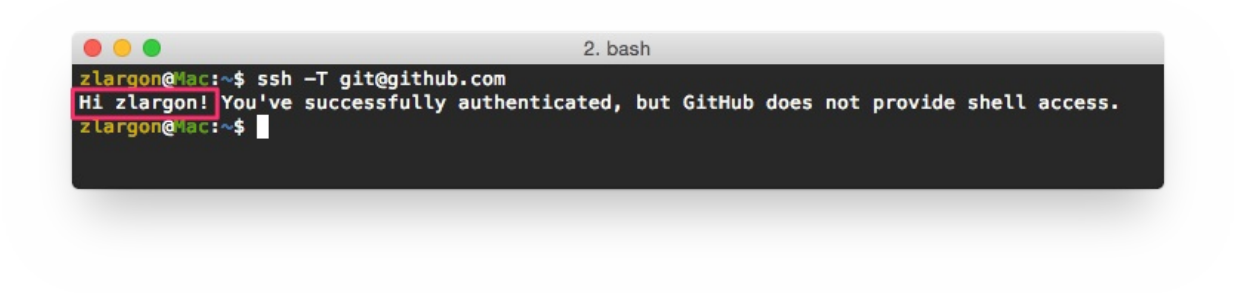
把 `~/.ssh/id_rsa.pub` 的內容複製貼上，點選 **"Add key"** 送出，這樣就大功告成了



測試 *SSH*

```
$ ssh -T git@github.com
```

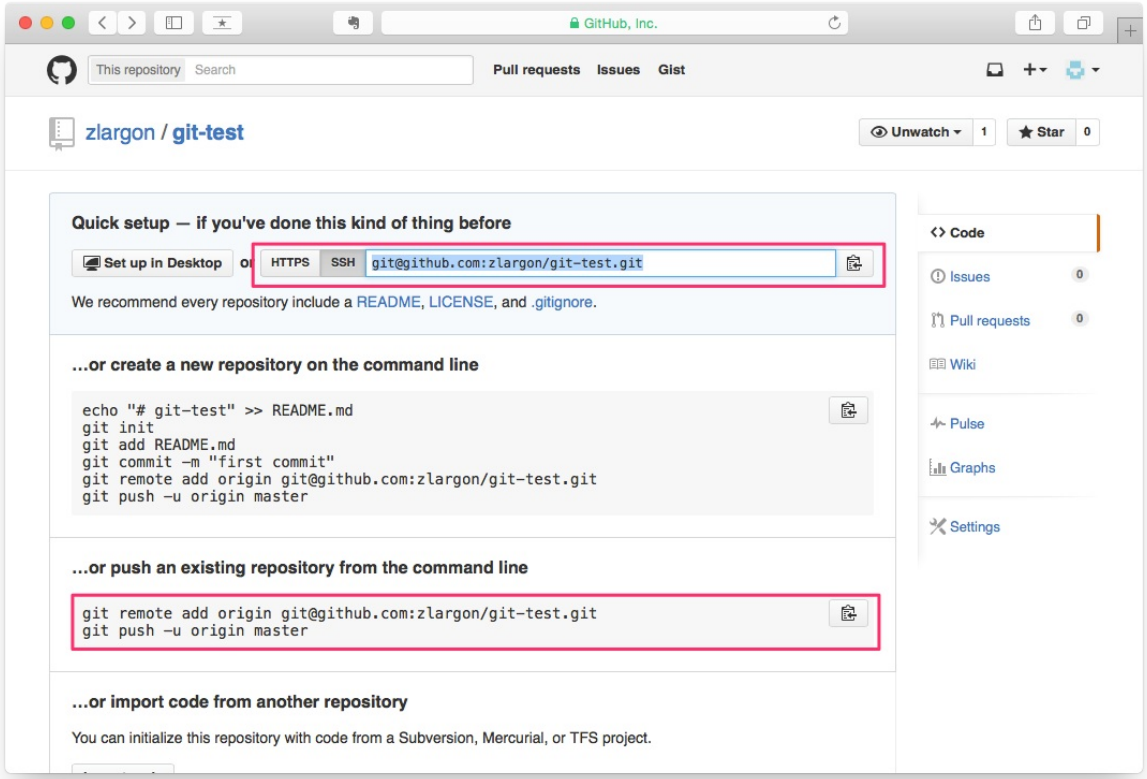
如果有出現以下字樣，就表示 SSH 設定成功！



初次連線 Github Server 可能會出現以下訊息，輸入 `yes` 即可

```
The authenticity of host 'github.com (207.97.227.239)' can't be
established.
# RSA key fingerprint is
16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.
# Are you sure you want to continue connecting (yes/no)? yes
```

設定 Repo URL



使用 `git remote add <short name> <repo url>` 新增遠端 repository URL

在我們新增 Github 專案之後，下方有指令提示，其中有這一行：

```
$ git remote add origin git@github.com:zlargon/git-test.git
```

意思是說，把 repo url 加到遠端的清單，並且以 `origin` 當作 short name

一般來說，Git 習慣把主要的 remote 命名爲 `origin`，不過我們也可以取其他的名字

使用 `git remote -v` 查看設定好的 remote 資訊

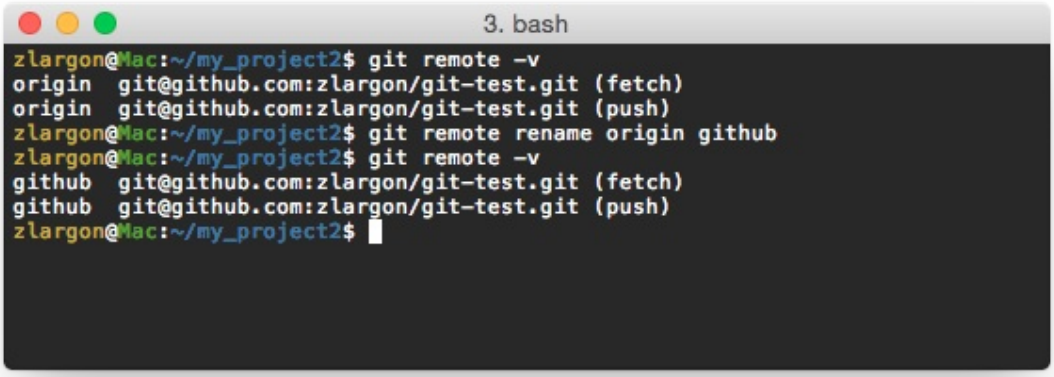
```
$ git remote -v
```

我們已經設定好 short name 之後，往後所看到的 **`origin`** 就代表了 `git@github.com:zlargon/git-test.git` 這個 repo URL

使用 `git remote rename <short name> <new name>` 修改 remote name

一般我習慣會用 **`github`** 當作 short name，因爲這樣比較有實質的意義

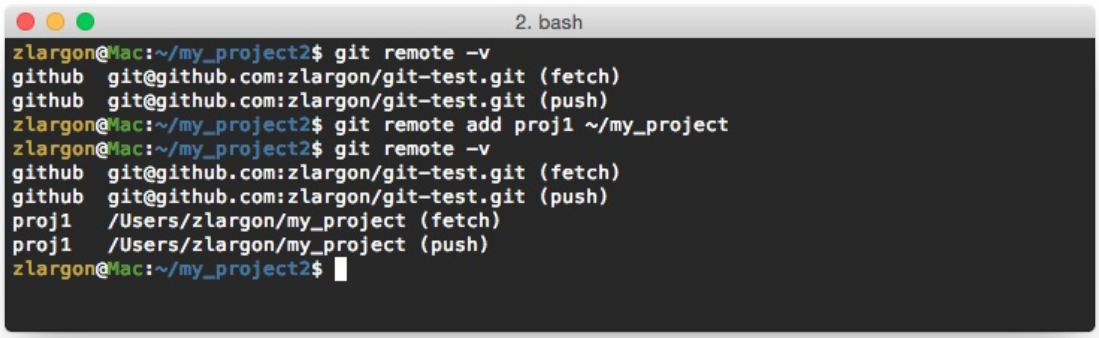
```
$ git remote rename origin github
```



或是用 `git remote rm <short name>` 把他刪掉，再重新加入也可以

Remote 的特性

- **Git** 可以設定多組 **remote**
- **Repo URL** 也可以是本機端其他 **git project** 的資料夾路徑



上傳分支

在前面我們將 Repo URL 設定好之後，接下來就是要上傳程式碼

Git 在上傳程式碼到 server 時，必須是以分支爲單位

使用 `git push <remote name> <branch name>` 上傳分支

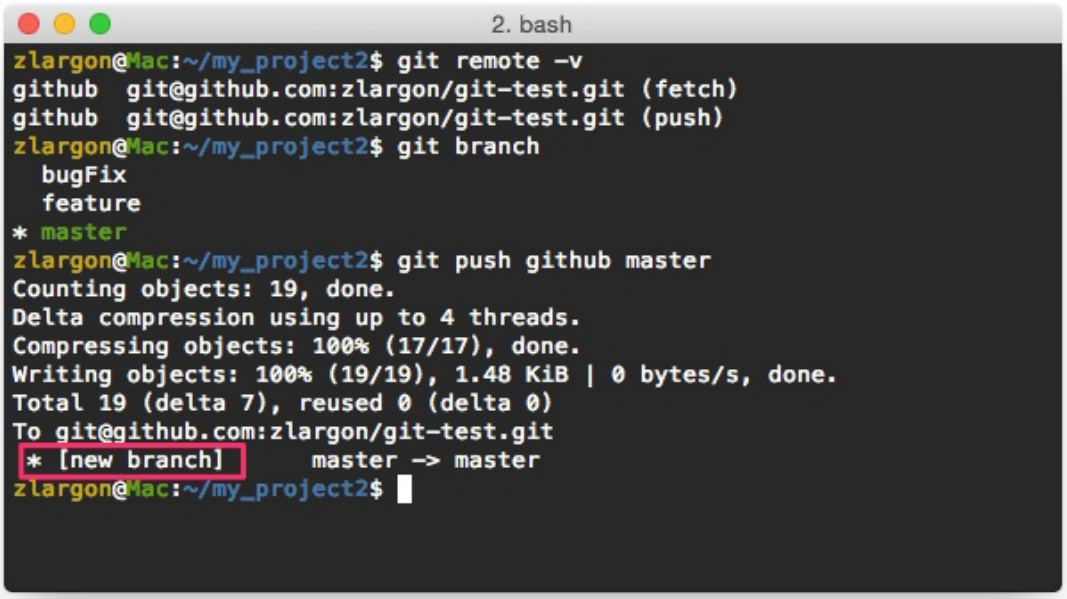
例如說我現在想把 `master` 上傳到 Github

```
$ git push github master        # 不需要先 checkout 到 master branch
```

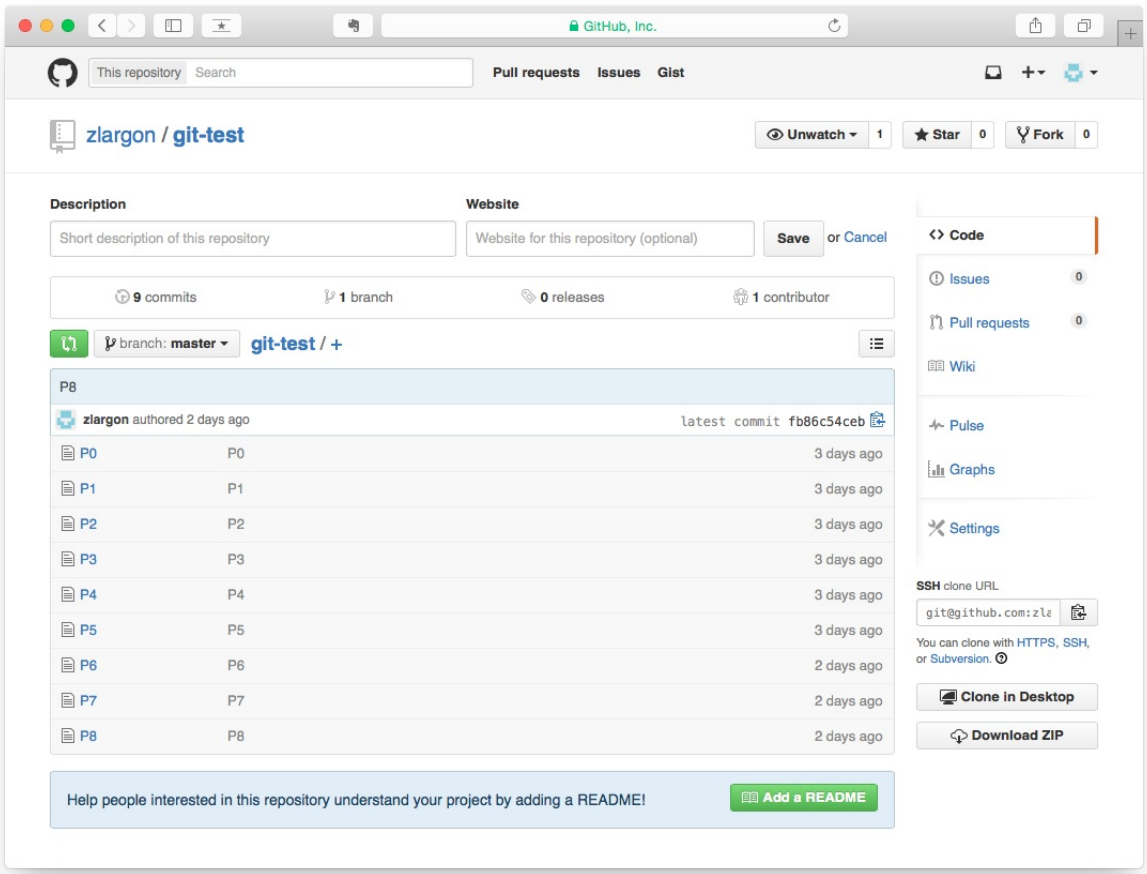
他會將本機端的 `master` 分支，上傳到 server 上

如果 server 上沒有 `master` 這個分支，他就會自動在 server 上添加 `master` 分支

> 注意：
> 本機端的分支名稱必須跟 server 上的分支名稱完全相同



這時候重新整理 Github 網頁，就會出現新的 `master` 分支了



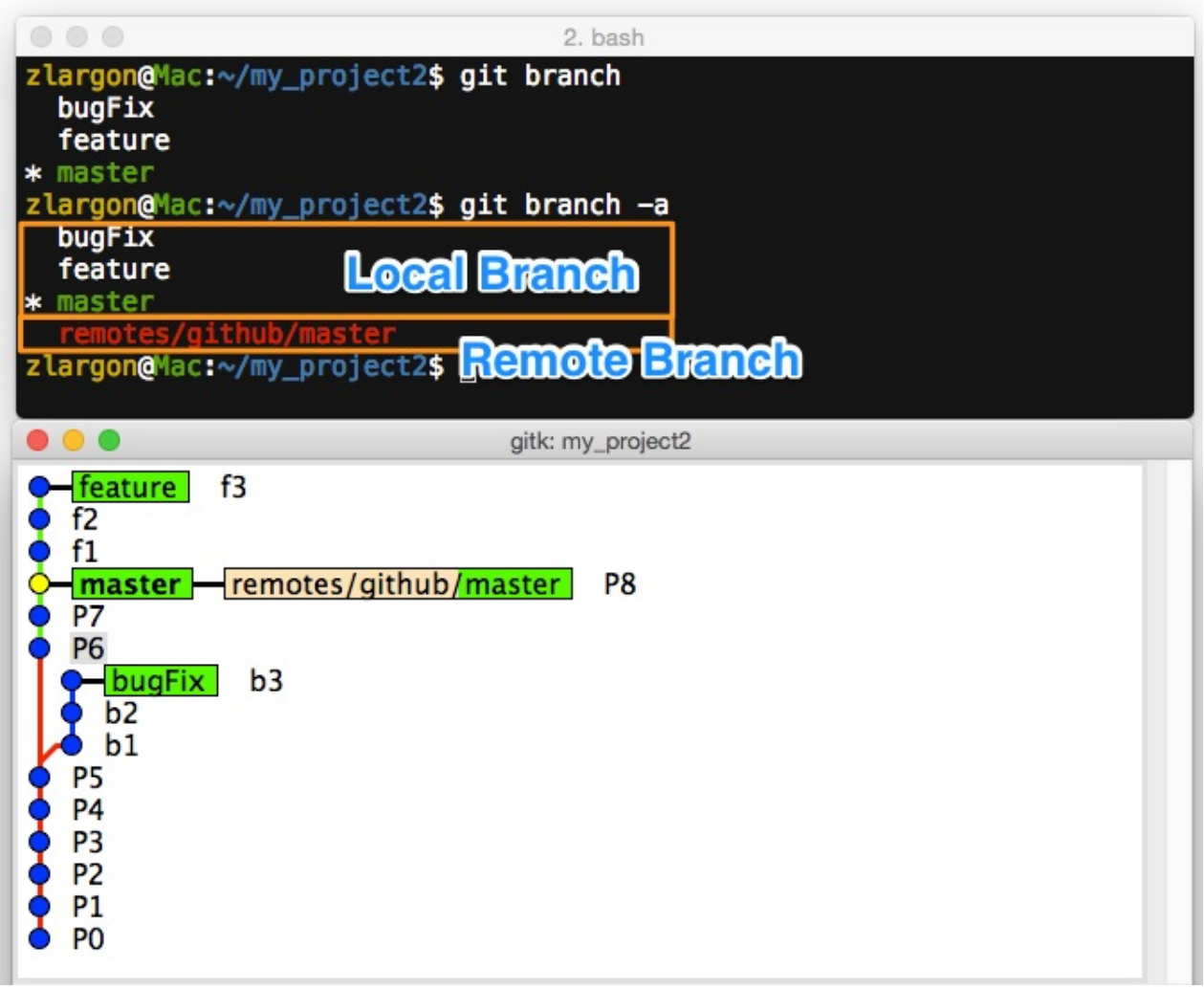
使用 `git branch -a` 查看本機端及遠端的分支

我們之前有教過使用 `git branch` 來查看分支，但是這只能看到本機端的分支

如果想要看到遠端的分支的話，就必須加上 `-a` 或是 `--all` 參數

遠端分支名稱：`remotes/<remote name>/<branch name>`

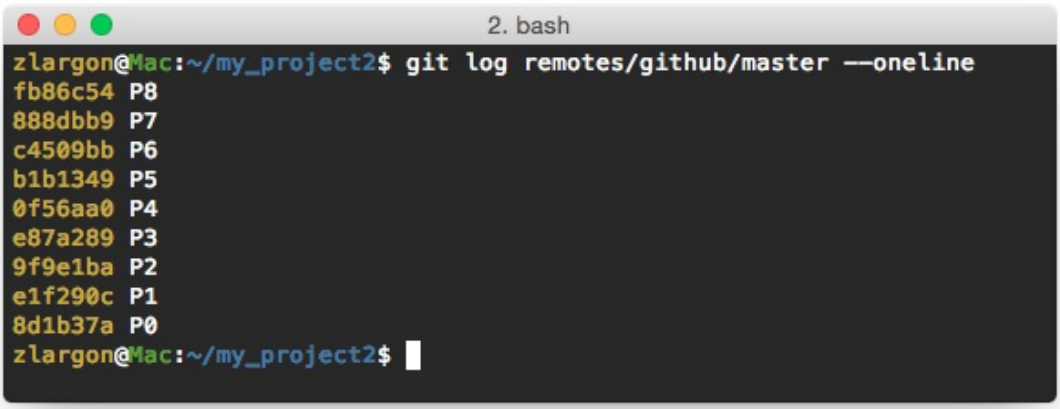
```
$ git branch -a
$ git branch --all
```



遠端分支的內容會完全跟 server 上的一致

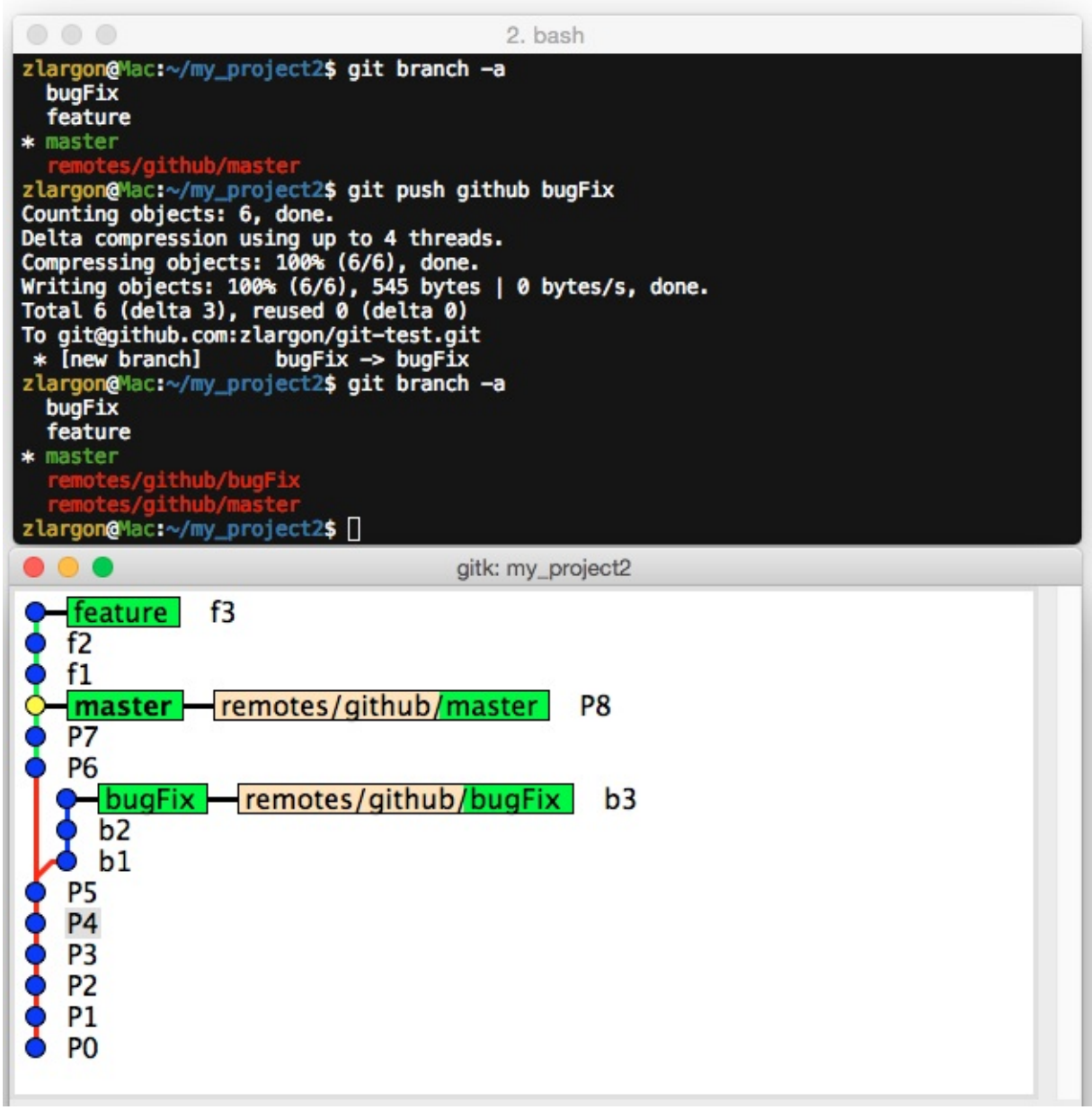
我們可以用 `git log` 查看遠端分支的內容，但是我們不能修改他的內容

```
$ git log remotes/github/master --oneline
```

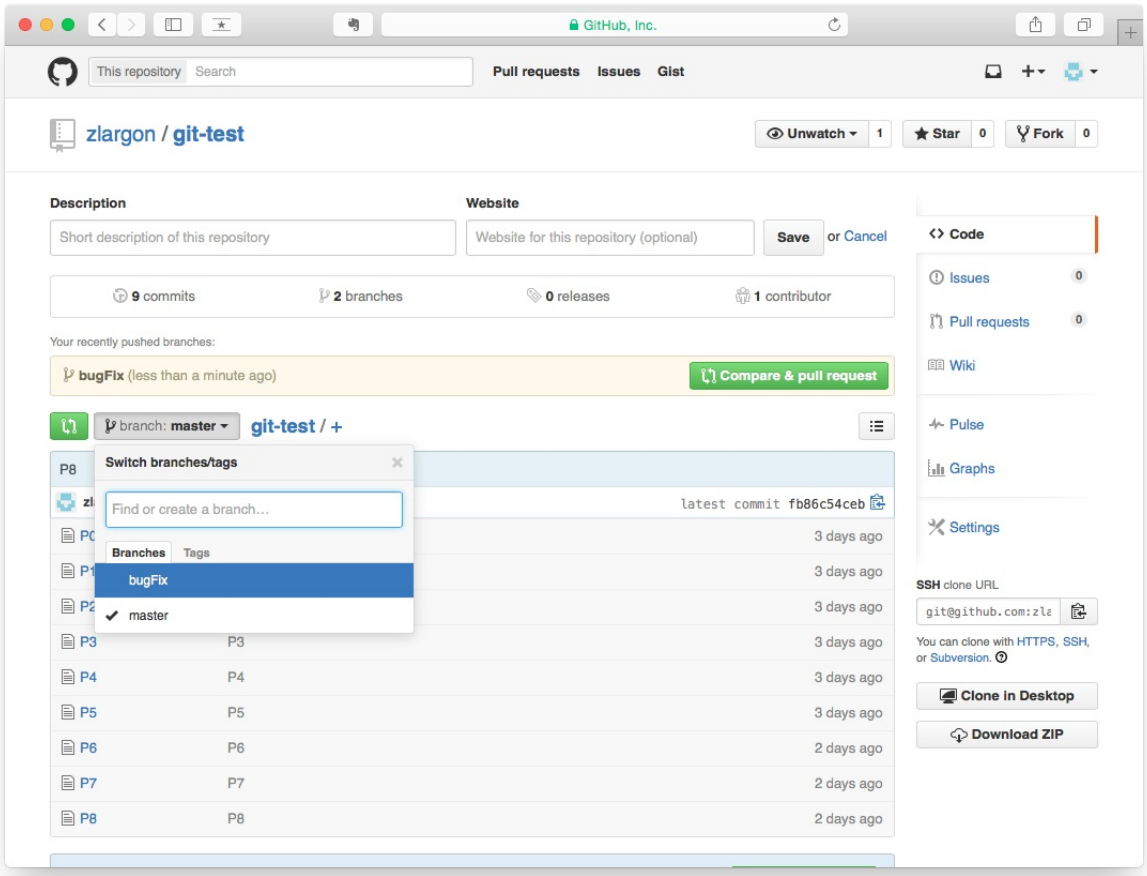


我們接下來再把 `bugFix` 分支上傳到 server

```
$ git push github bugFix        # 不需要先 checkout 到 bugFix branch
```



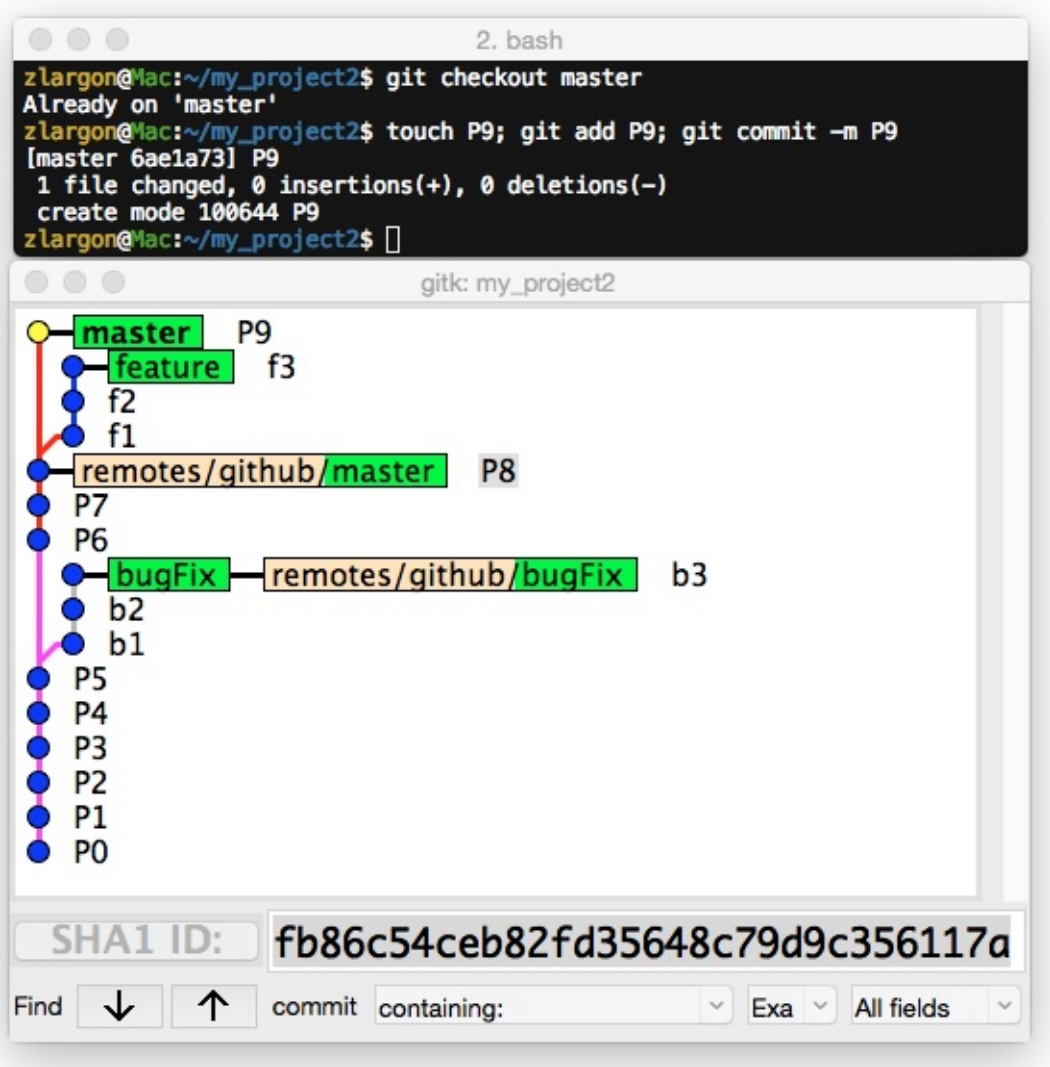
這時候重新整理 Github 網頁，就會出現新的 `bugFix` 分支了



上傳程式碼的流程

先在本機端修改 / 更新分支，然後在上傳到 server

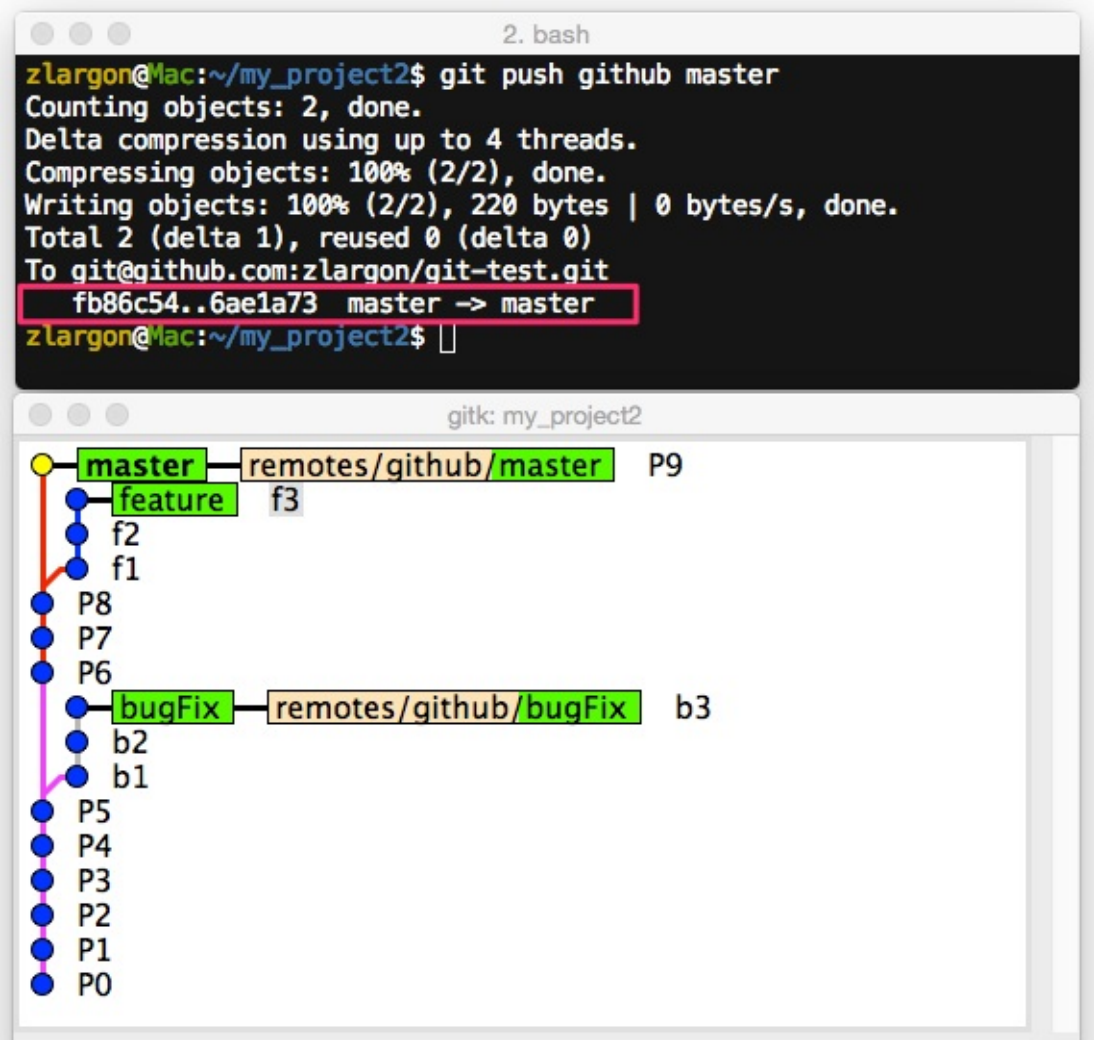
現在我們就在 local 的 `master` 分支提交一個新的 patch P9



這時候我們可以看到 `master` 與 `remotes/github/master` 的分支目前是不同步的情況

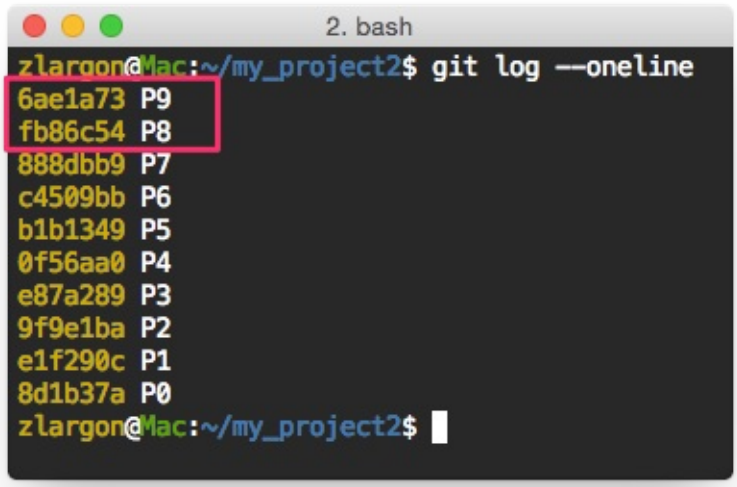
現在我們上傳 `master` 到 Github Server，更新遠端的 repository

```
$ git push gihub master
```




```
fb86c54..6ae1a73  master -> master
```

這表示遠端的 `master` 分支已經從 fb86c54（P8）更新成 6ae1a73（P9）

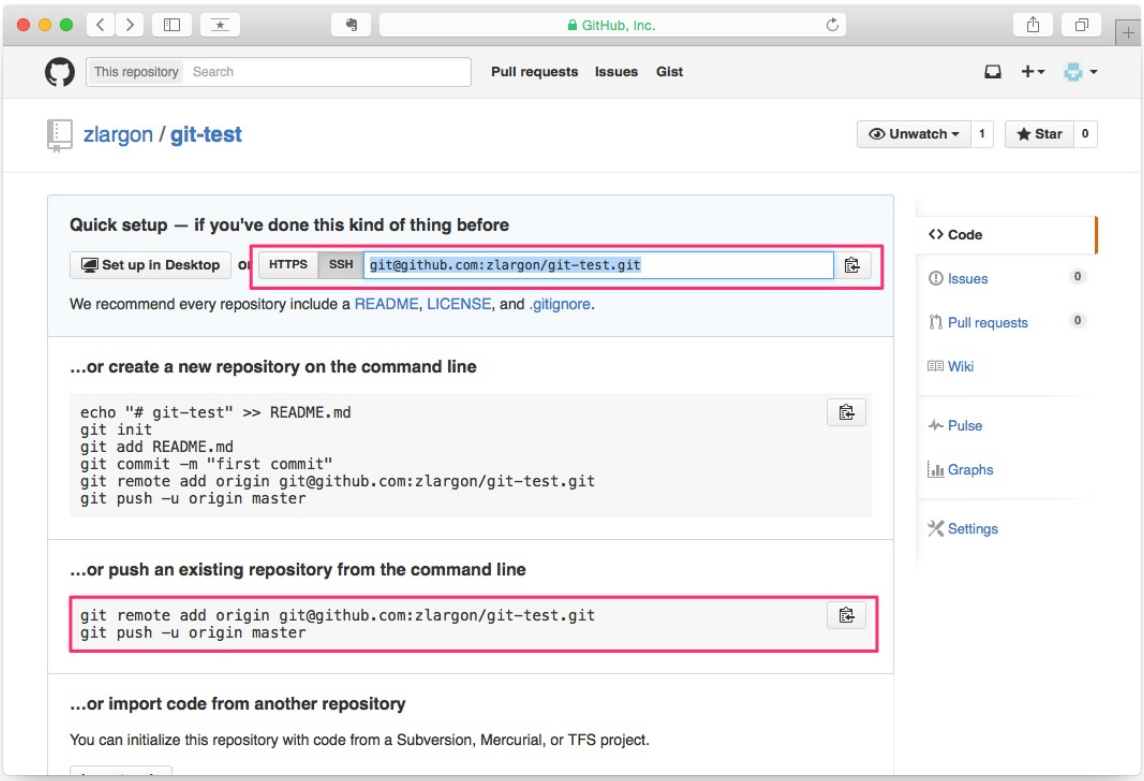


本章回顧

- 使用 `git branch -a` 查看本機端及遠端的分支
- 使用 `git push <remote name> <branch name>` 上傳分支

設定 Upstream

在 "建立專案" 的時候，Github 有提供我們一些指令提示



其中他做 `git push` 的時候，有帶一個參數 `-u`

這是什麼意思呢？

使用 `git push -u <remote name> <branch name>` 上傳分支，並且追蹤遠端的分支

參數 `-u` 等同於 `--set-upstream`，設定 upstream 可以使分支開始追蹤指定的遠端分支

只要做過一次 `git push -u <remote name> <branch name>`，並且成功 push 出去；本機端的 `master` 就會被設定去追蹤遠端的 `<remote name>/<branch name>` 分支

設定好 upstream 後，第二次以後要上傳分支時，就只需要透過 `git push` 就可以了，不必再帶 `<remote name>` 跟 `<branch name>` 等參數

之後我們講到指令 `git pull` 也會用得到

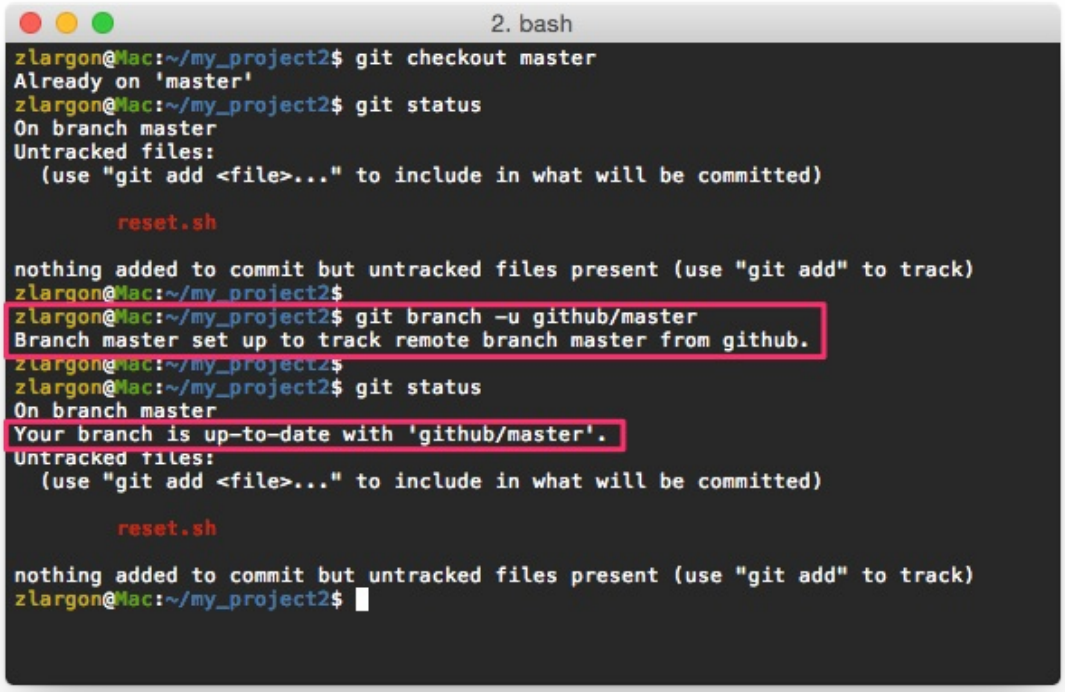
```
$ git push -u github master                 # 第一次上傳
$ git push --set-upstream github master     # 同上

$ ...

$ git checkout master       # 切到 master
$ git push                  # 第二次之後 push 就不用再帶 <remote name> 跟 <branch name>
```

使用 `git branch -u <remote>/<remote branch>` 設定 upstream

`git push -u github master` 指令可以拆解成以下的指令

```
$ git push github master
$ git checkout master
$ git branch -u github/master
```

當設定好分支的 upstream 後，使用 `git status` 會顯示追蹤的訊息

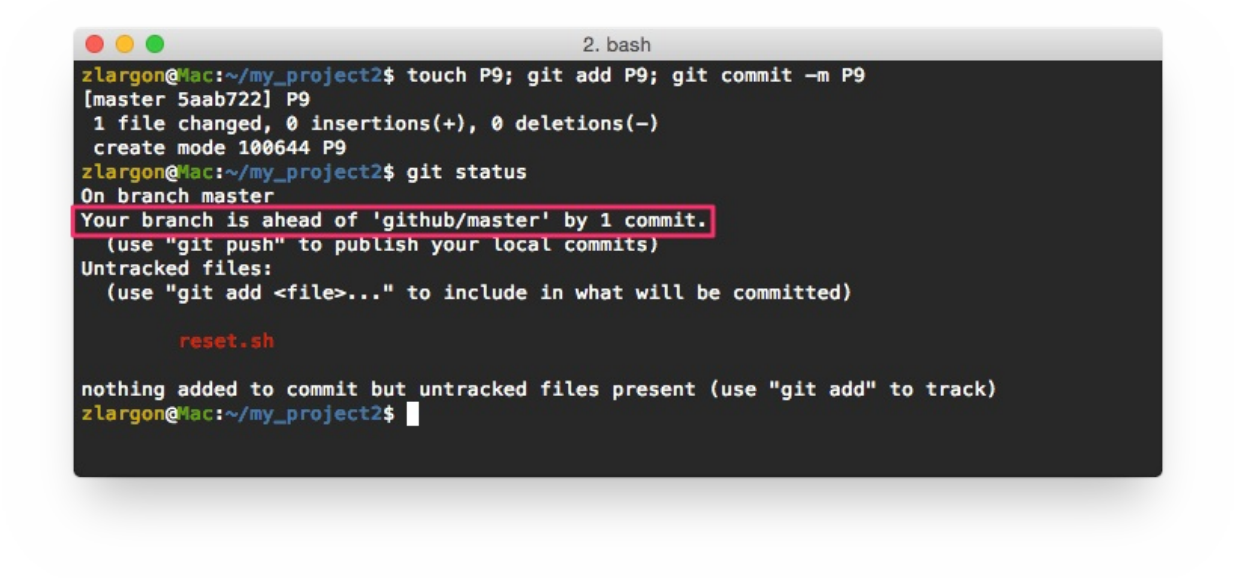


```
Your branch is up-to-date with 'github/master'.
```

表示目前 `master` 本機端與遠端的內容分支一致

其他他就只是把 `master` 去跟 `remotes/guthub/master` 做比對而已

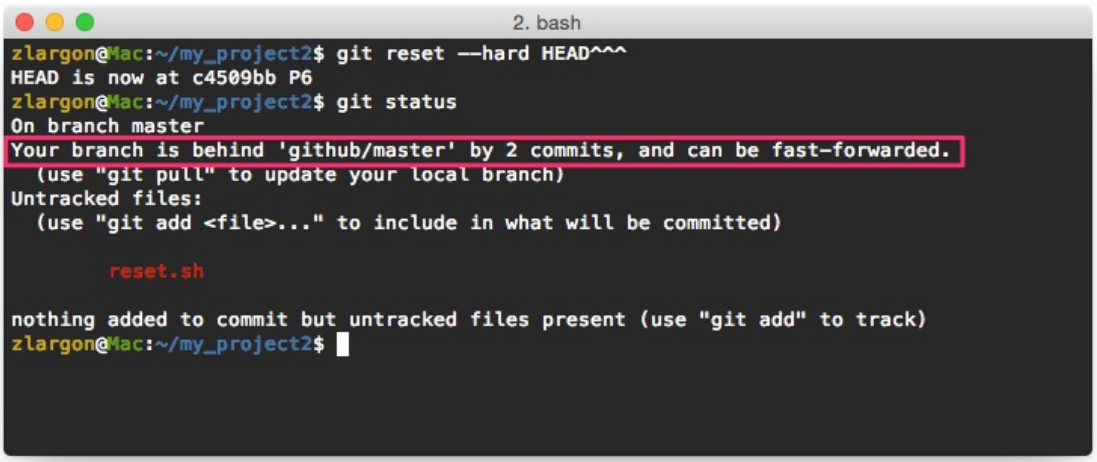
我們現在在 `master` 提交新的 patch P9




```
Your branch is ahead of 'github/master' by 1 commit.
```

他會顯示目前 `master` 分支的位置領先 `remotes/github/master` 一個 patch

如果我們倒退回前三個 patch




```
Your branch is behind 'github/master' by 2 commits, and can be
fast-forwarded.
```

則會看到目前 `master` 的分支落後於 `remotes/github/master` 兩個 patch

指令代換

```
$ git checkout master
$ git branch -u github/master
```

以上兩行指令可以合併成一行

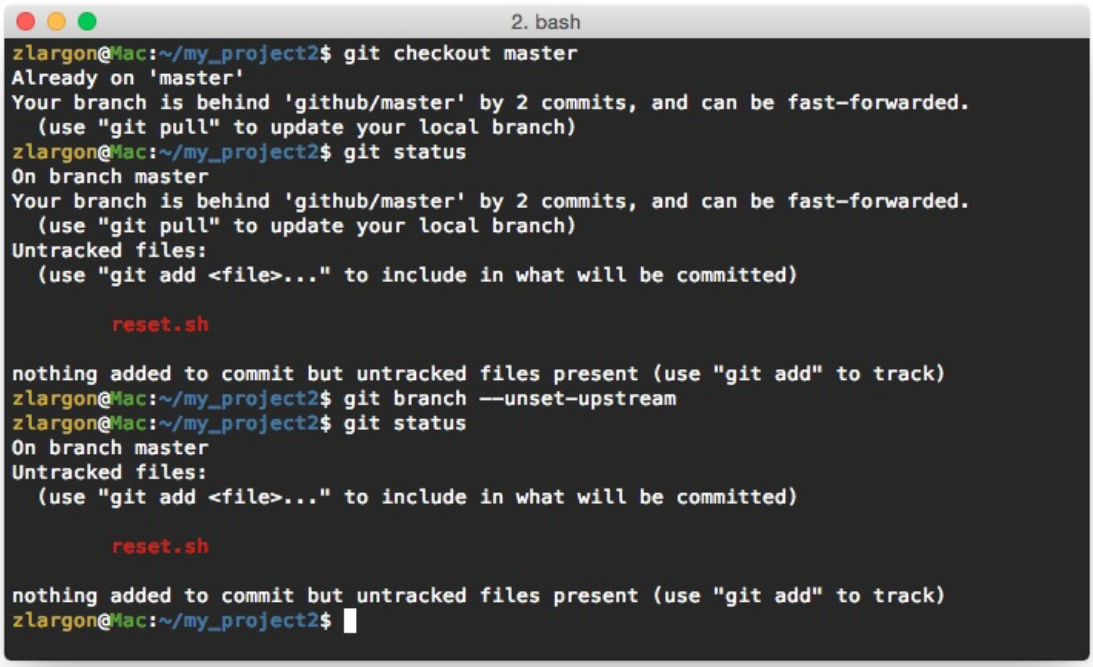
```
$ git branch -u github/master master        # 可以不用先切到 master
```

而 `-u` 也可以換成 `--set-upstream-to=<remote>/<branch>`

```
$ git branch --set-upstream-to=github/master master
```

使用 `git branch --unset-upstream` 取消追蹤遠端分支

```
$ git checkout master
$ git branch --unset-upstream
```



這個指令也可以寫成一行

```
$ git branch --unset-upstream master        # 可以不用先切到 master
```

本章回顧

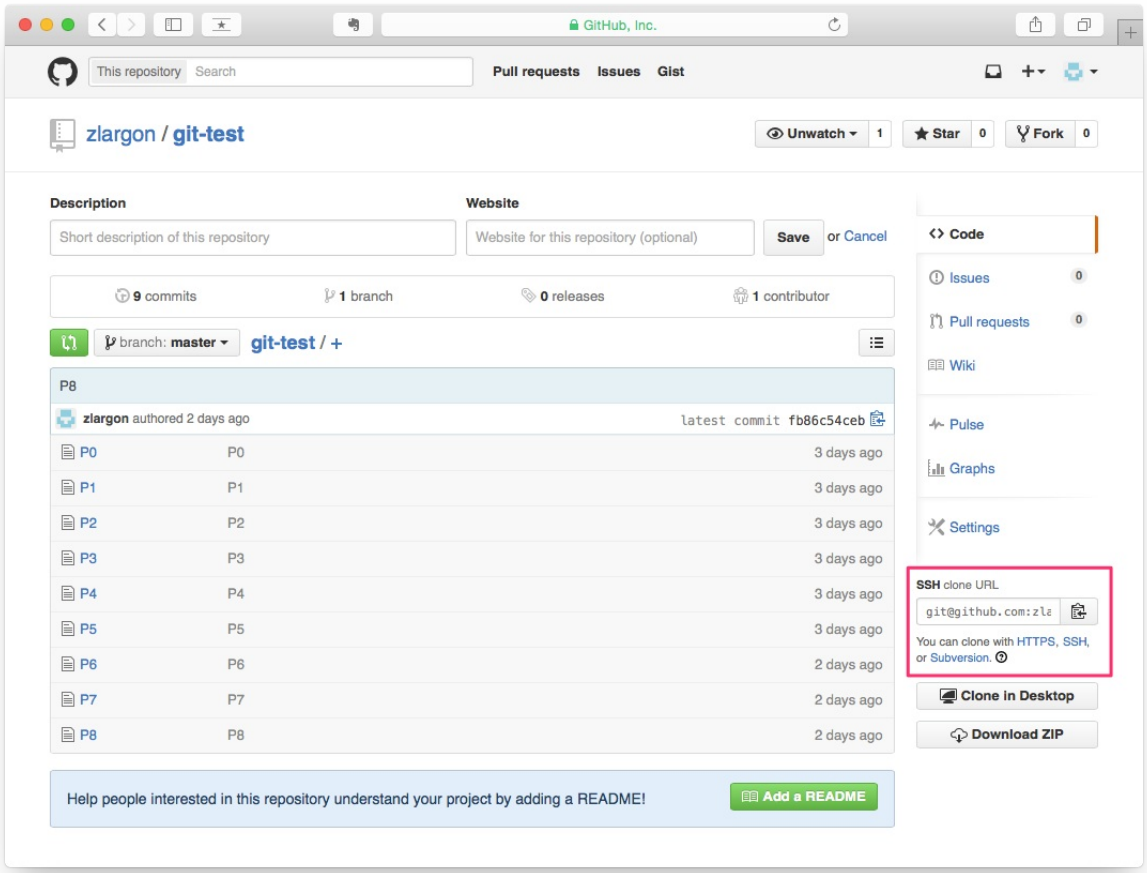
- 使用 `git push -u <remote name> <branch name>` 上傳分支，並且追蹤遠端的分支

  > 之後只需要透過 `git push` 就可以上傳分支，不必帶 `<remote name>` 跟 `<branch name>`

- 使用 `git branch -u <remote>/<remote branch>` 設定 upstream
- 使用 `git branch --unset-upstream` 取消追蹤遠端分支

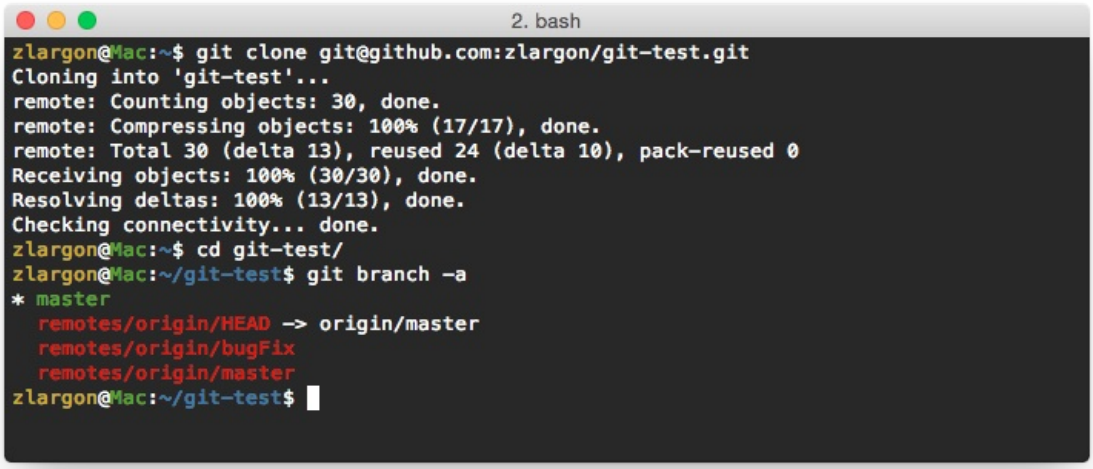
複製 / 下載專案

在我們成功把我們的專案上傳到 Github 之後，就可以透過 repo URL 下載專案



使用 `git clone <repo URL>` 下載專案

```
$ git clone git@github.com:zlargon/git-test.git
$ cd git-test
$ git branch -a
```



使用 `git clone` 的時候，git 預設會建立一個跟 project name 同名的資料夾（git-test）

用 `git branch -a` 檢視之後，發現有 1 個 local 分支和 3 個 remote 分支

```
Local Branch：
  master

Remote Branch：
  remotes/origin/HEAD -> origin/master
  remotes/origin/bugFix
  remotes/origin/master
```

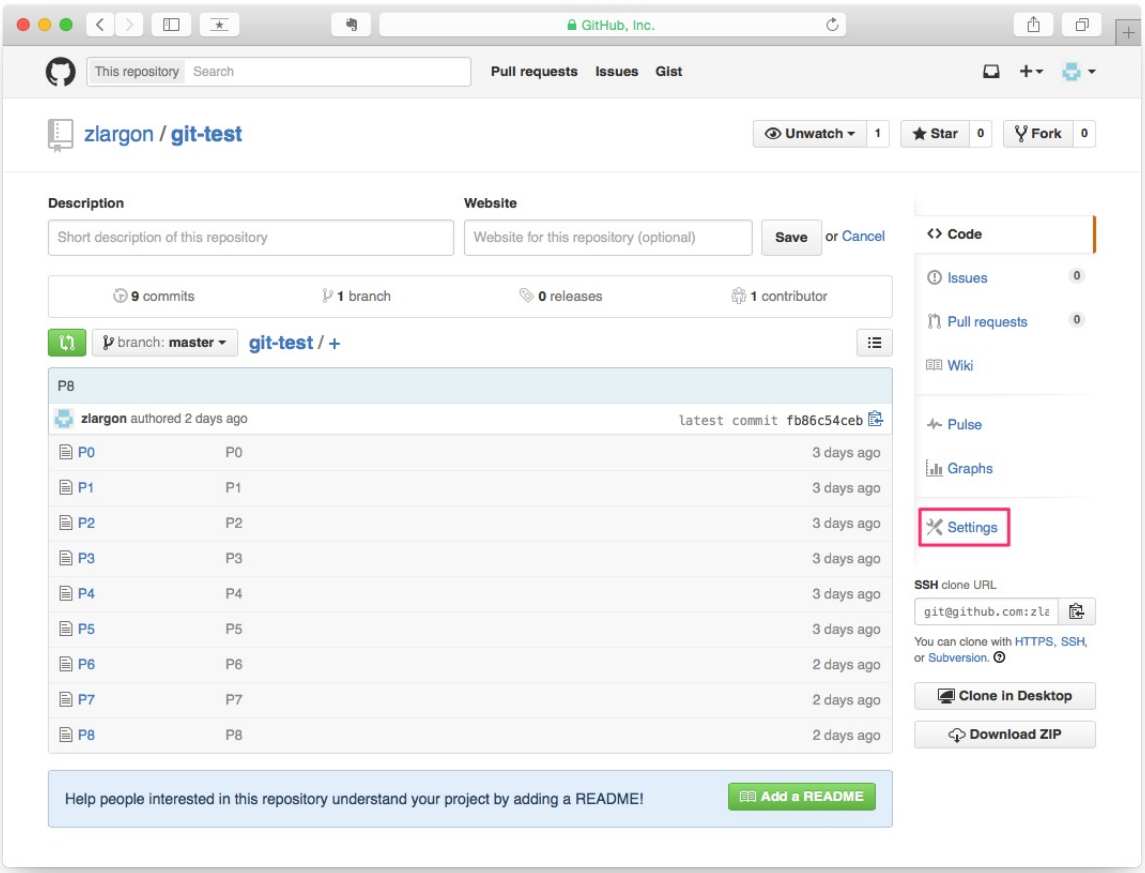
Git 預設的 remote 名稱爲 **`origin`**（可以用 `git remote rename` 來修改名稱）

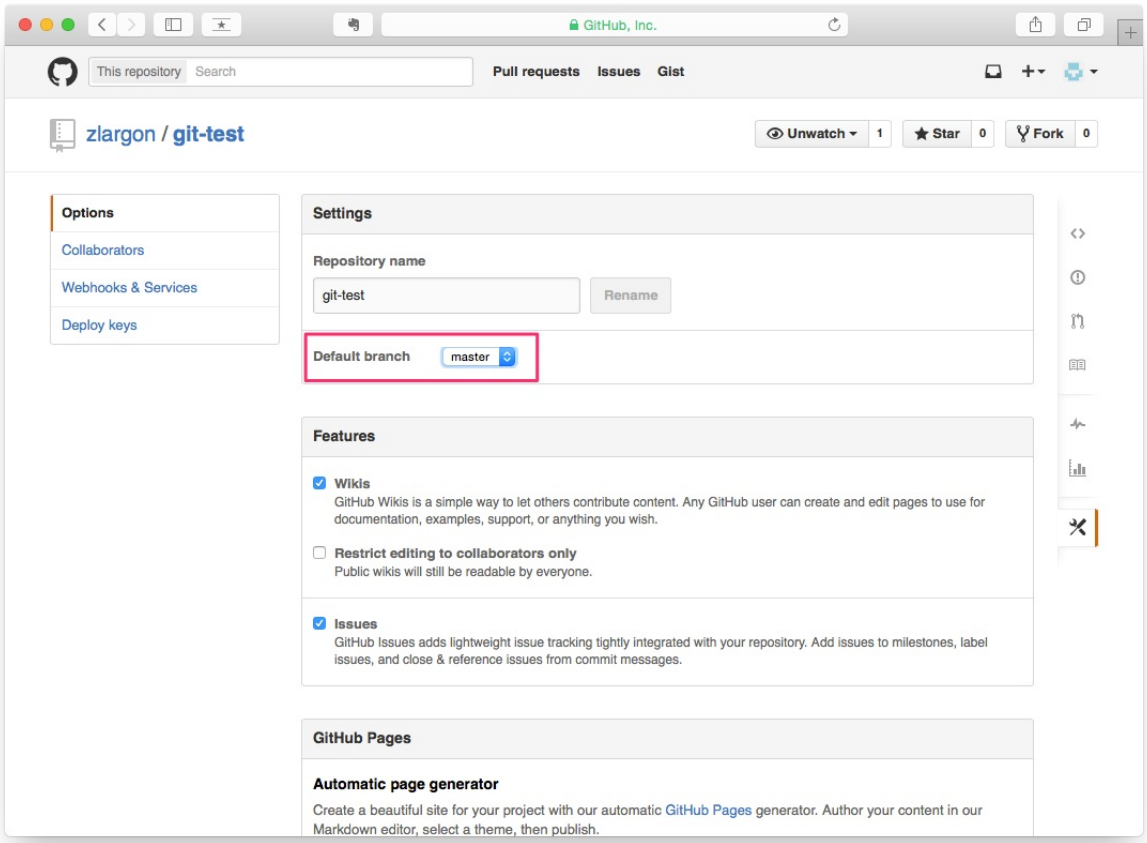
可是我們之前並沒有看過 **`remotes/origin/HEAD`**，這支個分支是什麼的呢？

`remotes/origin/HEAD` 是指專案的預設分支

而後面有一個箭頭指向 **`origin/master`**，表示專案的預設分支是 `master`

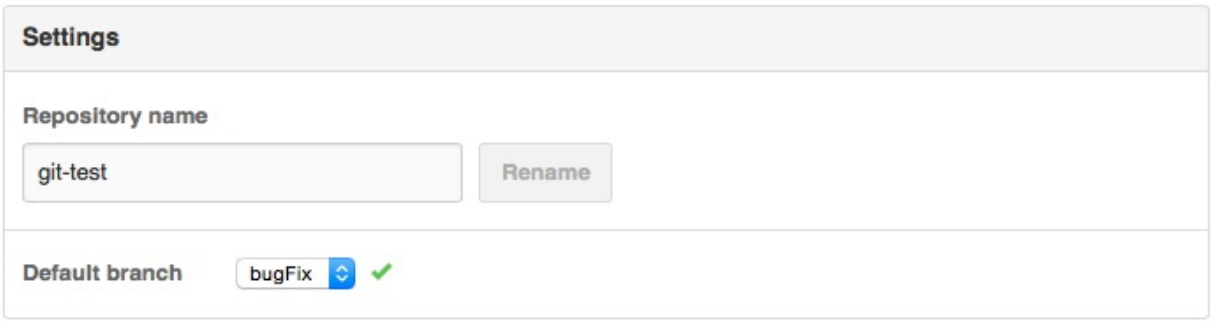
預設分支可以從 Github 的網頁介面去設定

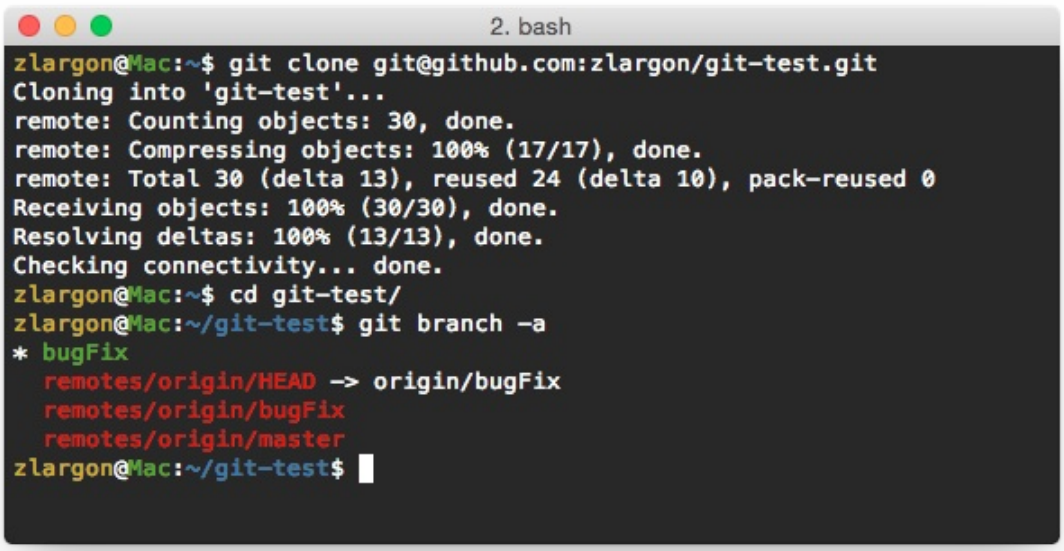

GitHub, Inc.
This repository
Search
Pull requests
Issues
Gist
zlargon / git-test
Unwatch 1
Star 0
Fork 0
Description
Website
Short description of this repository
Website for this repository (optional)
Save
or Cancel
9 commits
1 branch
0 releases
1 contributor
branch: master
git-test / +
P8
zlargon authored 2 days ago
latest commit fb86c54ceb
P0 P0 3 days ago
P1 P1 3 days ago
P2 P2 3 days ago
P3 P3 3 days ago
P4 P4 3 days ago
P5 P5 3 days ago
P6 P6 2 days ago
P7 P7 2 days ago
P8 P8 2 days ago
Help people interested in this repository understand your project by adding a README!
Add a README
Code
Issues 0
Pull requests 0
Wiki
Pulse
Graphs
Settings
SSH clone URL
git@github.com:zla
You can clone with HTTPS, SSH, or Subversion.
Clone in Desktop
Download ZIP



若我們把它改成 `bugFix`

那麼 **`remotes/origin/HEAD`** 就會變成指向 **`origin/bugFix`**

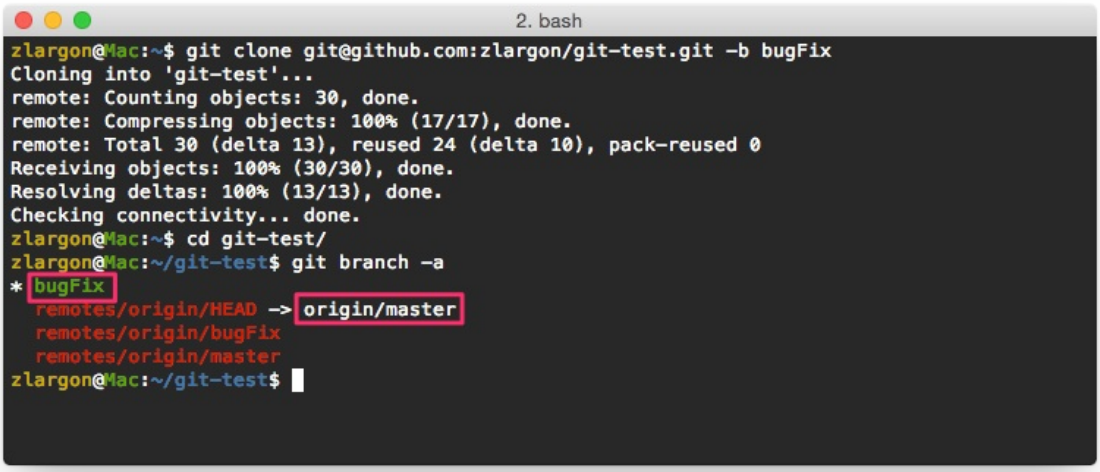




使用 `git clone <repo URL> -b <branch name>` 指定分支

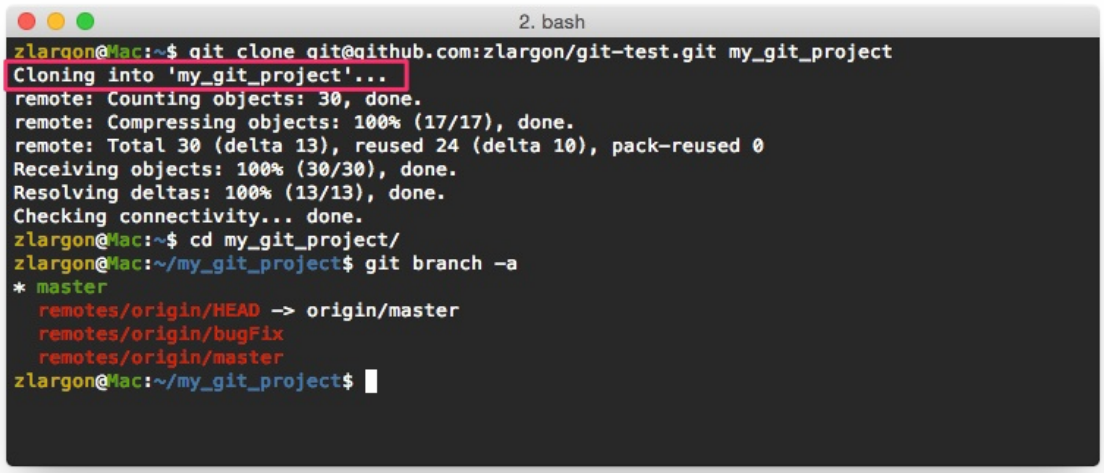
若沒有指定分支的話，git 就會使用 "專案的預設分支" 去建 local branch

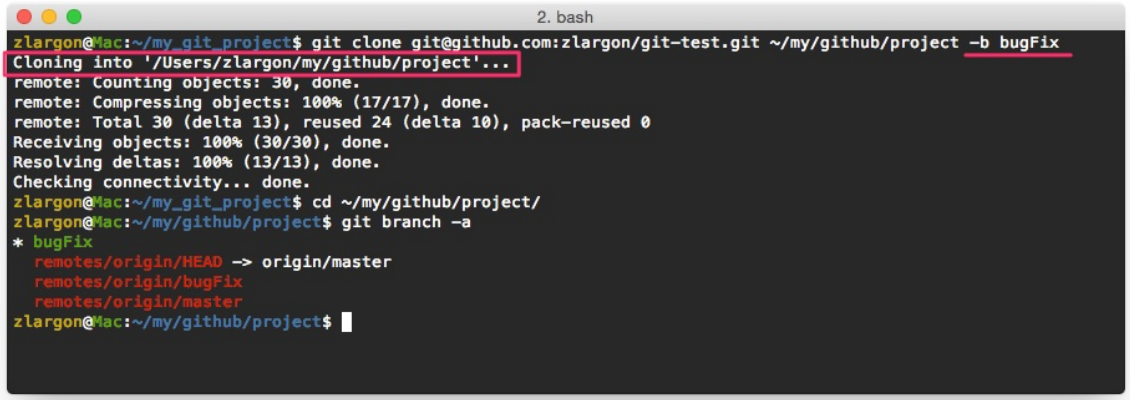
```
$ git clone git@github.com:zlargon/git-test.git -b bugFix
```



使用 `git clone <repo URL> <folder name/path>` 下載到指定位置

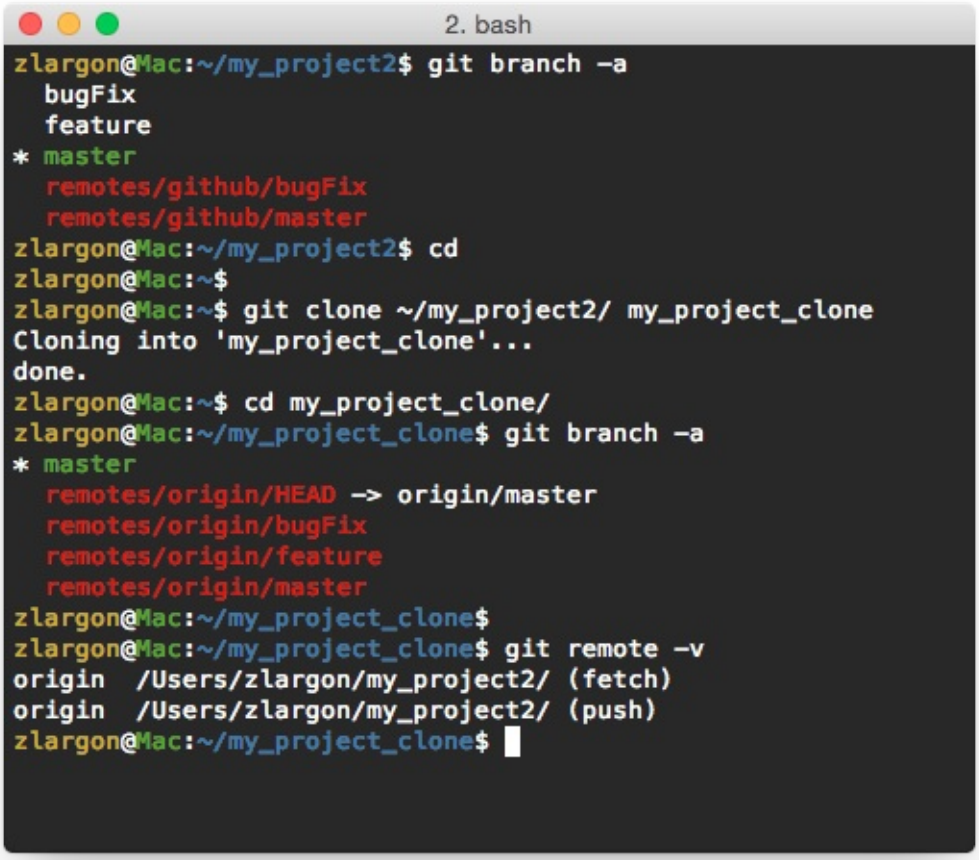
如果不想要用預設檔名的話，可以自己設定資料夾名稱或是資料夾的路徑





使用 `git clone <local project>` 建立專案副本

除了可以下載 Github 的專案之外，也可以爲本機端的 git project 建立專案副本



本章回顧

- 使用 `git clone <repo URL>` 下載專案
- 使用 `git clone <repo URL> -b <branch name>` 指定分支
- 使用 `git clone <repo URL> <folder name/path>` 下載到指定位置
- 使用 `git clone <local project>` 建立專案副本

同步遠端分支

在我們把 code 放到 server 之後，我們就可以在任何工作地從 Github 下載下來，並且開始進行開發

假設我們同時有 A, B, C 電腦，都 clone 了這個專案

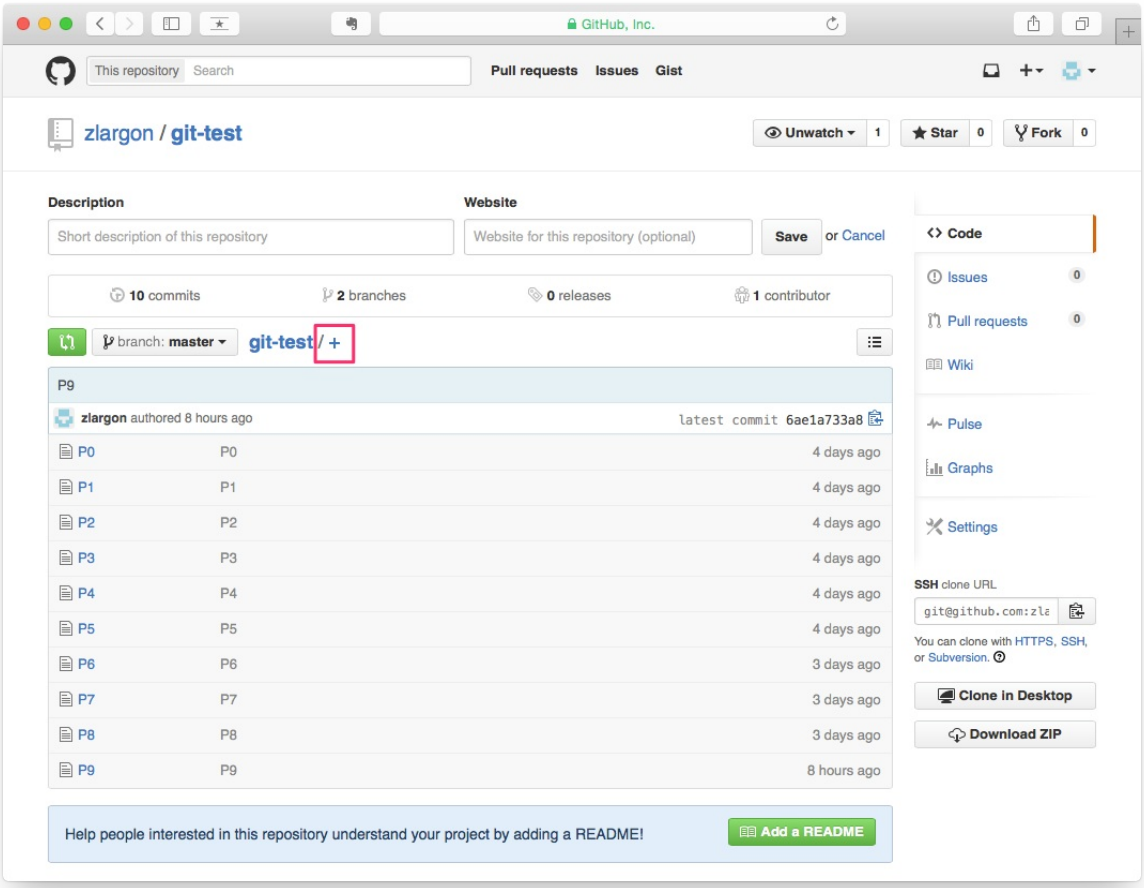
假設我們從電腦 A 上傳了一個 patch 到 server

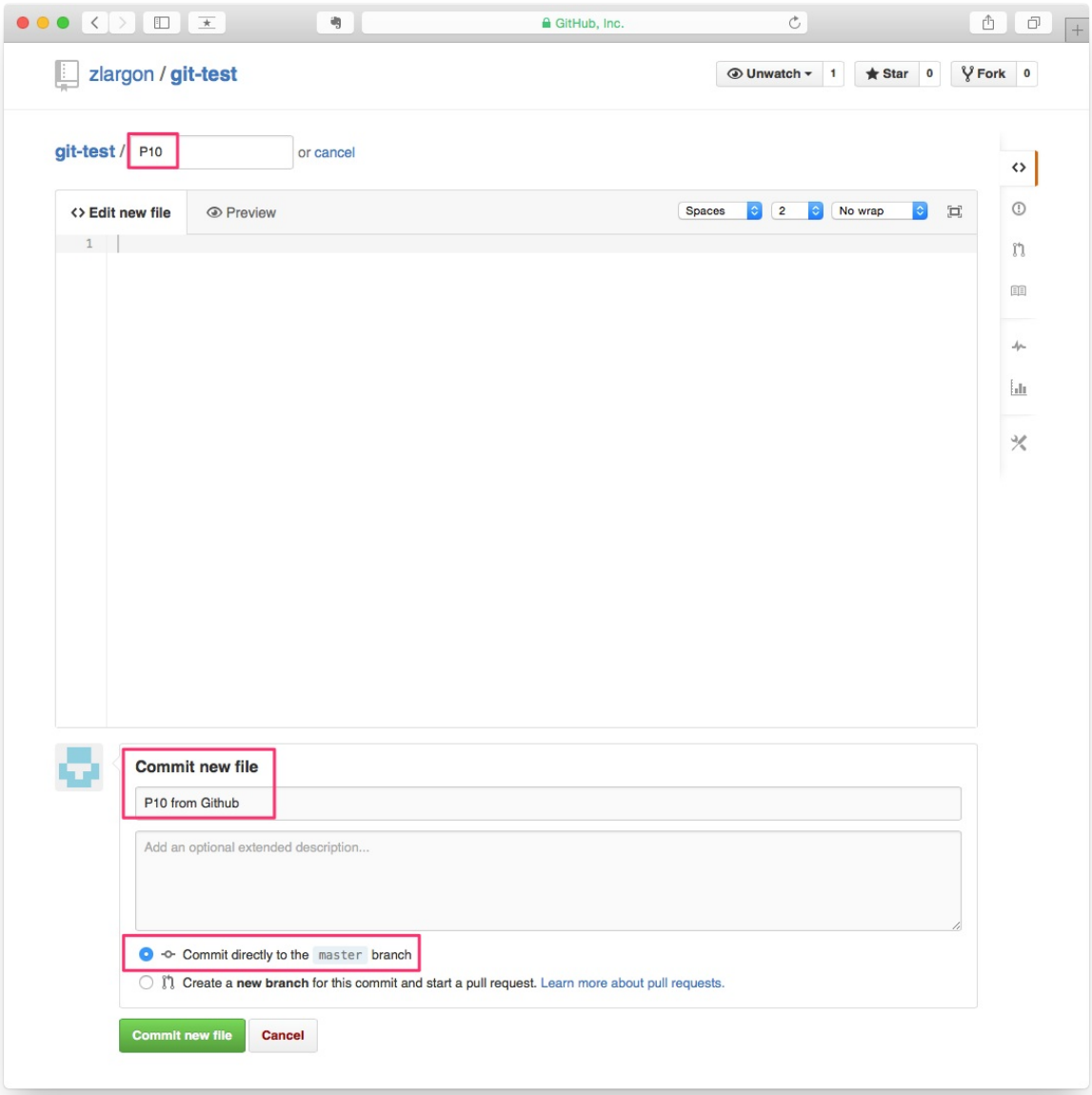
但對於電腦 B, C 來說，他們 local 端的 repo 會與遠端的不同步

電腦 B, C 甚至自己另外還上了新的 patch 但是還沒有上傳到 server

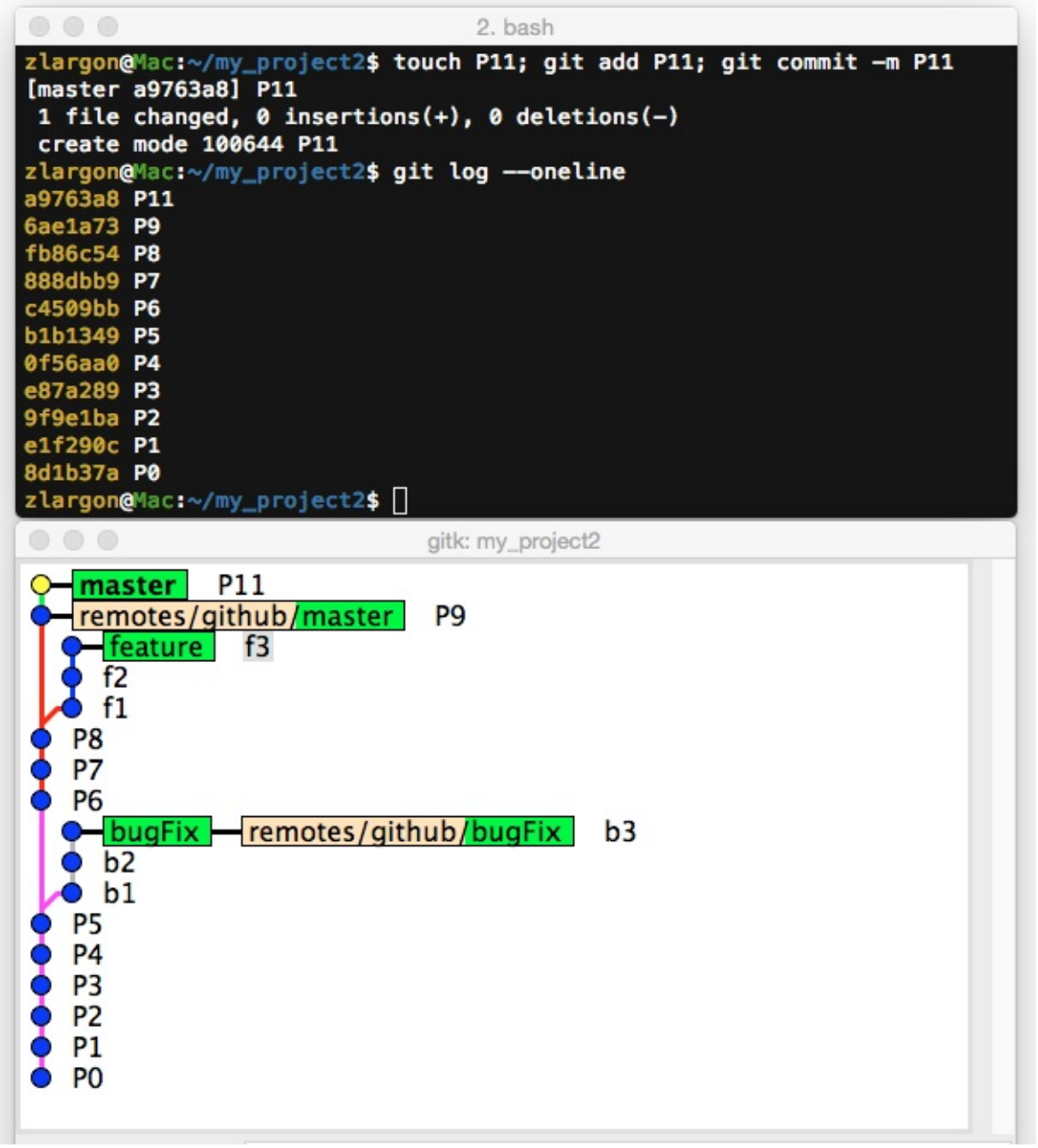
這時候我們就要來處理同步的問題

我們先直接從 Github 上，直接提交一個空白的檔案 P10



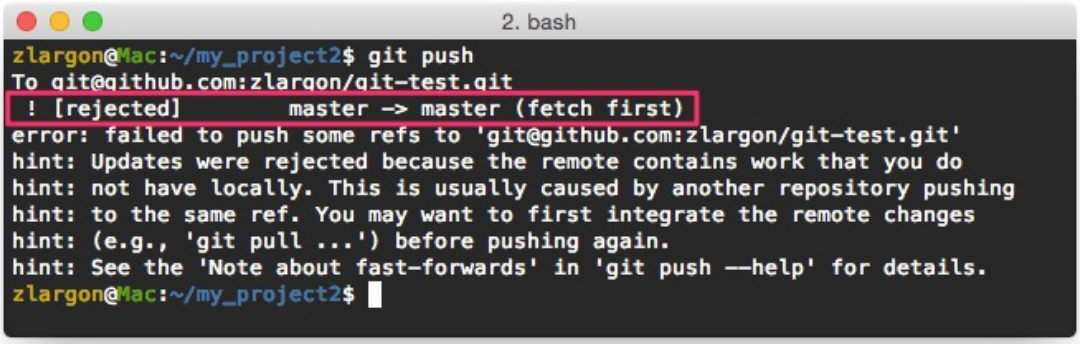


接著我們在本機端提交一個 P11 的 patch



這時候 server 上的檔案，就會跟我們現在本機端的不同步

假設我們不曉得 server 上的 code 已經更新了，就直接使用 `git push` 就會被 server 拒絕



原因是 `master` 與 `github/master` 已經產生了版本衝突

Git Server 預設的行爲是，你上傳的 branch 必須在基於 remote branch 之上，否則就拒絕

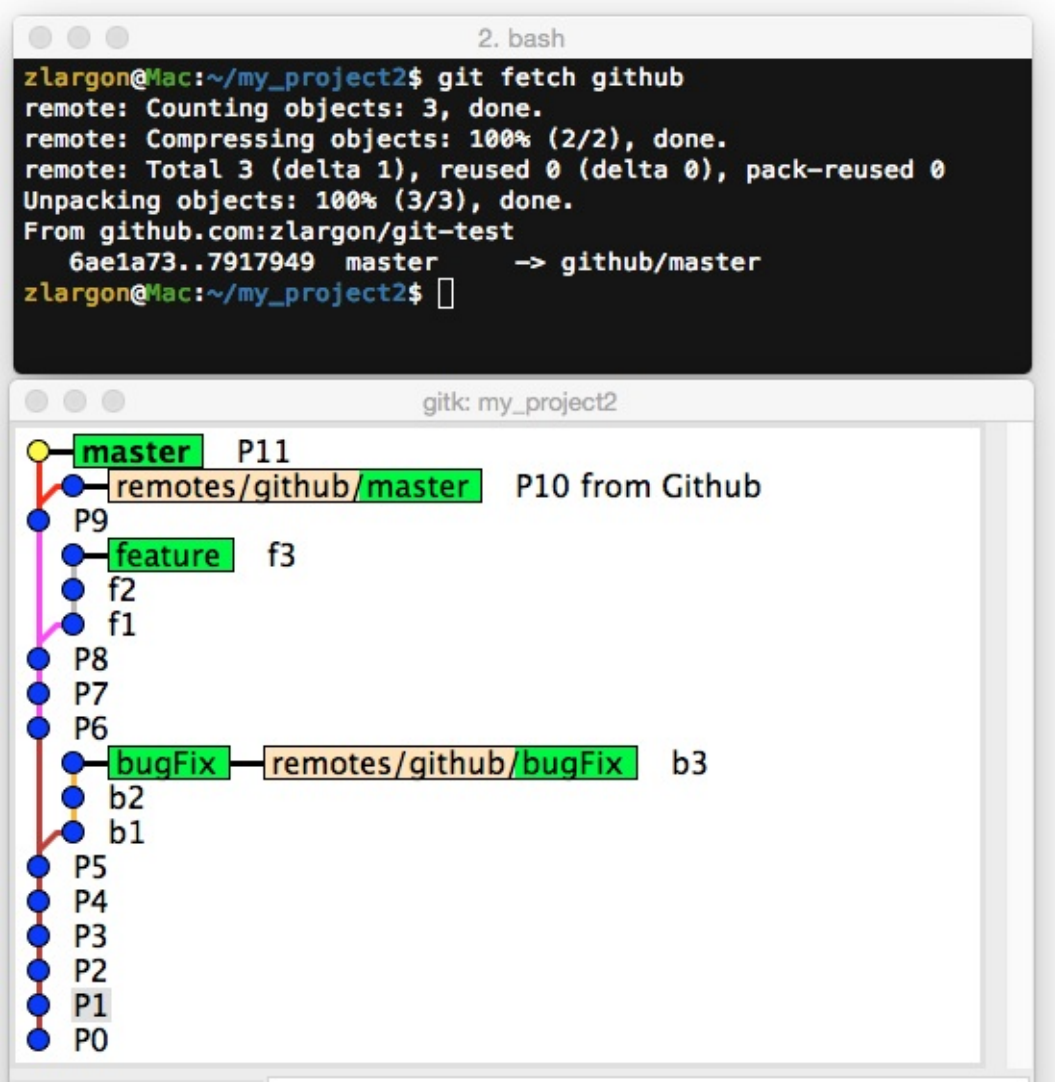
使用 `git fetch <remote name>` 更新 "指定" remote 底下的分支

我們之前有提到，`remotes/<remote name>/<branch name>` 會存放對應 remote branch

因此，首先我們要做的第一件事情，就是更新 remote branch

```
$ git fetch github
```

fetch 會去讀取 remote repo 的內容，並且更新 remote branch 的內容



從 `gitk` 可以看出 `remotes/github/master` 已經更新，但是 `master` 是在 P11 的位置

使用 `git fetch --all` 更新 "所有" remote 底下的分支

`git fetch <remote name>` 這個指令一次只能更新一個 remote

如果你的專案底下有多個 remote，而且全部都想要更新的話，就可以使用這個指令

`git fetch --all` 指令也等同於以下這個指令

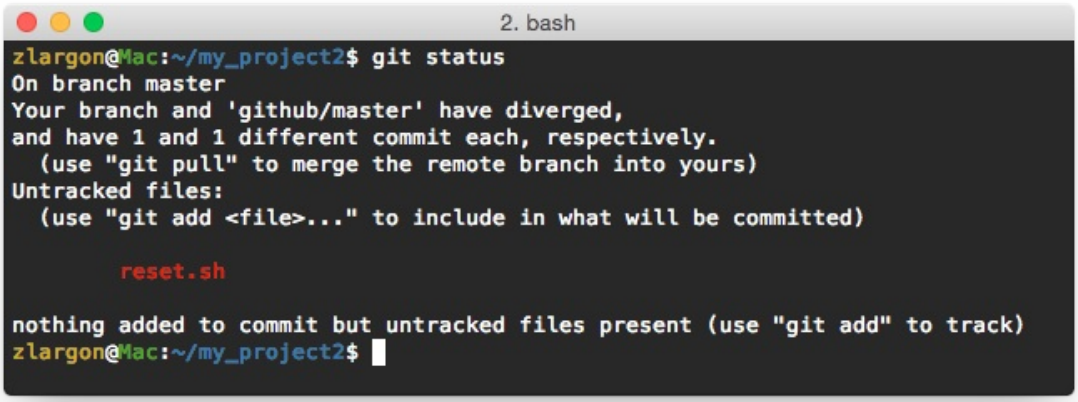
```
$ git remote update
```

使用 `git pull` 同步分支（Merge）

這時候，如果我們之前有設定 upstream 的話

`git status` 的追蹤訊息有提示我們可以用 `git pull` 來 merge 遠端的分支

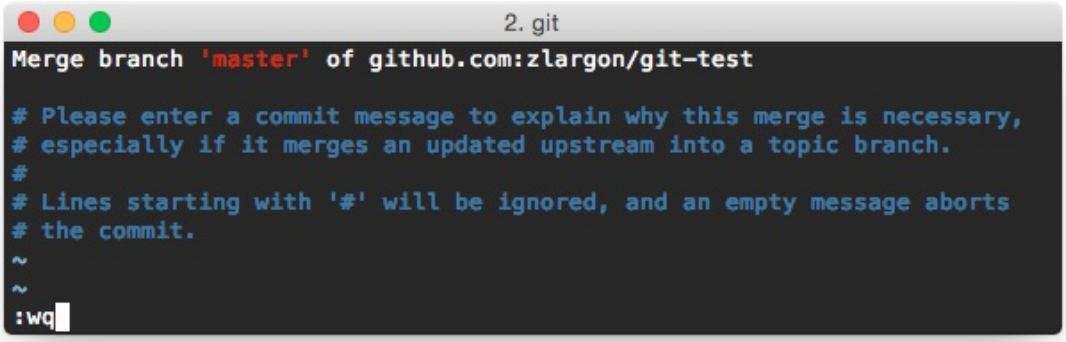
```
Your branch and 'github/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
```

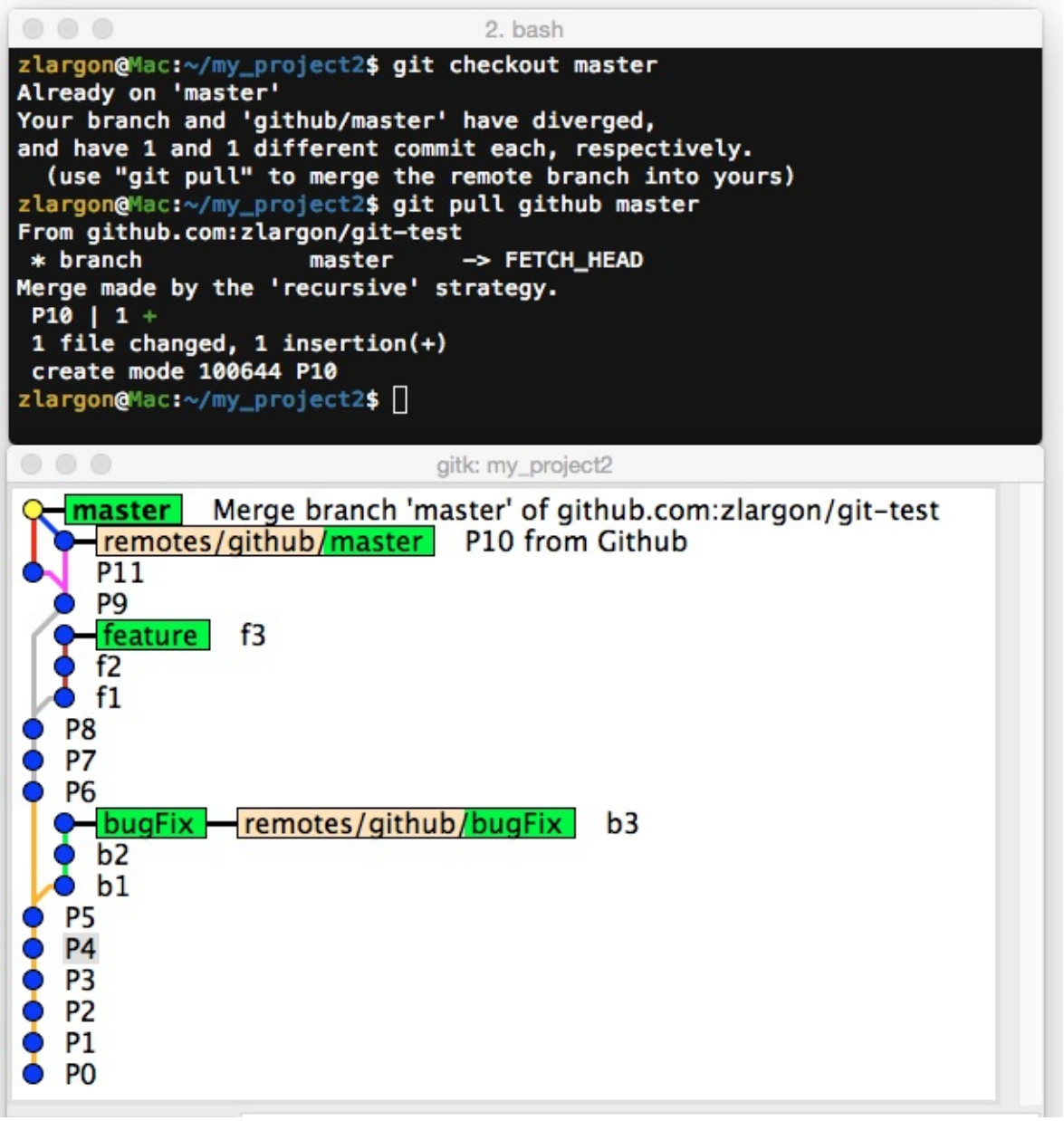



```
$ git checkout master
$ git pull              # 如果沒有設定 upstream，就一定要加 <remote name> 跟 <branch name>
```

由於 git pull 會幫我們做 `merge`，若沒有發生版本衝突，就會直接 Fast-Forward

但若發生了版本衝突的時候，就需要提交 Merge Patch





`git pull` 的作用等同於以下的指令

```
$ git fetch github
$ git merge remotes/github/master
```

其實在之前 "Merge 合併分支" 有討論過 merge 的缺點

這裡的 P10 跟 P11 其實改動很少，使用 merge 會產生多餘且沒意義的 Merge Patch

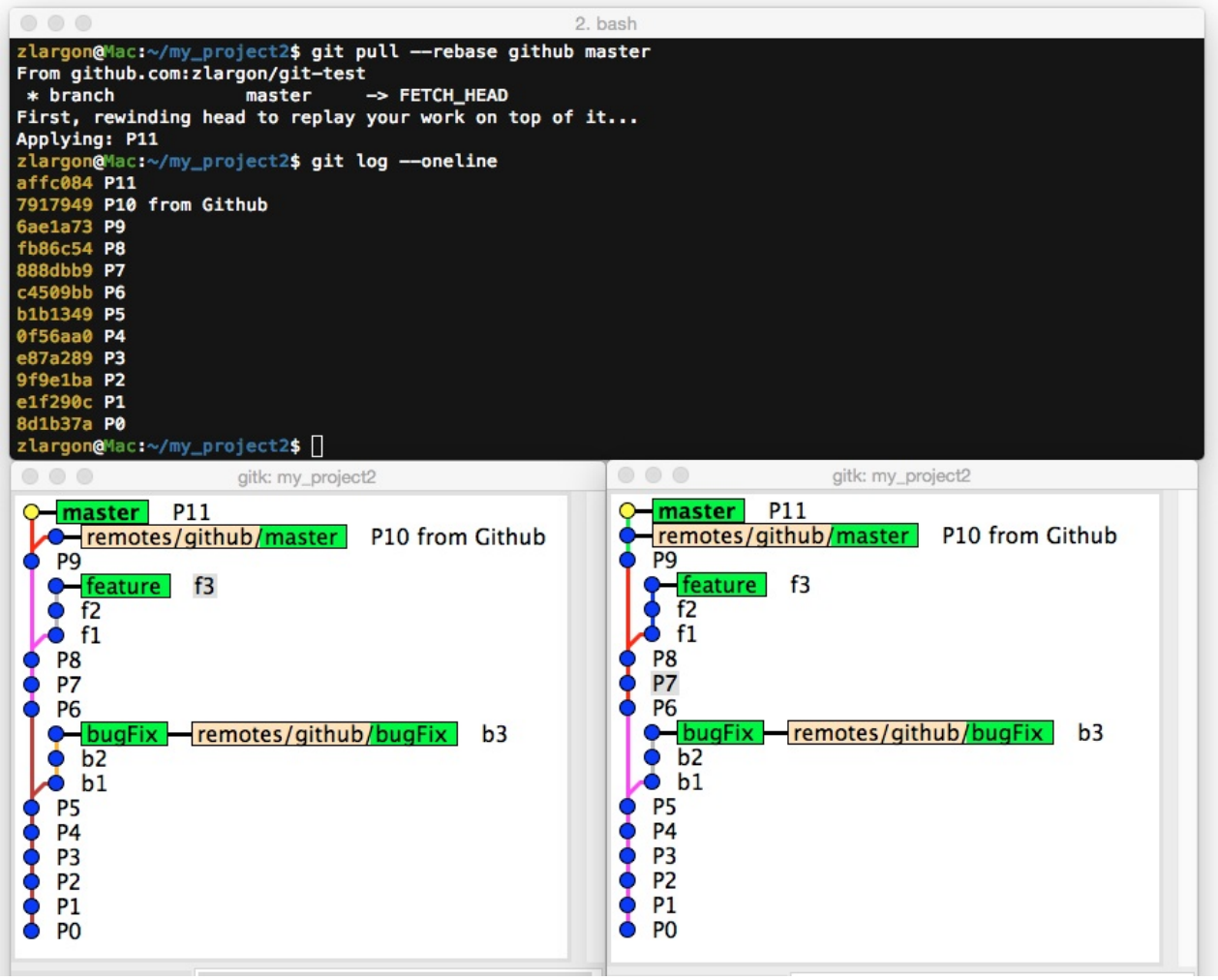
如果我們不希望他產生 Merge Patch，就要用 `rebase` 的方式來同步

使用 `git pull --rebase` 同步分支（Rebase）

當 `git pull` 遇到版本衝突的時候，預設會使用 `merge` 來解

加上 `--rebase` 參數的話，git 就會用 `rebase` 來解 conflict

```
$ git checkout master
$ git pull --rebase     # 如果沒有設定 upstream，就一定要加 <remote name> 跟 <branch name>
```



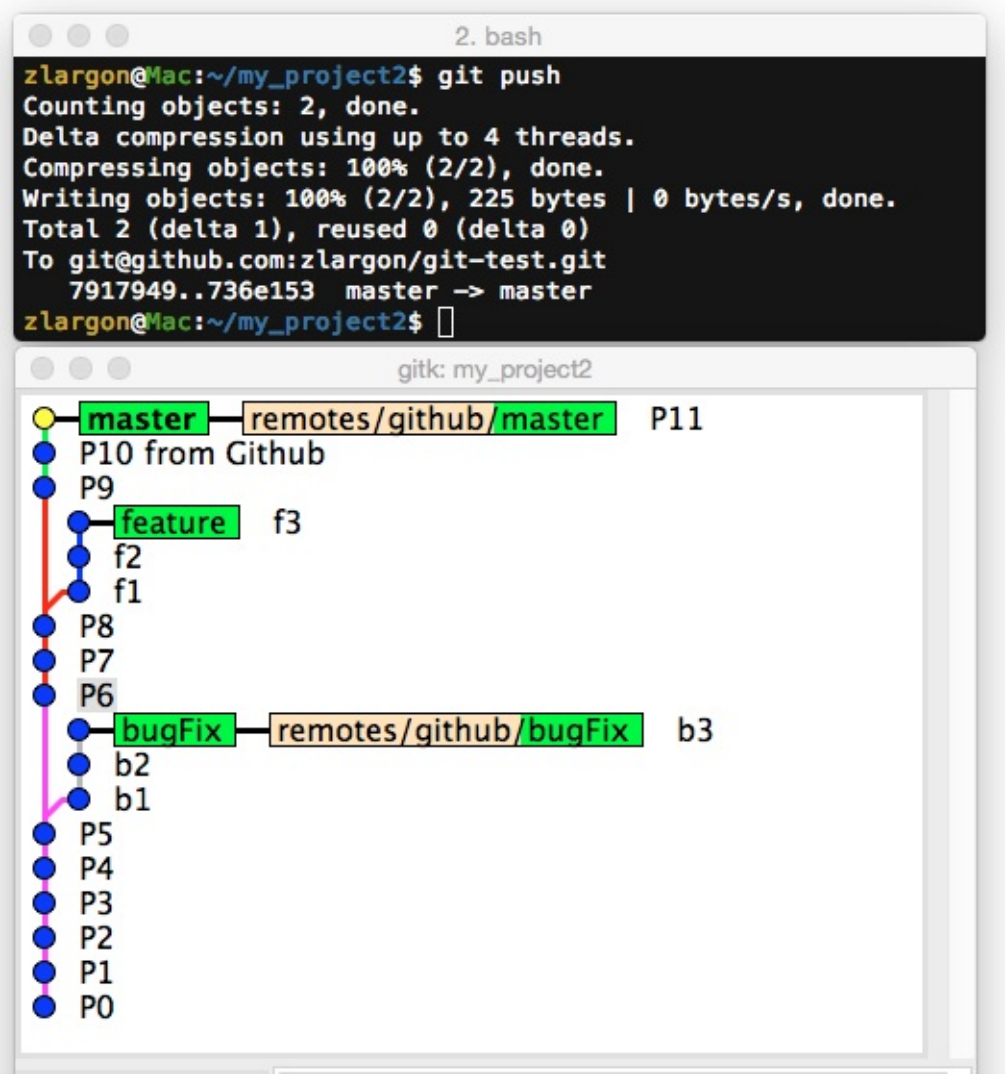
從 `gitk` 可以看出，他把 P11 重新 rebase 到 P10 後面

這樣分支就會看起來乾淨

`git pull --rebase` 的作用等同於以下的指令

```
$ git fetch github
$ git rebase remotes/github/master
```

在我們 `master` 分支同步成功之後，就可以把 P11 push 出去了



本章回顧

- 使用 `git fetch <remote name>` 更新 "指定" remote 底下的分支
- 使用 `git fetch --all` 更新 "所有" remote 底下的分支

  > 同 `git remote update`

- 使用 `git pull` 同步分支（Merge）

  > 若沒有設定 upstream，就一定要加 `<remote name>` 跟 `<branch name>`

- 使用 `git pull --rebase` 同步分支（Rebase）

  > 若沒有設定 upstream，就一定要加 `<remote name>` 跟 `<branch name>`

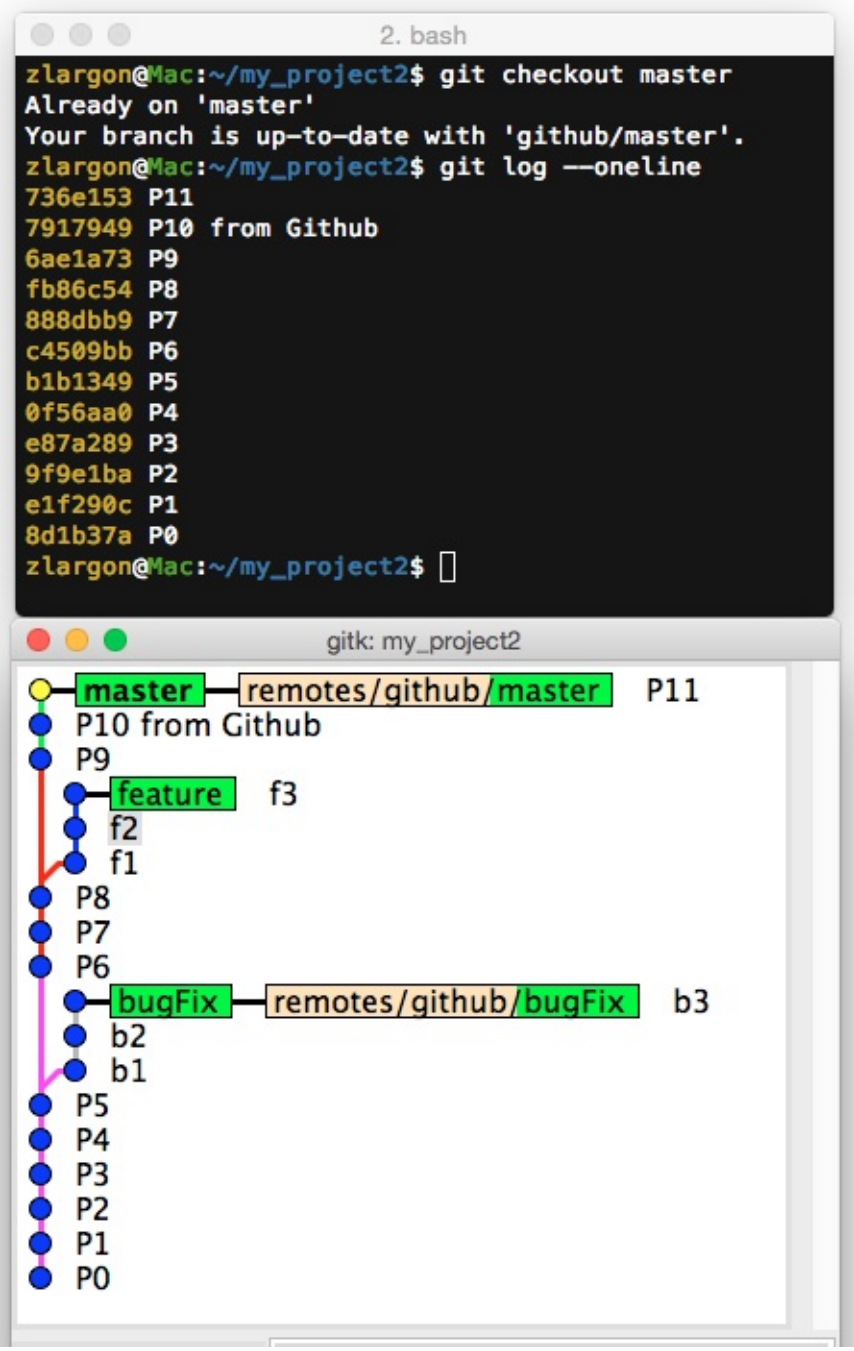
強制更新遠端分支

我們在 "修改 / 訂正 Patch" 有提到

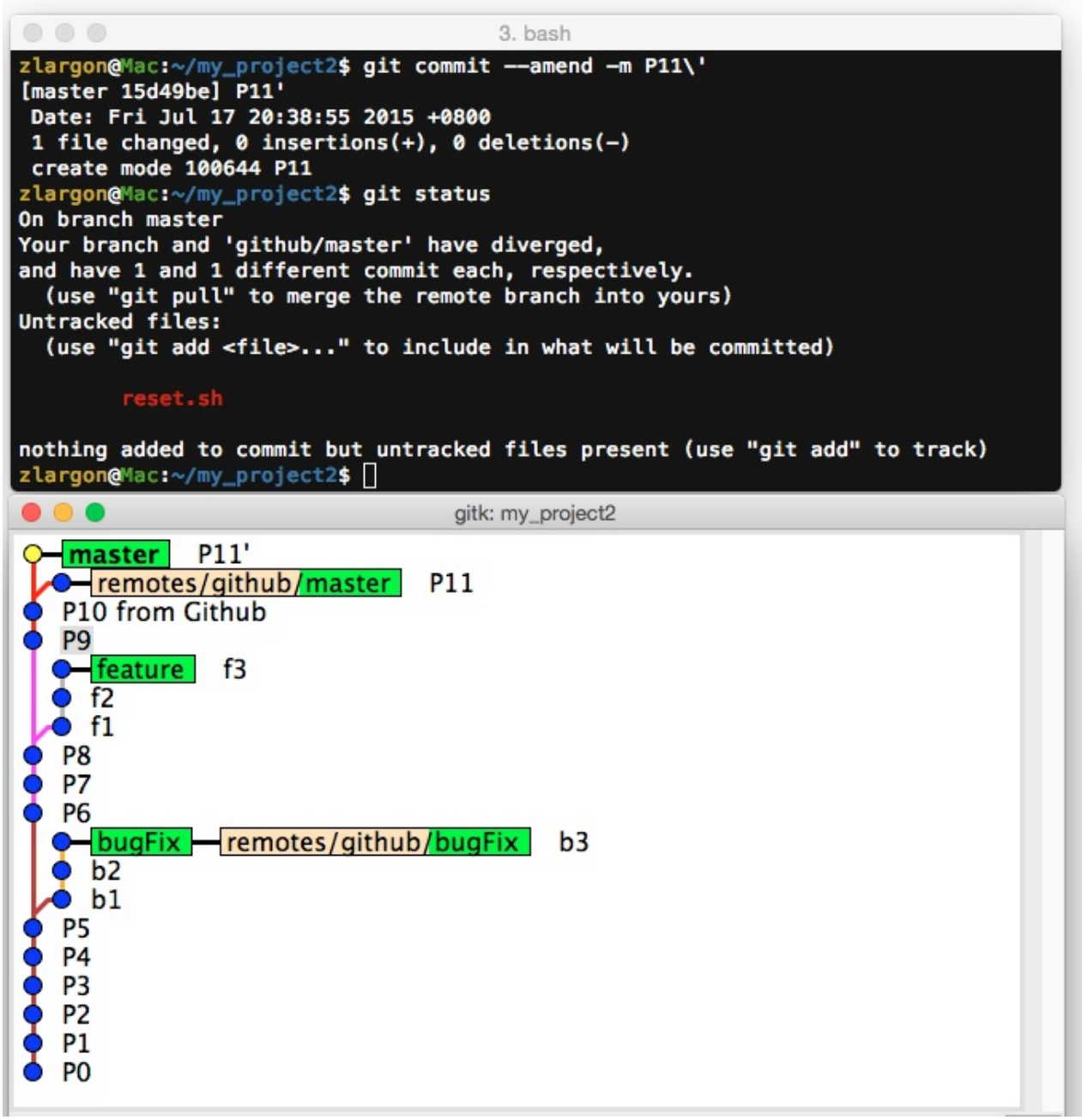
有時候我們在 local 端上 patch 會不小心打錯提交訊息，或是有一些小錯誤

我們不想要只爲了一個錯誤，而另外再上新的 patch，所以我們會用 `git commit --amend` 來訂正他

但是如果我們已經 `push` 到 server 上的時候，要怎麼修正呢？



我們現在把 `master` 的 patch P11，用 `git commit --amend` 修改提交訊息成 P11'

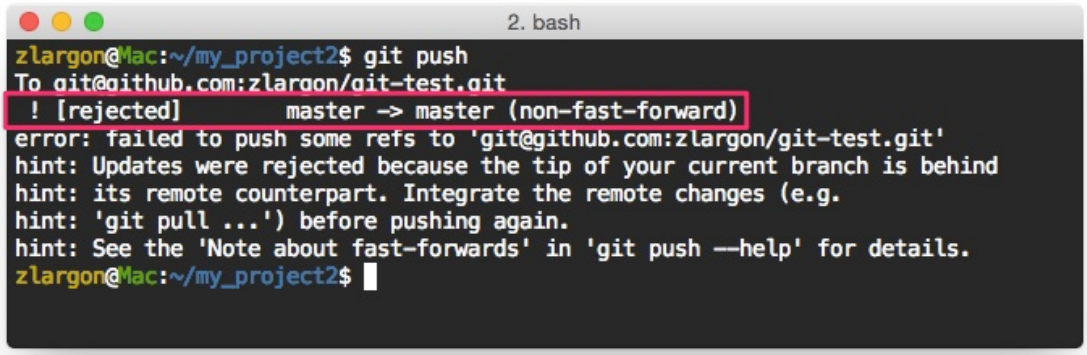


我們從 `gitk` 也可以看到，現在 `master (P11')` 分支跟 `remotes/github/master (P11)` 已經不同步了

剛剛我們有設定 upstream，因此 git 有提供我們一些追蹤的資訊，表示 `master` 與 `github/master` 已經產生了分歧

```
Your branch and 'github/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)
```

如果我們現在把 `master` 分支 push 出去，會被 server 拒絕



如果做 `git pull` 會跑出 Merge Patch

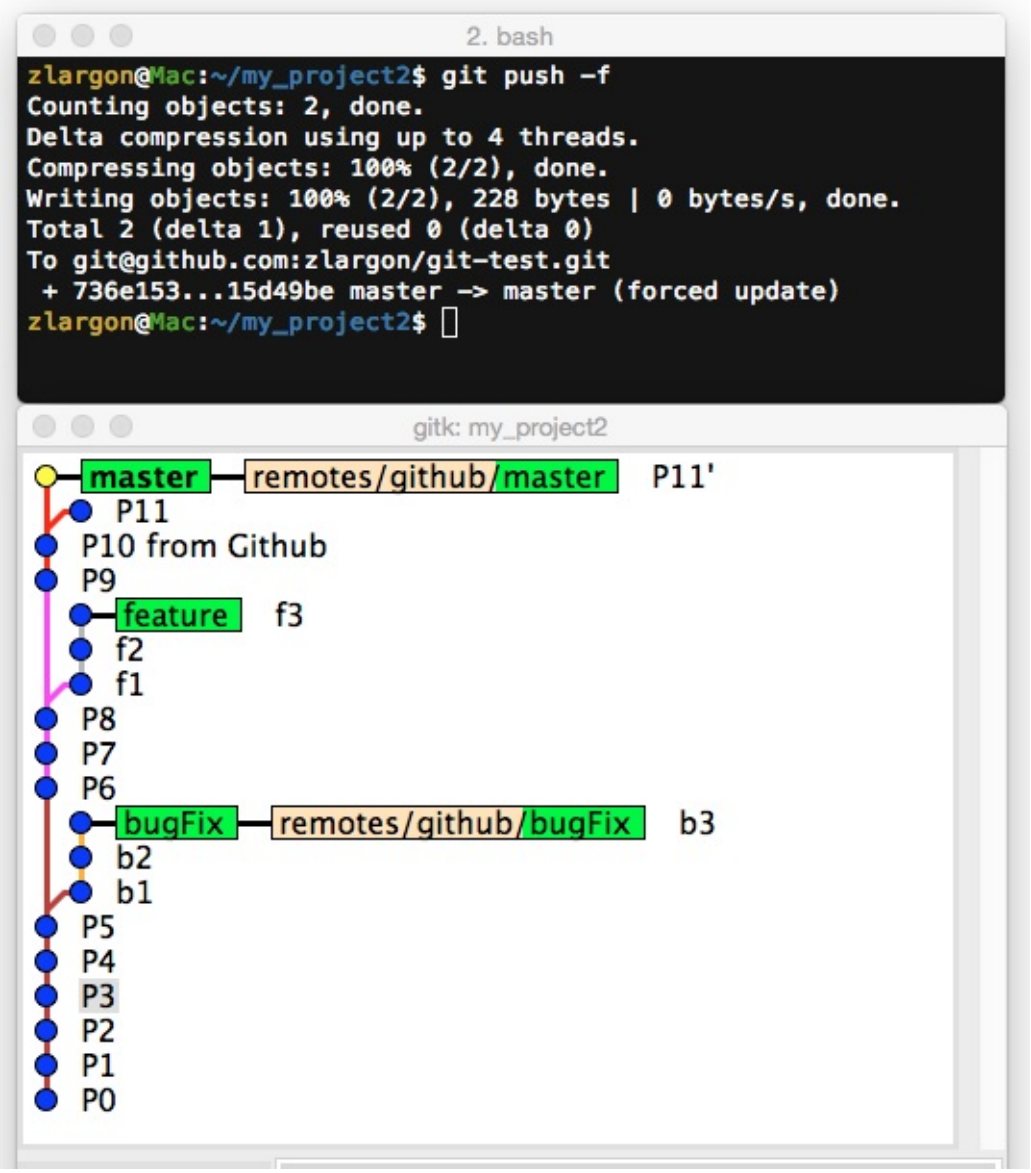
如果做 `git pull --rebase` 會把 P11' 移到 P11 之後

這些都不是我們想要的結果

使用 `git push -f` 強制更新遠端分支

參數 `-f` 等同於 `--force`，表示強制的意思

他可以強迫上傳，並且覆蓋掉遠端的分支



> 注意：
>
> 如果我們沒有設定 upstream 的話，後面還是要加 `<remote name>` 跟 `<branch name>`
>
> `$ git push -f <remote name> <branch name>`

注意事項

在是多人協同開發一個專案的時候， `git push -f` 是非常危險的指令

很可能會不小心把別人上傳的 code 整個覆蓋掉

通常只有在單人開發，或是非常有把握的時候，才會用 `git push -f` 來強制更新分支

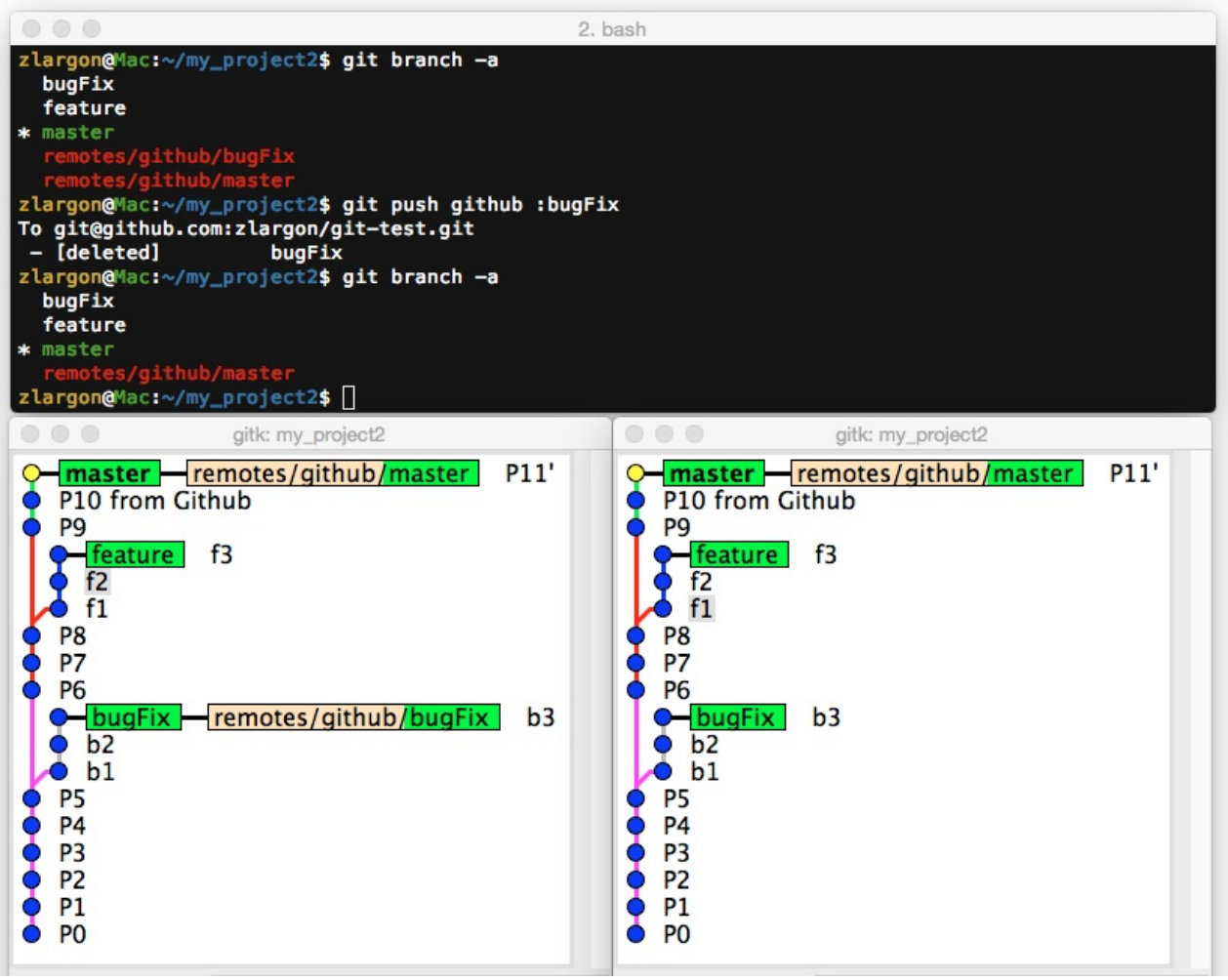
刪除遠端分支

在 "上傳分支" 教過如何使用 `git push` 添加遠端的分支

接下來我們想要把分支 `bugFix` 刪除

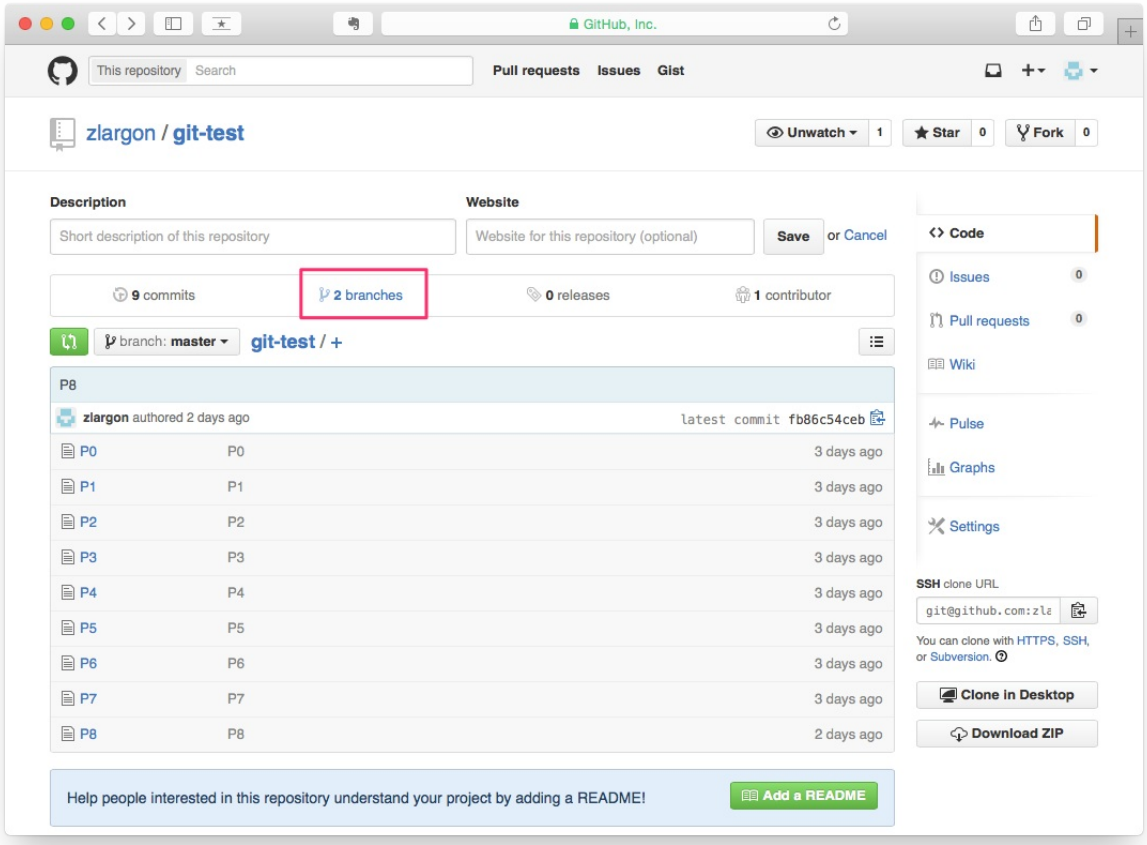
使用 `git push <remote name> :<branch name>` 刪除遠端分支

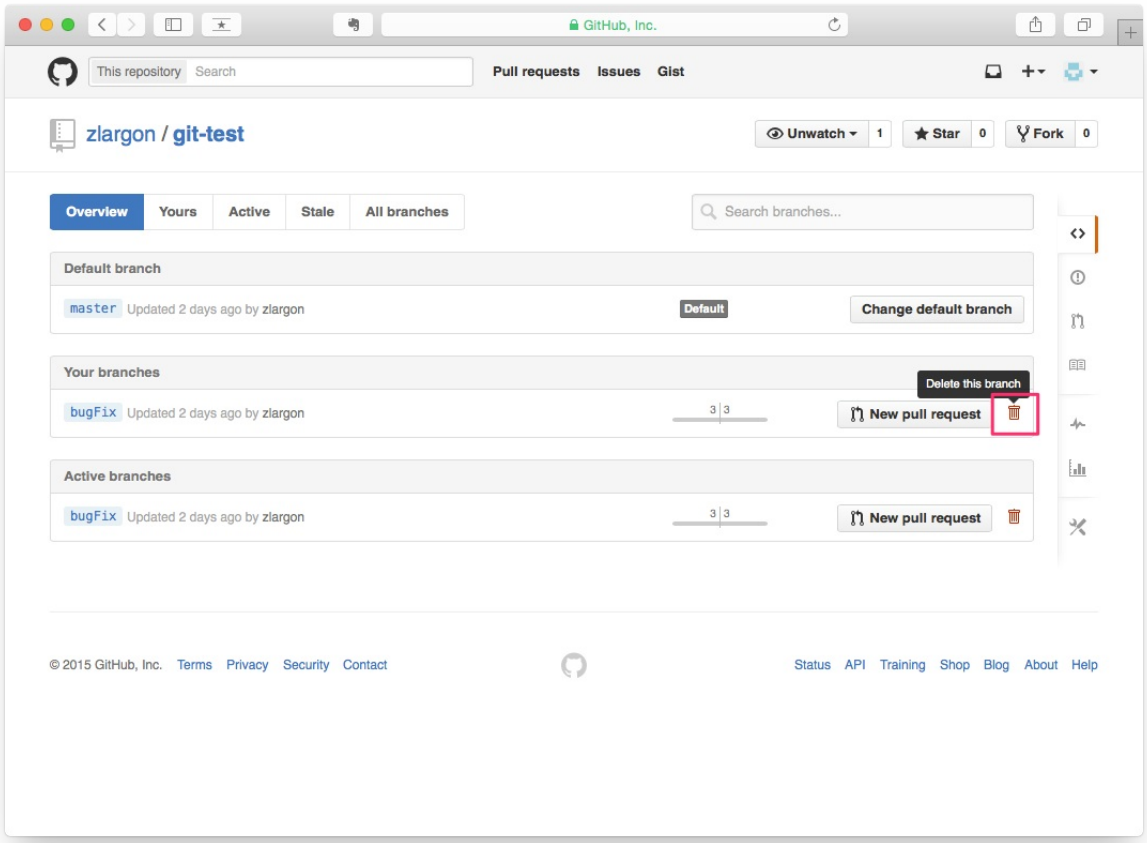
這裡一樣是用 `git push`，只要在分支名稱的前面多加一個 `:`（冒號）



使用 Github 網頁介面刪除分支

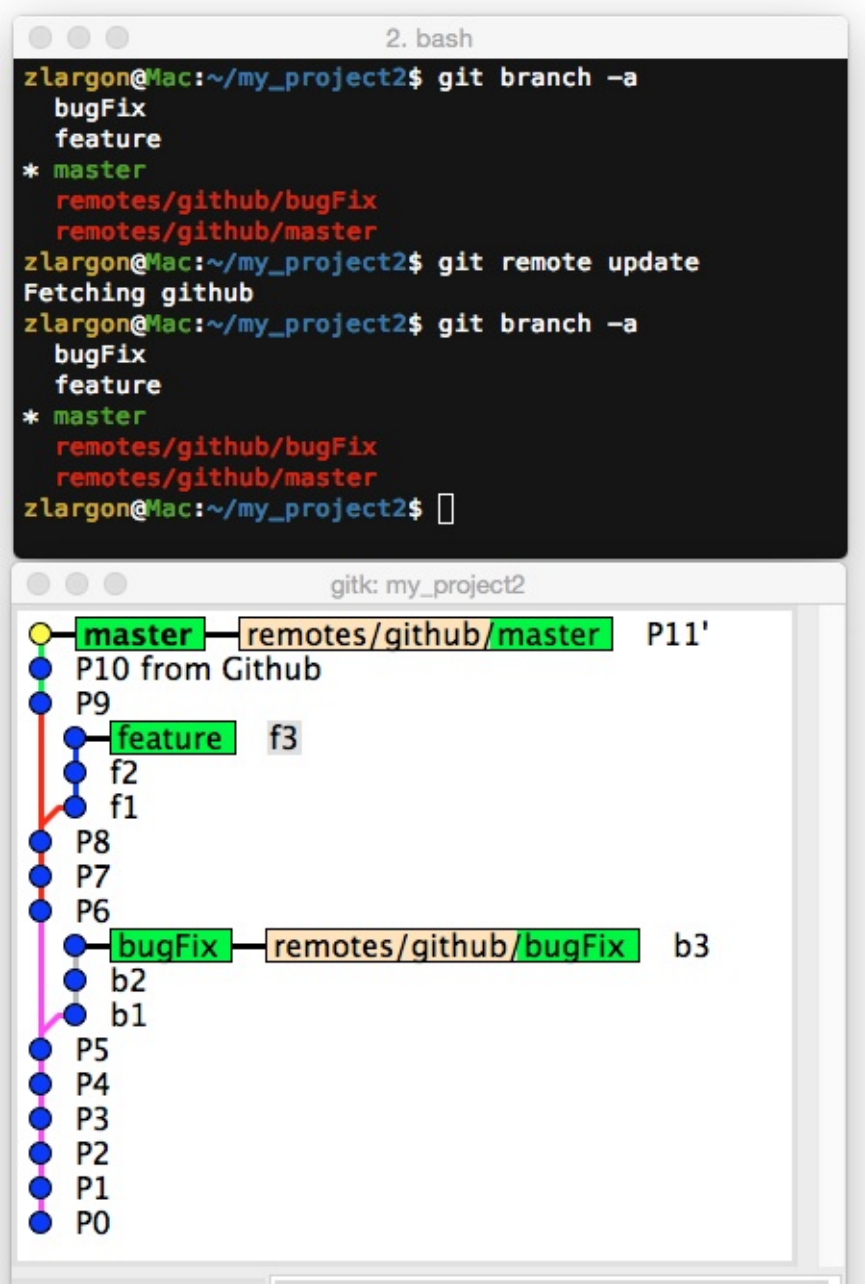
除此之外，也可以透過 Github 的網頁介面來操作



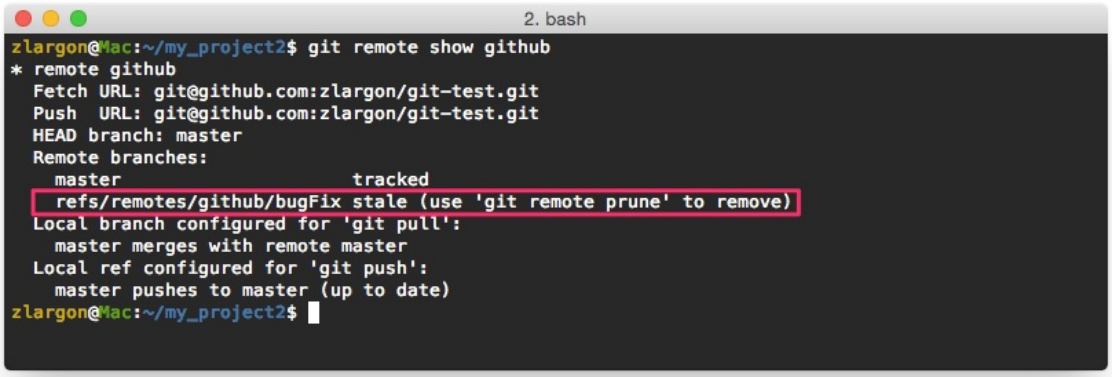


在我們刪除完 `bugFix` 分支之後，我們查看我們本機端的狀況，發現 **`remotes/github/bugFix`** 並沒有消失

即使我們用了 `git remote update` 之後，結果還是一樣，本機端的分支依舊無法跟遠端同步



使用 `git remote show <remote name>` 查看更多關於 **remote** 的資訊

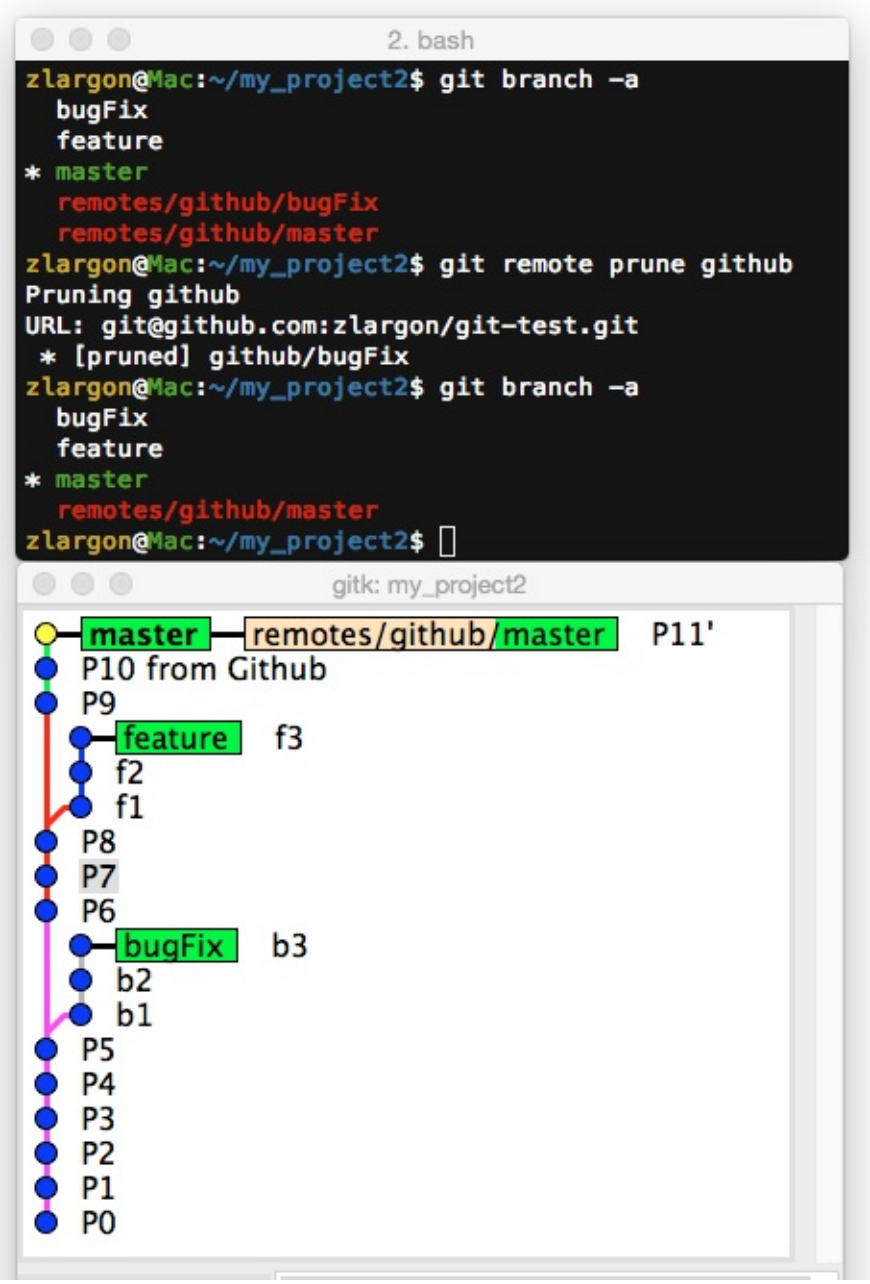



```
refs/remotes/github/bugFix stale (use 'git remote prune' to
remove)
```

這行的意思是說，遠端的 bugFix 已經過期了，請用 `git remote prune` 將它移除

使用 `git remote prune <remote name>` 刪除 remote 底下所有過時的分支

```
$ git remote prune github
```



這個指令也可以用 `git fetch` 來代換

```
$ git fetch -p
$ git fetch --prune     # 同上
```

使用 `git remote update -p` 同步遠端分支，並且移除過時的遠端分支

Git 可能是爲了避免遠端的分支被刪，所以才預設同步時保留過時的分支

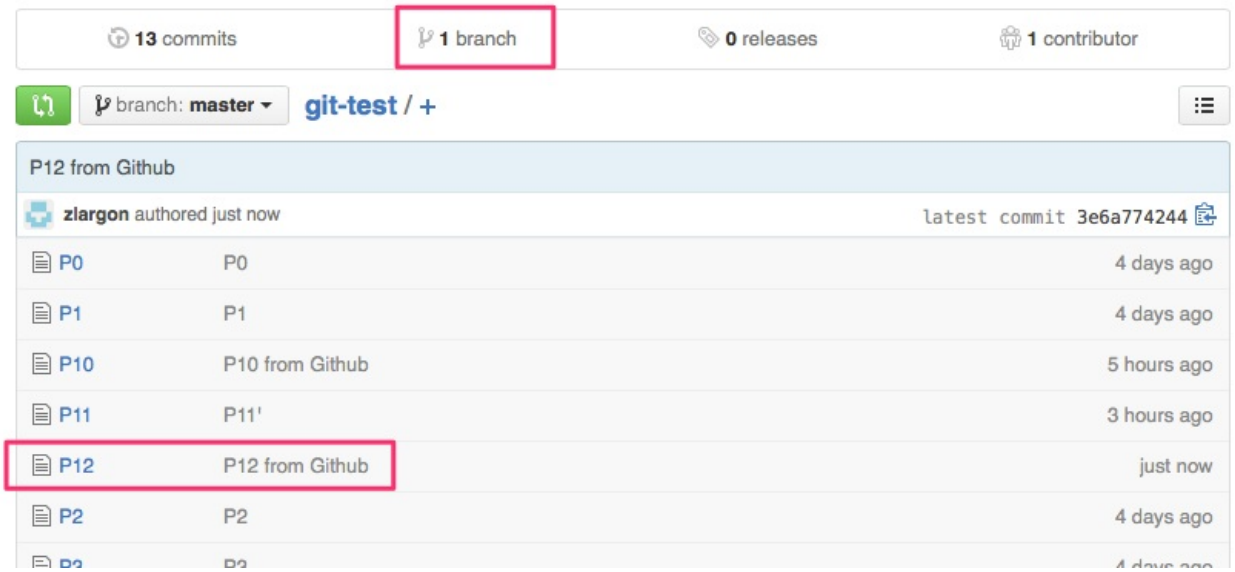
但是爲了要刪這個遠端的分支，要先用 `git remote show` 檢查，再用 `git remote prune` 刪除，這樣實在太麻煩了

`git remote update -p` 一次幫你全部搞定

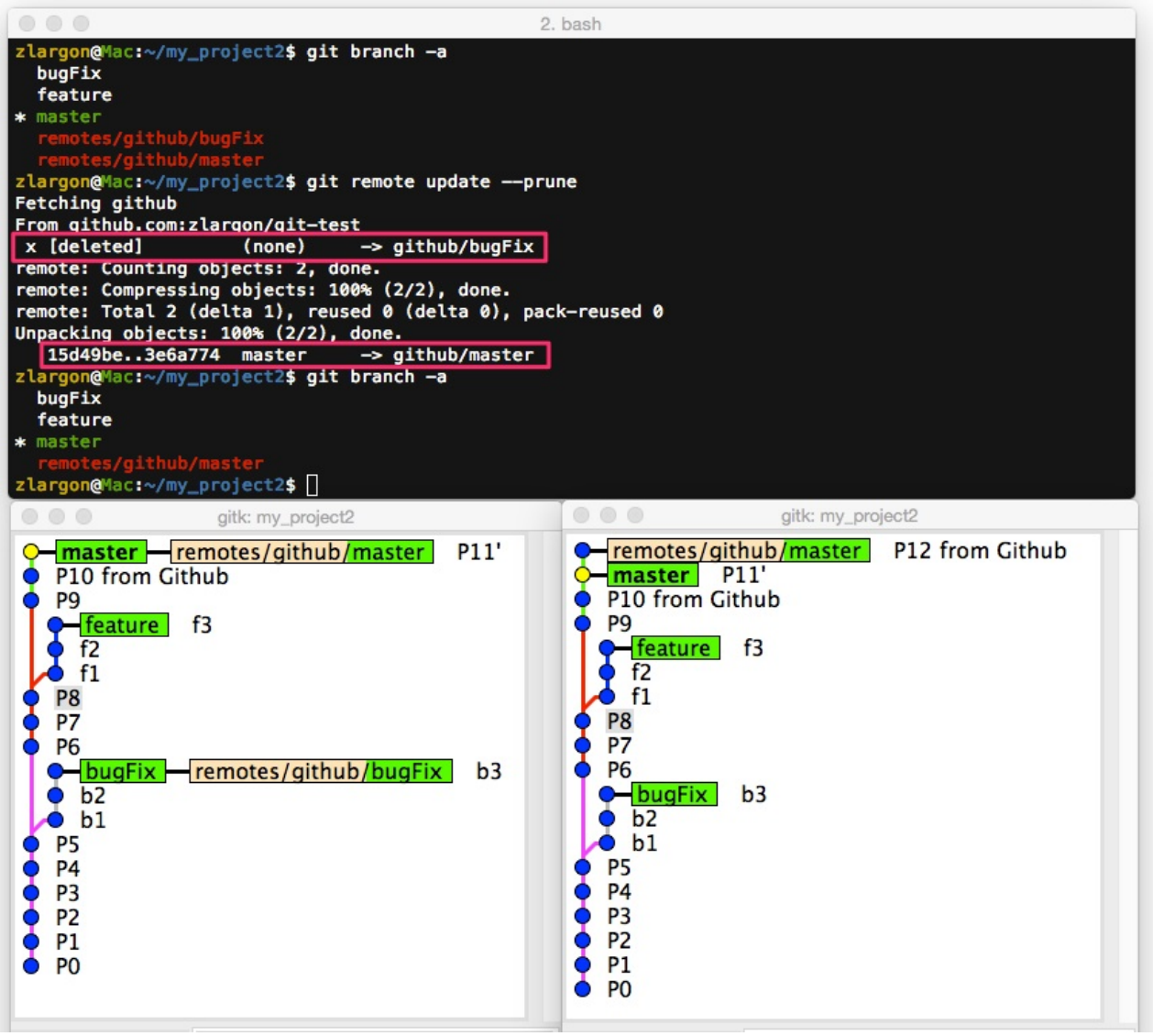
參數 `-p` 等同於 `--prune`

我們把分支 `bugFix` 再 push 回去 server

從 Github 的網頁介面新增一個 P12 的檔案，並且把分支 `bugFix` 刪除




```
$ git remote update -p     # 一次搞定
```



這個指令也可以用 `git fetch` 來代換

```
$ git fetch --all -p
$ git fetch --all --prune     # 同上
```

本章回顧

- 使用 `git push <remote name> :<branch name>` 刪除遠端分支
- 使用 `git remote show <remote name>` 查看更多關於 remote 的資訊
- 使用 `git remote prune <remote name>` 刪除 remote 底下所有過時的分支

  > 可用 `git fetch -p` 代換

- 使用 `git remote update -p` 同步遠端分支，並且移除過時的遠端分支

  > 同 `git fetch --all -p`

進階篇

檔案暫存

```
git stash
git stash list
git stash pop
git stash pop stash@{n}
git stash drop
git stash drop stash@{n}
git stash clear
```

Add / Checkout 檔案部分內容

```
git add -p
git checkout -p
```

版本標籤

```
git tag
git tag <tag name> <commit id>
git tag -a <tag name> <commit id>
git tag -a <tag name> <commit id> -m <msg1> -m <msg2>
git tag -a <tag name> <tag name>^{} -f
git tag -d <tag name>

git push <remote name> <tag name>
git push <remote name> --tags
git push <remote name> :<tag name>
git push -f <remote name> <tag name>
```

子模組

```
git submodule init
git submodule update
git submodule add <repo url> <project path>
git clone --recursive <repo URL>
```

檔案暫存

有時候我們開發到一半的時候，突然有同事跟你說他剛剛上一個 patch，想請你幫忙驗看看

或是你開發新功能到一半的時候，老闆突然跟你說，有發現一個 bug 要立即解決

或是你突然有一個 idea，想要做一個簡單的測試

這時候我們可以將開發到一半的檔案，全部備份到另外一個分支

```
$ git checkout -b backup            # 新增 backup 分支，並且切換過去
$ git add -A
$ git commit -m "this is backup"    # 提交所有檔案進來
$ git checkout master               # 回到 master 分支做其他事情
```

這時候在 `master` 做其他工作，完成後用再把 `backup` 分支挑回來 `master` 繼續開發

```
$ git cherry-pick backup            # 把 backup 的 patch 挑回 master
$ git reset HEAD^                   # 讓 backup 的所儲存的內容，全部回到檔案提交前的狀態
$ git branch -D backup              # 刪除 backup 分支
```

不過如果檔案改動沒有很大，但是還是要這樣開新 branch 感覺有點麻煩

Git 有提供一個方法可以快速暫存檔案，然後再叫回來

使用 `git stash` 暫存被修改的檔案

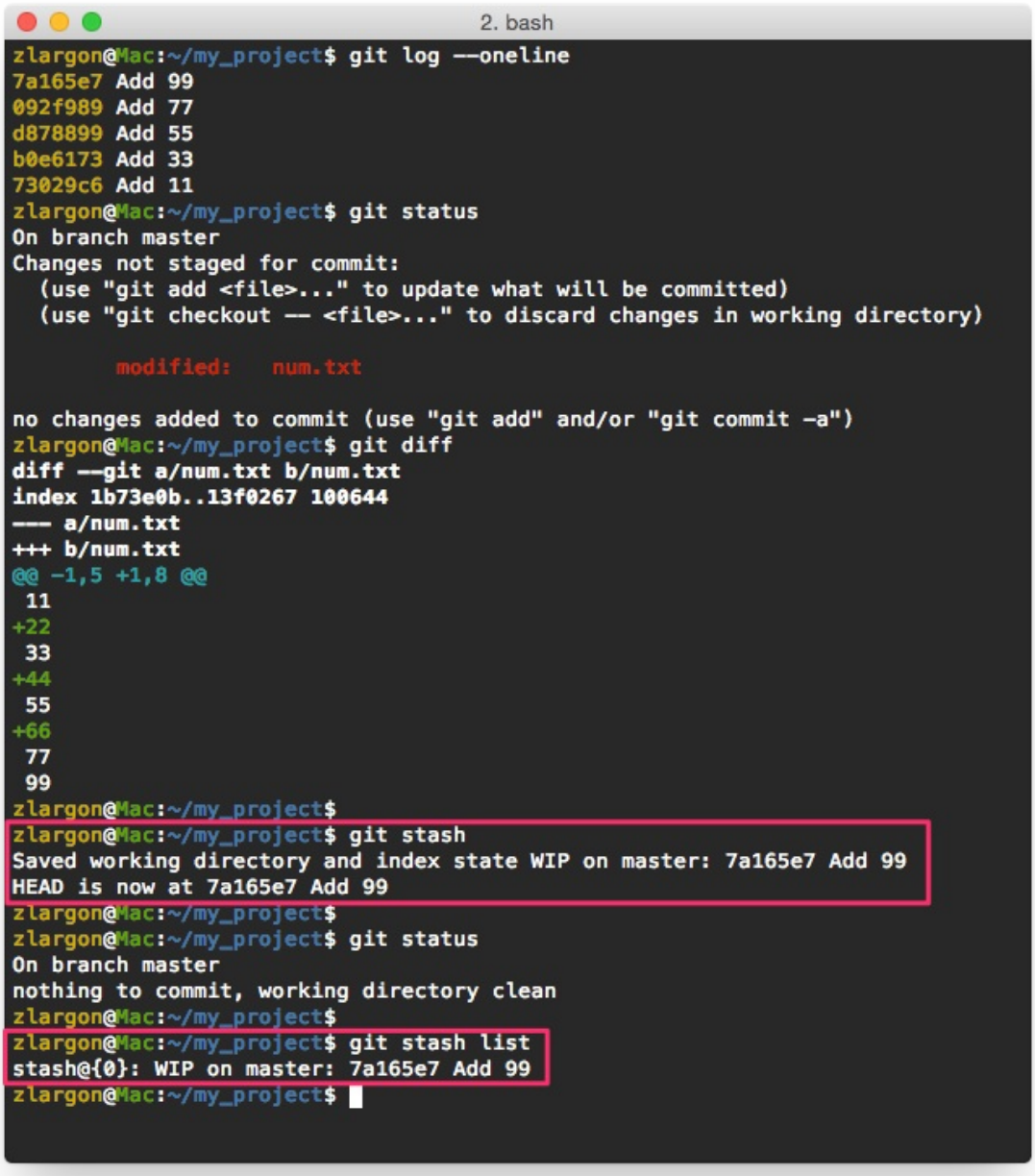
num.txt

```
11
33
55
77
99
```

這是原本 `num.txt` 的檔案內容，接下來我們新增 22, 44, 66

然後我們用 `git stash` 把他暫存起來

```
$ git stash             # 暫存目前被修改的檔案
$ git stash list        # 列出所有 stash 紀錄
```




```
Saved working directory and index state WIP on master: 7a165e7
Add 99
HEAD is now at 7a165e7 Add 99
```

他表示這些是在 `7a165e7` 開發到一半，被 stash 起來的

這裡的 `WIP` 的意思是 `Working In Progress`

而 `HEAD` 也被移到 patch `7a165e7`，也就是 `master` 的最後一個 patch

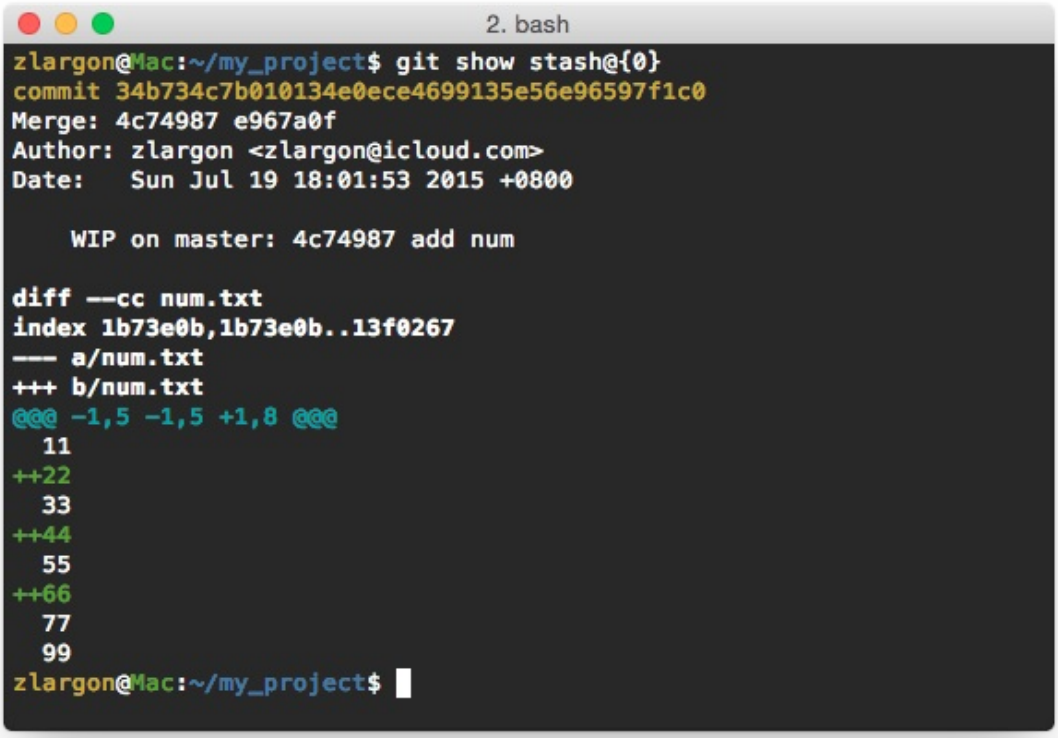
使用 `git stash list` 來查看所有被我們 stash 的內容

```
$ git stash list
```

Short Name	Description
`stash@{0}`	WIP on master: 7a165e7 Add 99

我們也可以用 `git show <short name>` 來查看他的改動內容

```
$ git show stash@{0}
```

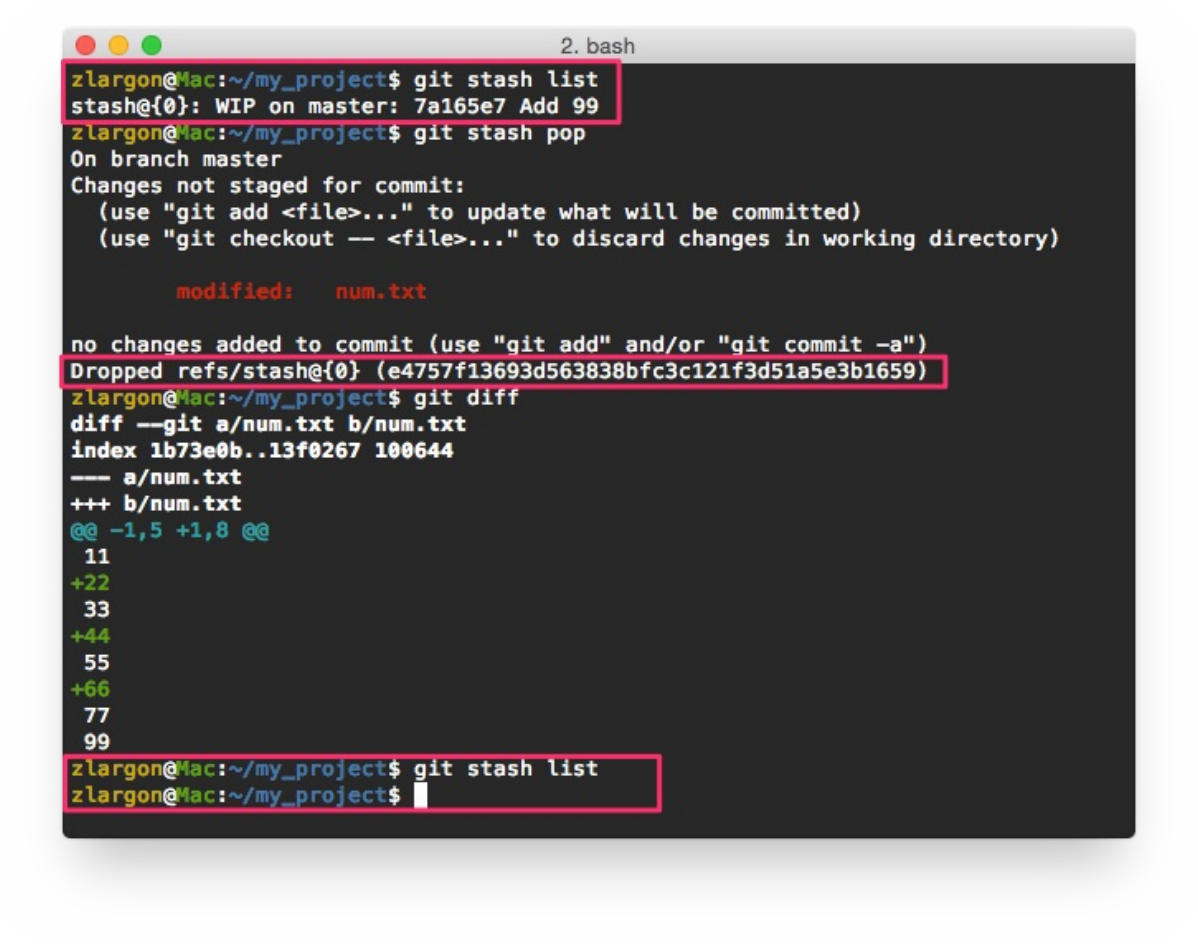


使用 `git stash pop` 把暫存的檔案叫回來

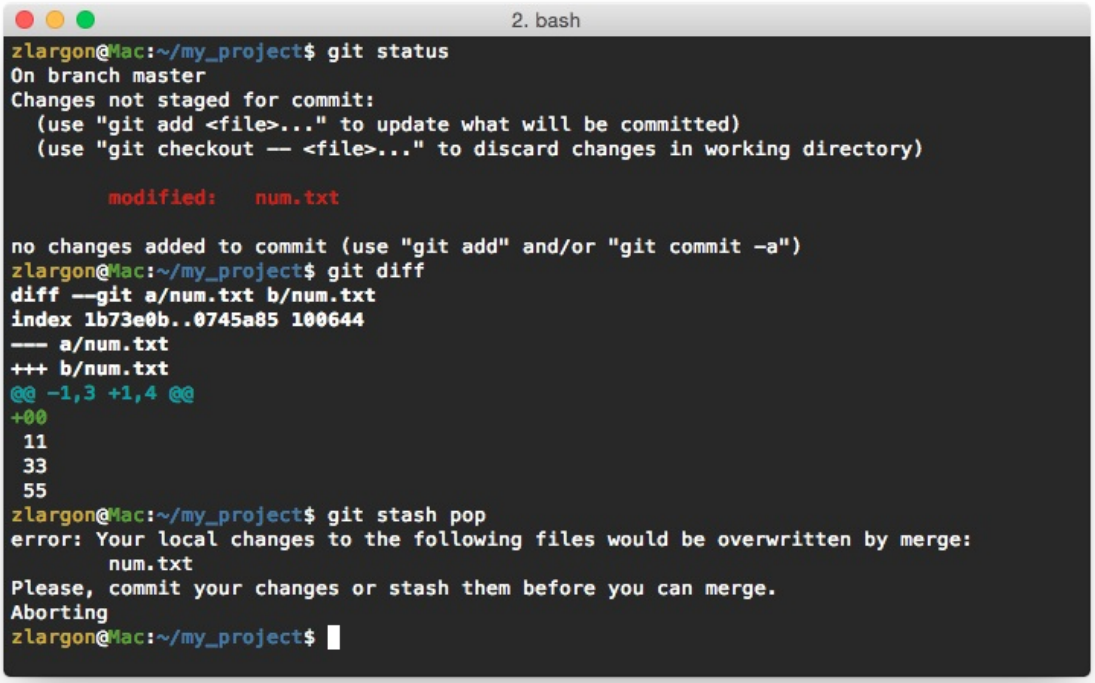
使用 `git stash pop` 把檔案叫回來的同時，也被從 stash list 被移除

可以在後面加上 short name，來指定要 pop 出來的 stash 編號

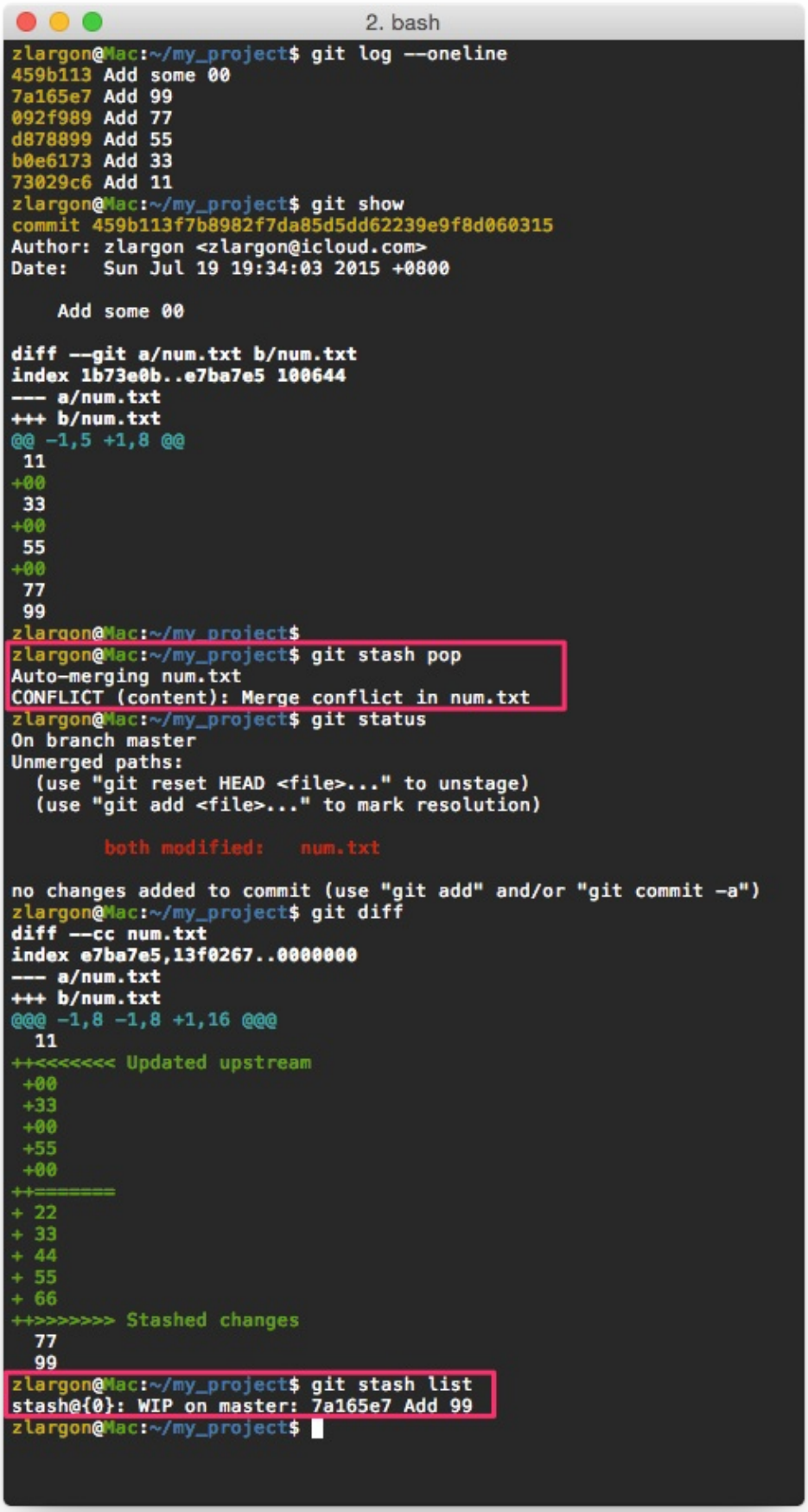
```
$ git stash pop                 # pop 第一個 stash（stash@{0}）
$ git stash pop stash@{n}       # 指定要 pop 出來的 stash 編號
```



使用 `git stash pop` 之前，要先把 `git status` 清空



注意，如果後來又提交了新的 patch，再做 `stash pop` 有可能會發生 ***conflict***

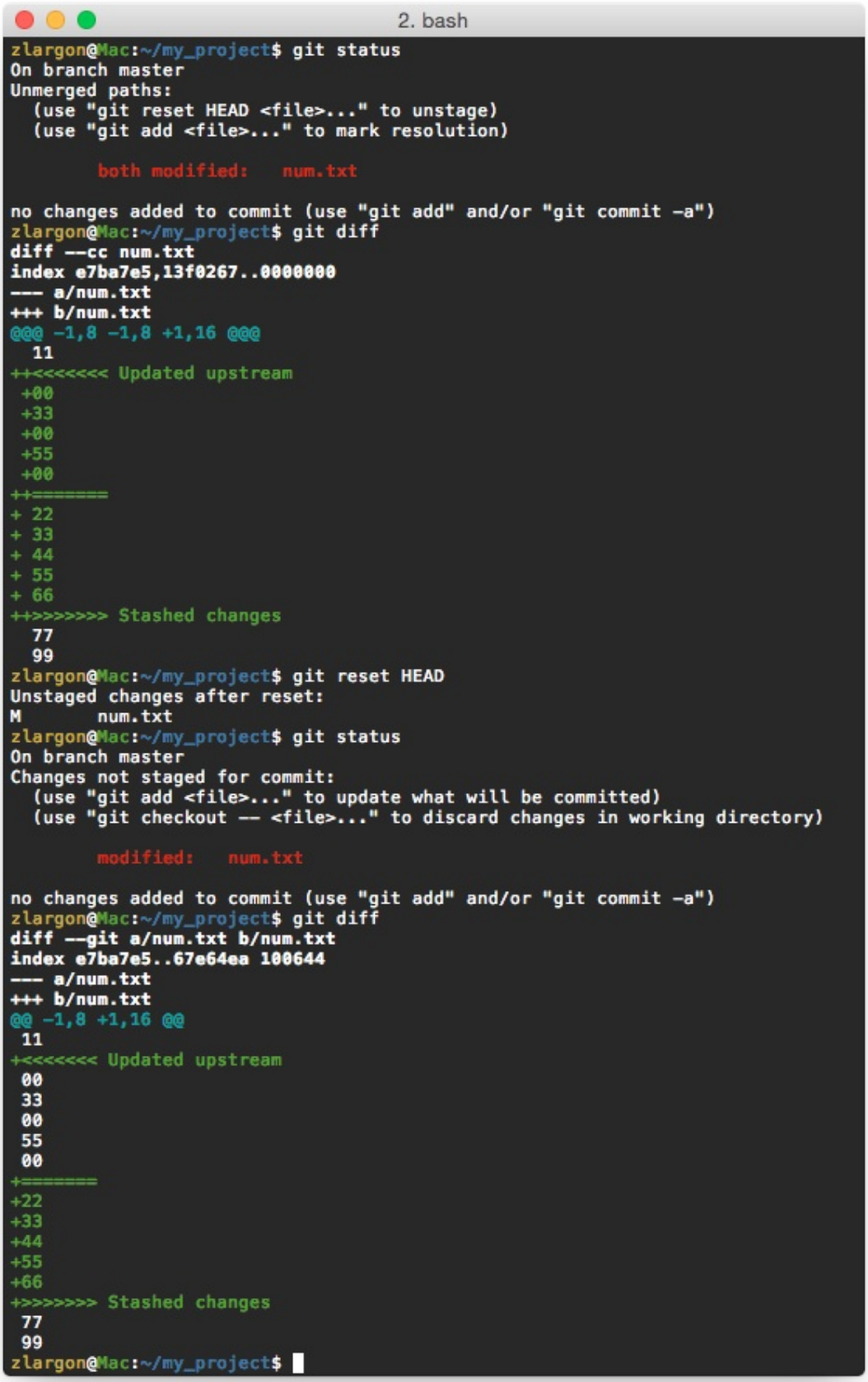

2. bash
zlargon@Mac:~/my_project$ git log --oneline
459b113 Add some 00
7a165e7 Add 99
092f989 Add 77
d878899 Add 55
b0e6173 Add 33
73029c6 Add 11
zlargon@Mac:~/my_project$ git show
commit 459b113f7b8982f7da85d5dd62239e9f8d060315
Author: zlargon <zlargon@icloud.com>
Date: Sun Jul 19 19:34:03 2015 +0800
Add some 00
diff --git a/num.txt b/num.txt
index 1b73e0b..e7ba7e5 100644
--- a/num.txt
+++ b/num.txt
@@ -1,5 +1,8 @@
11
+00
33
+00
55
+00
77
99
zlargon@Mac:~/my_project$
zlargon@Mac:~/my_project$ git stash pop
Auto-merging num.txt
CONFLICT (content): Merge conflict in num.txt
zlargon@Mac:~/my_project$ git status
On branch master
Unmerged paths:
(use "git reset HEAD <file>..." to unstage)
(use "git add <file>..." to mark resolution)
both modified: num.txt
no changes added to commit (use "git add" and/or "git commit -a")
zlargon@Mac:~/my_project$ git diff
diff --cc num.txt
index e7ba7e5,13f0267..0000000
--- a/num.txt
+++ b/num.txt
@@@ -1,8 -1,8 +1,16 @@@
11
++<<<<<<< Updated upstream
+00
+33
+00
+55
+00
++=======
+ 22
+ 33
+ 44
+ 55
+ 66
++>>>>>>> Stashed changes
77
99
zlargon@Mac:~/my_project$ git stash list
stash@{0}: WIP on master: 7a165e7 Add 99
zlargon@Mac:~/my_project$

發生 ***conflict*** 的時候，stash 不會被清掉

所以可以 `git reset --hard HEAD` 還原之後，再重做一次 `git stash pop`

`Update Stream` 是你後來提交的部分，`Stashed changes` 就是 stash 所修改的部分

可以先用 `git reset HEAD` 取消檔案 `both modified` 的狀態再來解

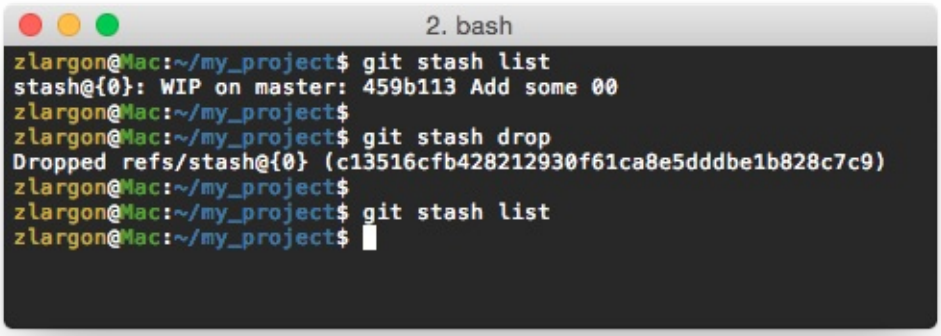

2. bash
zlargon@Mac:~/my_project$ git status
On branch master
Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both modified: num.txt
no changes added to commit (use "git add" and/or "git commit -a")
zlargon@Mac:~/my_project$ git diff
diff --cc num.txt
index e7ba7e5,13f0267..0000000
--- a/num.txt
+++ b/num.txt
@@@ -1,8 -1,8 +1,16 @@@
  11
++<<<<<<< Updated upstream
 +00
 +33
 +00
 +55
 +00
++=======
+ 22
+ 33
+ 44
+ 55
+ 66
++>>>>>>> Stashed changes
  77
  99
zlargon@Mac:~/my_project$ git reset HEAD
Unstaged changes after reset:
M   num.txt
zlargon@Mac:~/my_project$ 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: num.txt
no changes added to commit (use "git add" and/or "git commit -a")
zlargon@Mac:~/my_project$ git diff
diff --git a/num.txt b/num.txt
index e7ba7e5..67e64ea 100644
--- a/num.txt
+++ b/num.txt
@@ -1,8 +1,16 @@
 11
+<<<<<<< Updated upstream
 00
 33
 00
 55
 00
+=======
+22
+33
+44
+55
+66
+>>>>>>> Stashed changes
 77
 99
zlargon@Mac:~/my_project$

使用 `git stash drop` 丟棄暫存的檔案

`git stash drop` 會丟棄 stash list 裡面的第一個 stash，也就是 stash@{0}

可以在後面加上 short name，指定要丟地的 stash 編號

```
$ git stash drop                # 丟棄第一個 stash（stash@{0}）
$ git stash drop stash@{n}      # 丟棄編號 n 的 stash
```



使用 `git stash clear` 清空 stash list

```
$ git stash clear
```

注意事項

`git stash` 雖然很方便，但是千萬不要過度使用

因爲如果到時候 stash list 檔案太多的話，會整個亂掉，很難找出哪一個才是我們要的 stash

如果是重要的功能的話，建議還是另外開一個分支會比較安全

stash 比較適合輕量、檔案改動不大、暫時的狀況

平時盡量保持 stash list 爲清空的狀態

本章回顧

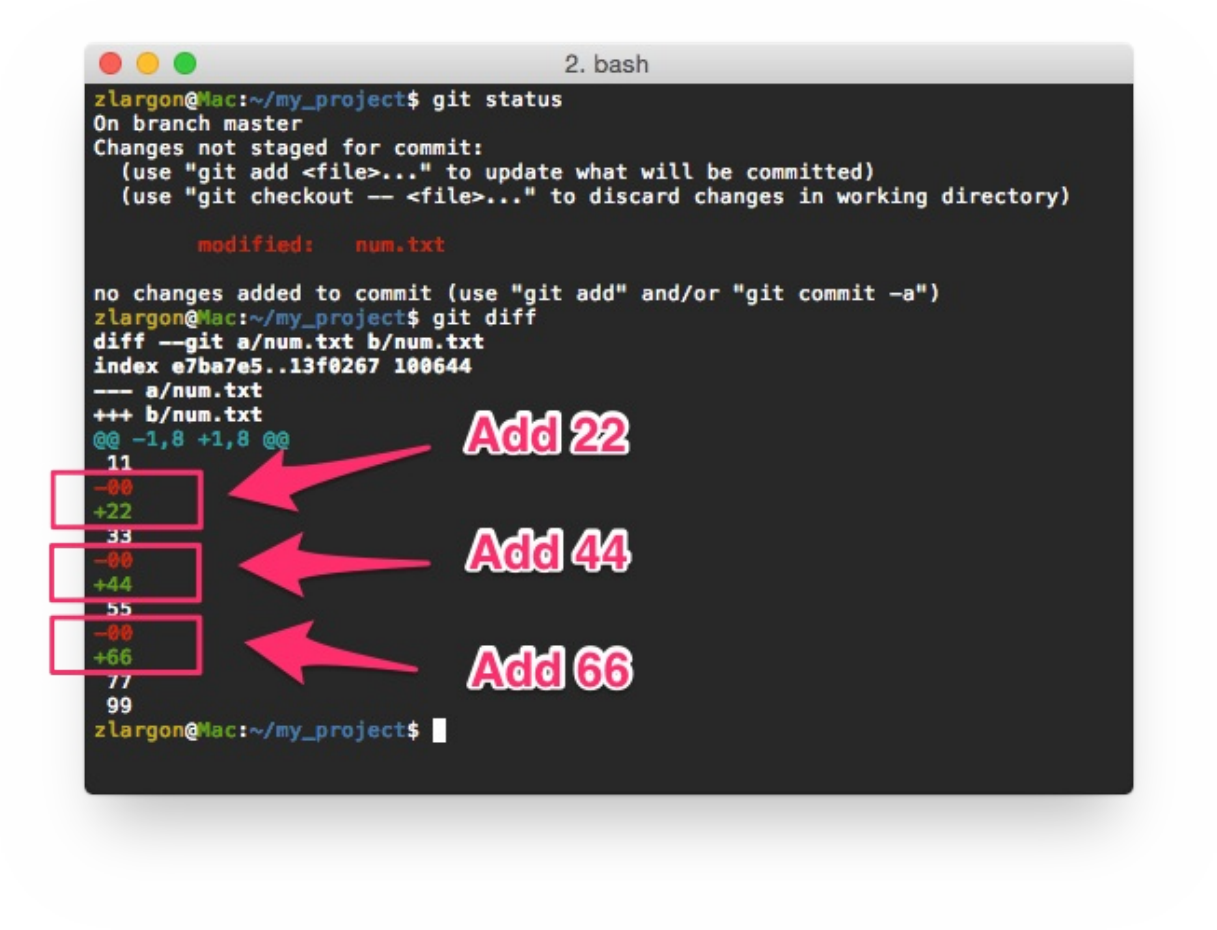
- 使用 `git stash` 暫存被修改的檔案
- 使用 `git stash list` 來查看所有被我們 stash 的內容
- 使用 `git stash pop` 把暫存的檔案叫回來
- 使用 `git stash drop` 丟棄暫存的檔案
- 使用 `git stash clear` 一次清空 stash list 裡面所有的 stash

Add / Checkout 檔案部分內容

我們知道可以用 `git add <file>` 來新增整個檔案，但是如果我們只想要新增檔案部分的內容的話

就可以加上 `-p` 的參數，同 `--patch` 來新增部分的內容

我們的目標是，把這次的改動分成三次來提交，分別是 "Add 22", "Add 44" 跟 "Add 66"



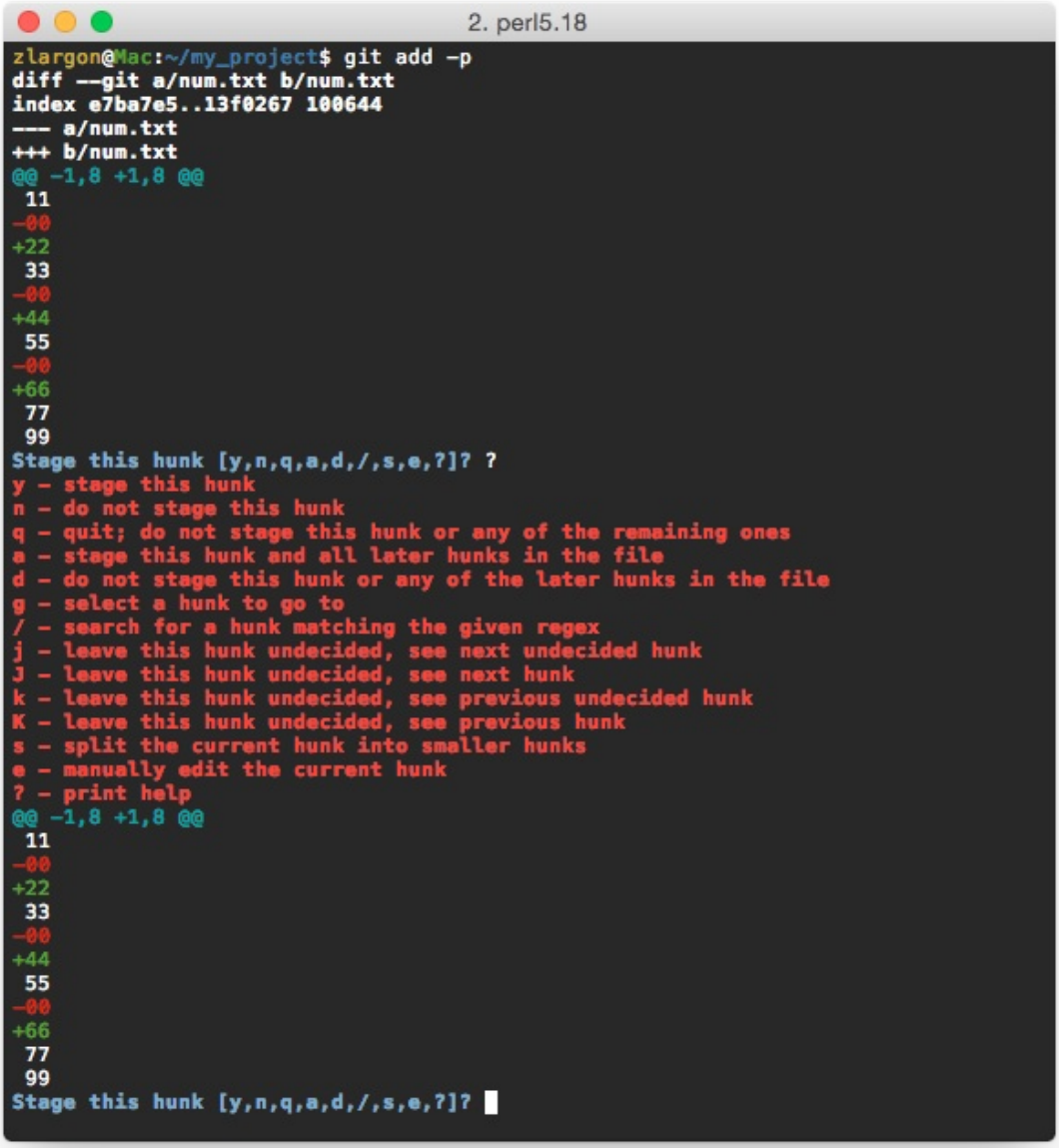
使用 `git add -p` 提交檔案部分的內容

```
$ git add -p
$ git add --patch   # 同上
```

Git 一次會出現一個區塊（hunk），問你要做什麼動作，並且列出選項

```
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

輸入 `?` 或是其他不在列子裡面的選項，就會顯示說明

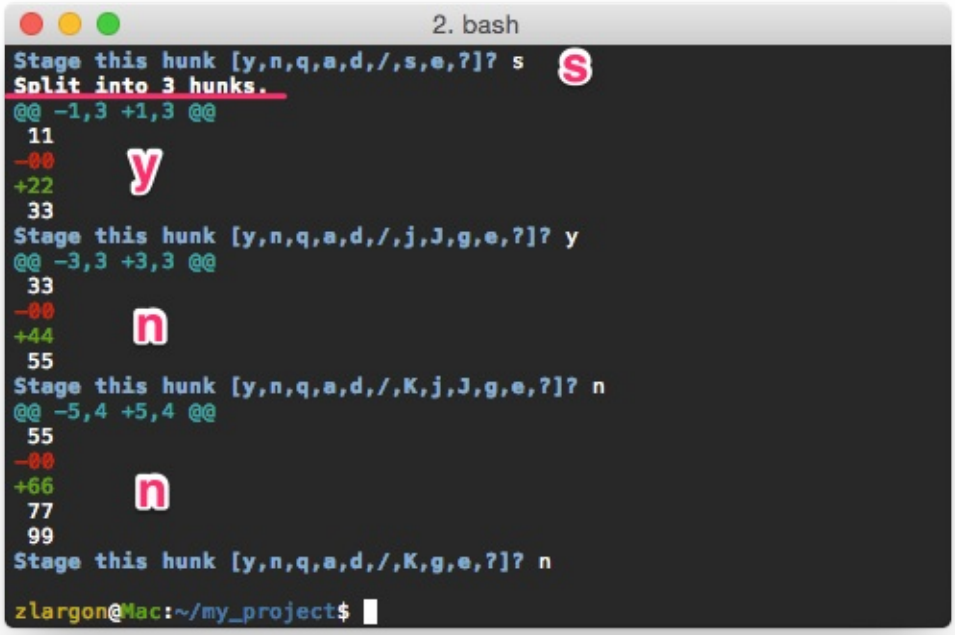

2. perl5.18
zlargon@Mac:~/my_project$ git add -p
diff --git a/num.txt b/num.txt
index e7ba7e5..13f0267 100644
--- a/num.txt
+++ b/num.txt
@@ -1,8 +1,8 @@
11
-00
+22
33
-00
+44
55
-00
+66
77
99
Stage this hunk [y,n,q,a,d,/,s,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
@@ -1,8 +1,8 @@
11
-00
+22
33
-00
+44
55
-00
+66
77
99
Stage this hunk [y,n,q,a,d,/,s,e,?]?

```
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

通常只會用到 y, n, s

- y 就是 yes，要 add 這個 hunk
- n 就是 no，不要 add 這個 hunk
- s 把目前的 hunk 再切成更小的 hunk

由於我們這次個改動的位置都太近了，所以用 s 把他再切成更小的 hunk



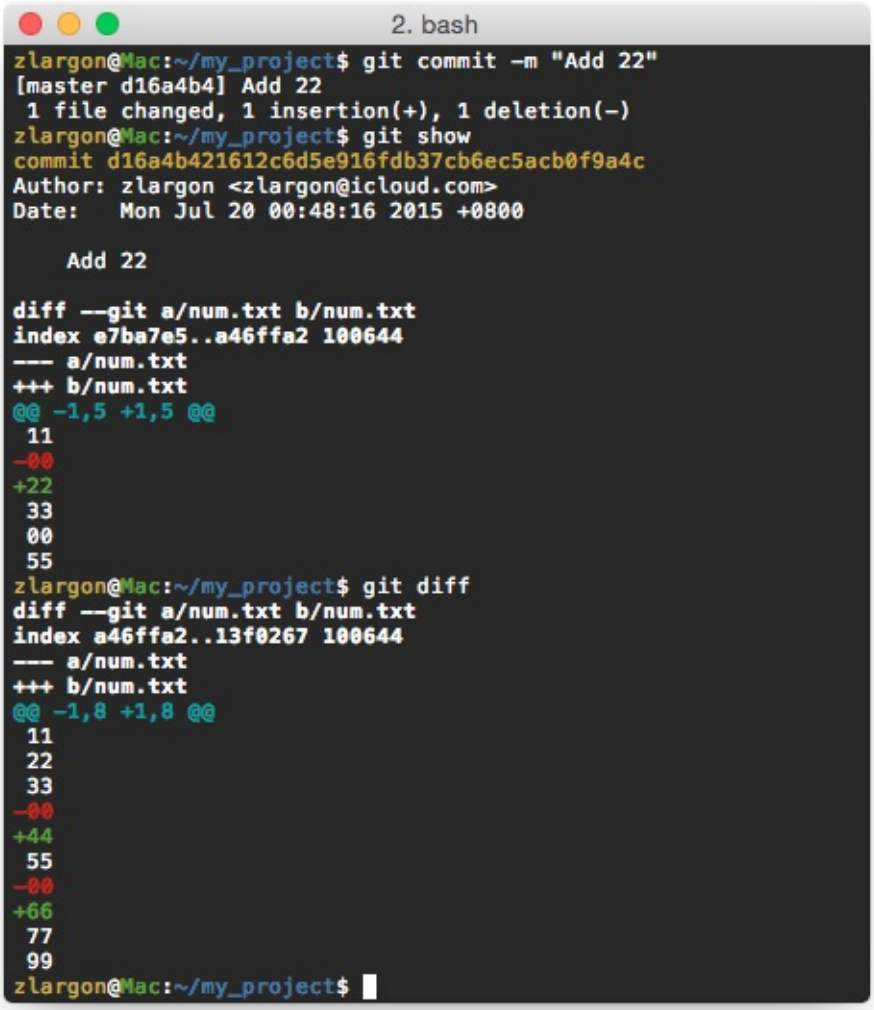
他把這次的 hunk 再切成三個小 hunk

第一部分我們要 22，輸入 y

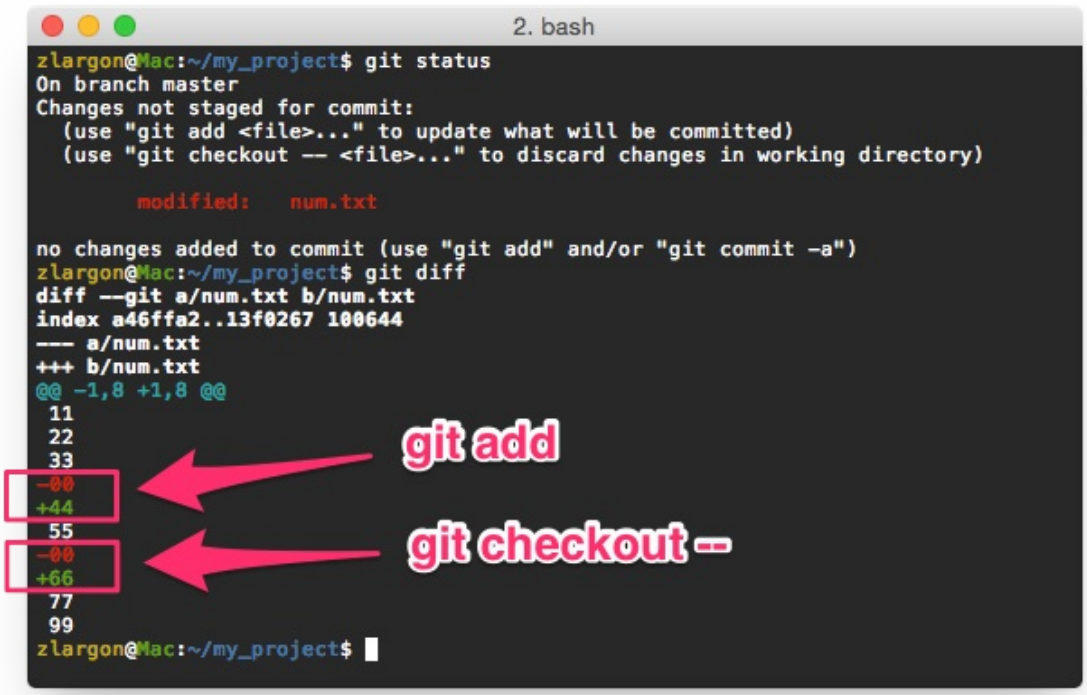
其他部分輸入 n

接著我們用 `git status / diff` 來查看

這樣我們就可以只提交 22 的部分



使用 `git checkout -p` 回復檔案部分的內容



這時候我們想把 "66" 修改的部分取消，那就只要在 `git checkout` 加上參數 `-p` 或是 `--patch`

```
$ git checkout -p
$ git checkout --patch      # 同上
```

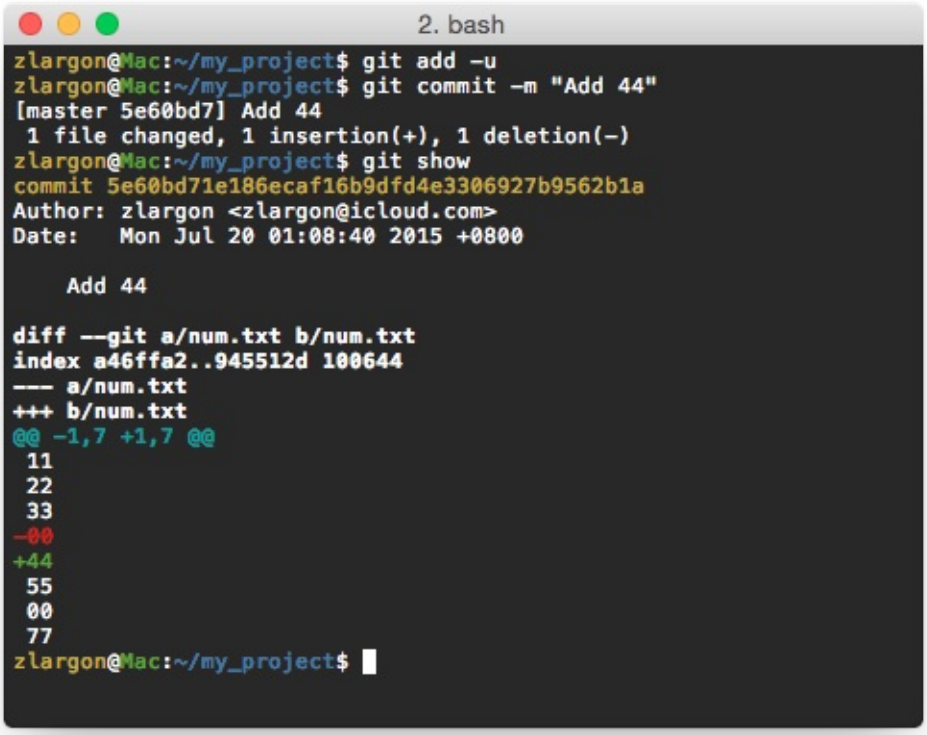
就跟 `git add -p` 一樣，Git 一次會出現一個區塊（hunk），問你要做什麼動作，並且列出選項

```
Discard this hunk from worktree [y,n,q,a,d,/,s,e,?]?
```

輸入 `?` 或是其他不在列子裡面的選項，就會顯示說明

先用 s，把 hunk 切成兩個小 hunk，再把 "66" 的部分 checkout

最後將 "44" 提交出去



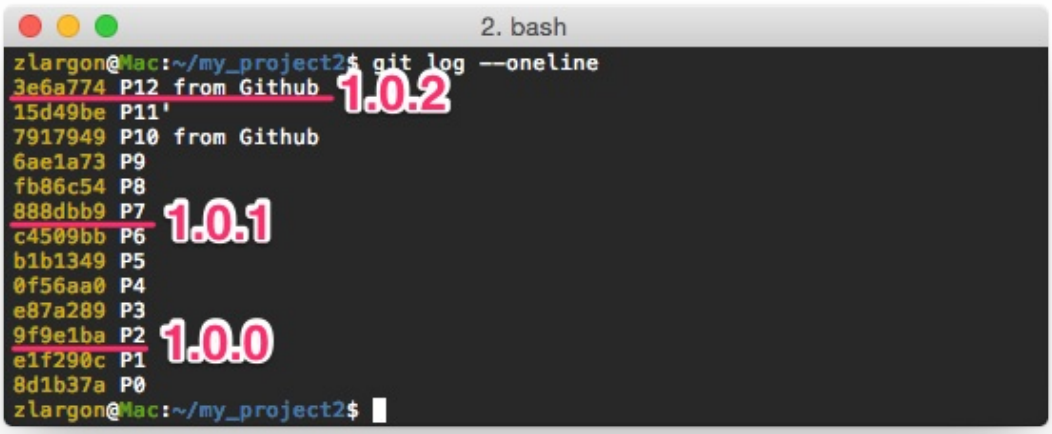
本章回顧

- 使用 `git add -p` 提交檔案部分的內容
- 使用 `git checkout -p` 回復檔案部分的內容

版本標籤

通常我們要 release 的時候，會有一個對外的版本號，例如：1.0.1 之類的

這時候我們就可以爲這個版本加上一個標籤



例如說，我們現在要爲 `9f9e1ba` 新增標籤 `1.0.0`

爲 `888dbb9` 新增標籤 `1.0.1`

爲最新的 patch 新增標籤 `1.0.2`

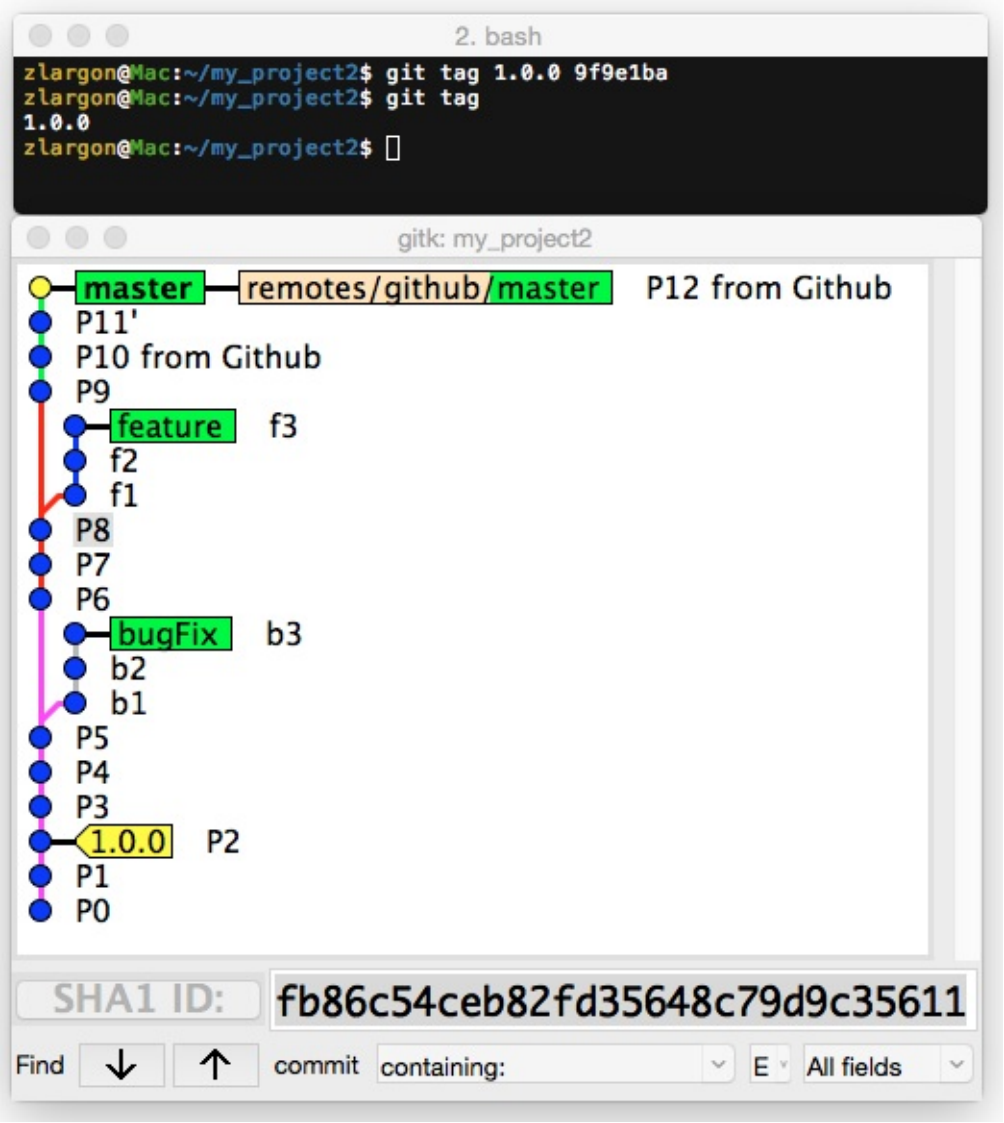
Tag 有分兩種，一種是「lightweight tags」這種可以視爲某個 patch 的「別名」

另外一種是「annotated tags」除了有輕量 tag 的功能之外，還可以寫 tag message

他會紀錄發布者的名稱、日期等等資訊，比較適合正式 release 使用

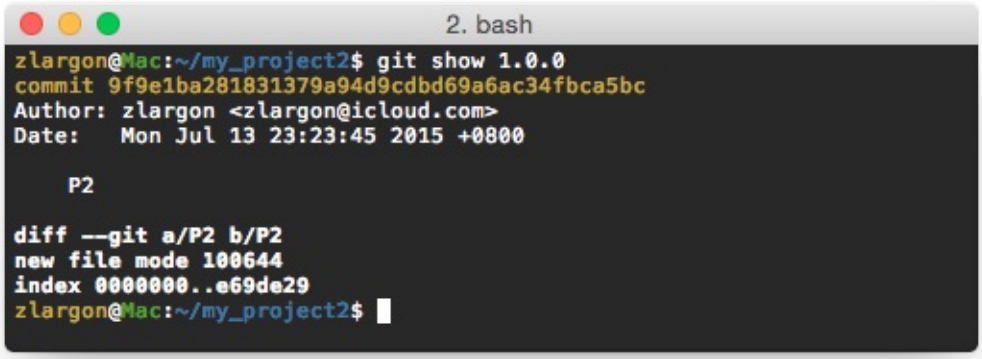
使用 `git tag <tag name> <commit id>` 新增 lightweight tag

```
$ git tag 1.0.0 9f9e1ba      # P2 = 9f9e1ba
$ git tag                    # 查看所有的 tags
```



可以使用 `git show <tag name>` 來查看

`gitk` 會用黃色的標籤來表示 tag



輕量標籤，就只是某個 patch 的別名而已

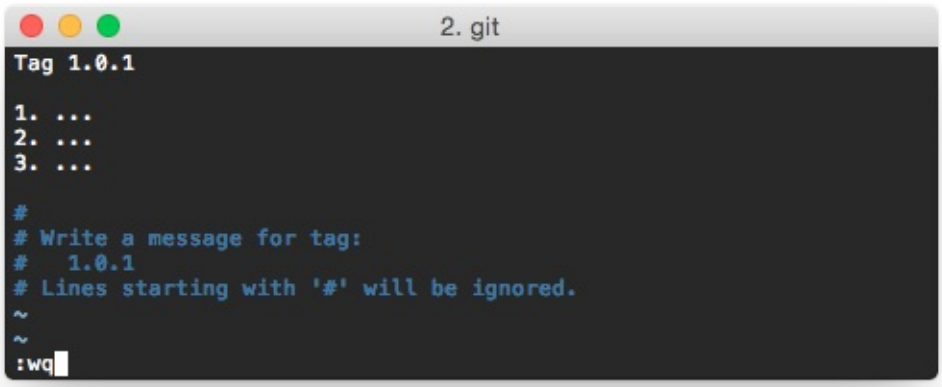
如果不加參數 `<commit id>`，那麼 git 會 tag 目前的 patch（HEAD）

使用 `git tag -a <tag name> <commit id>` 新增 annotated tag

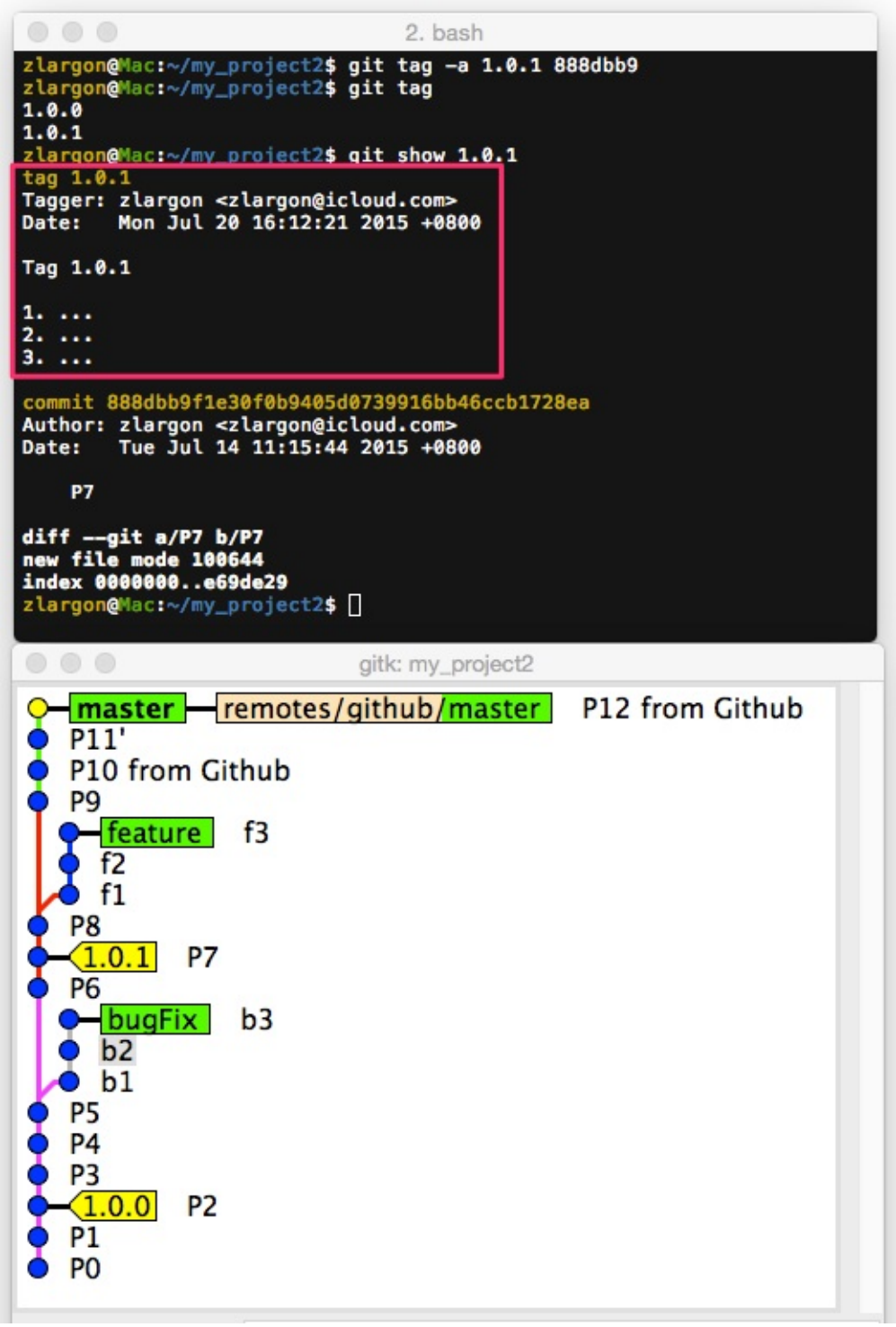
參數 `-a` 等同於 `--annotate`

如果後面沒加 `<commit id>` 就會 tag 目前的 patch（HEAD）

```
$ git tag -a 1.0.1 888dbb9              # 按下 enter 後，會進入文字編輯模式，要求輸入 tag me
ssage
```



我們用 `git show` 來看 tag 1.0.1



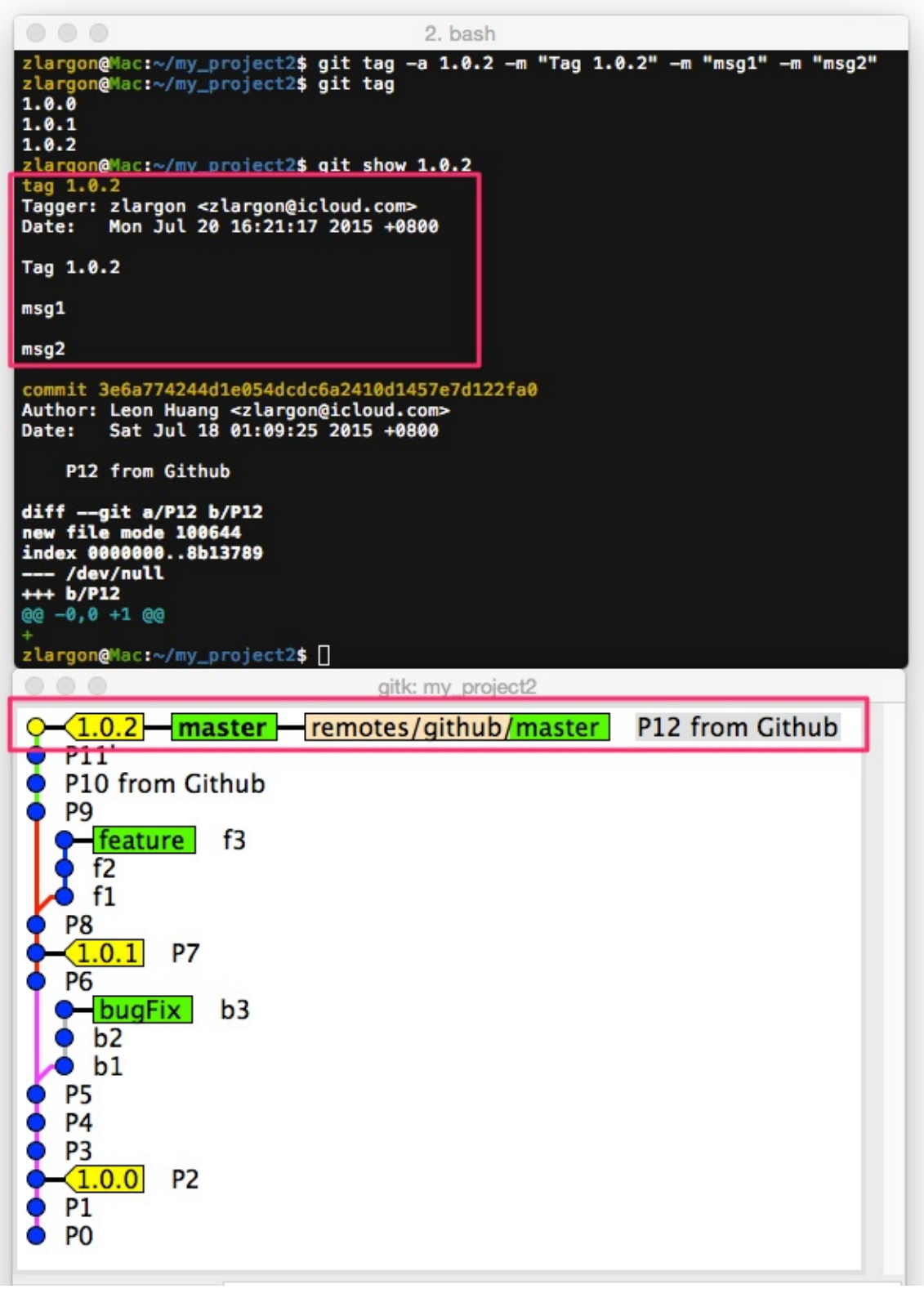
相較於 lightweight tag，他還多了 tag message

使用 `git tag -a <tag name> <commit id> -m <message>` 新增 annotated tag 以及 message

新增 annotated tag 的時候，後面可以加上多個 `-m <message>` 直接輸入 tag message，跳過文字編輯模式

就跟 `git commit -m <msg1> -m <msg2> -m <msg3>` 的用法一樣，可以一次新增多個

```
$ git tag -a 1.0.2 -m "Tag 1.0.2" -m "msg1" -m "msg2"   # 不加 <commit id>，就會 tag 目前的 patch
```

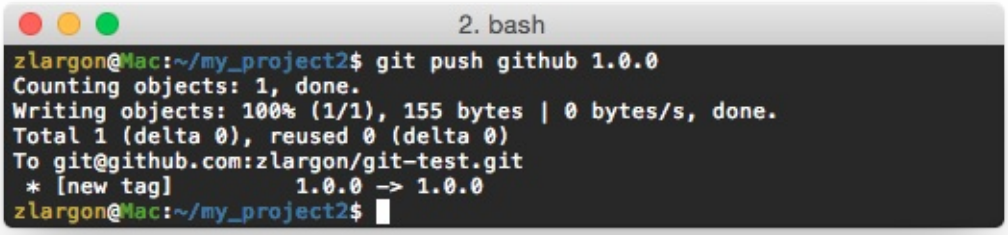


> 注意，這些標籤都還只是本機端的標籤而已
> 稍後我們會再把他上傳到遠端

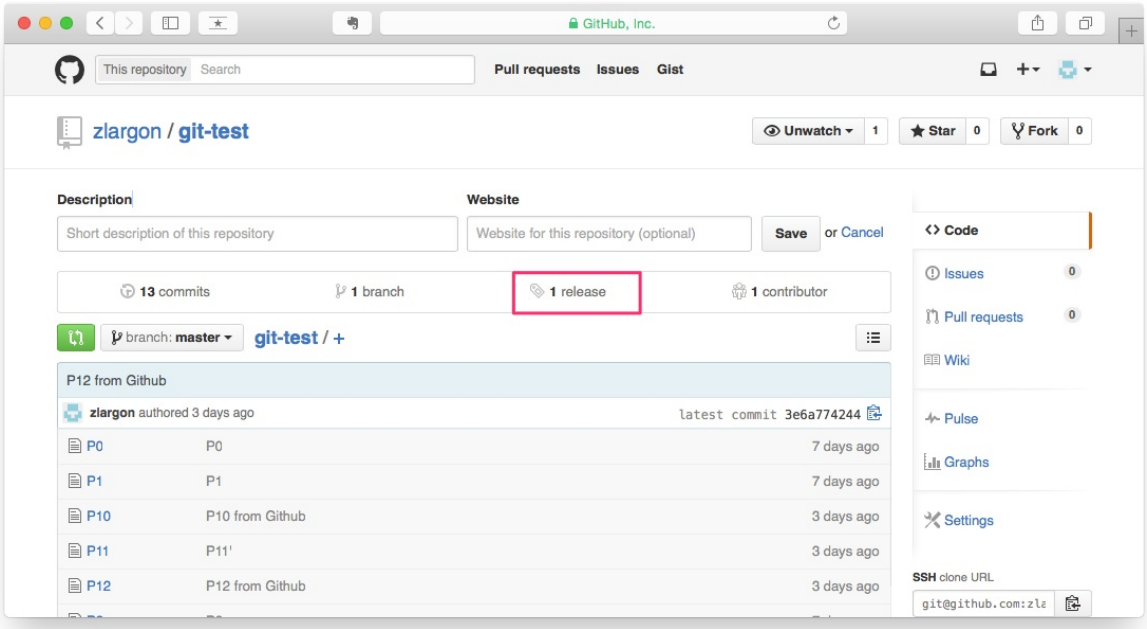
使用 `git push <remote name> <tag name>` 指定上傳的標籤

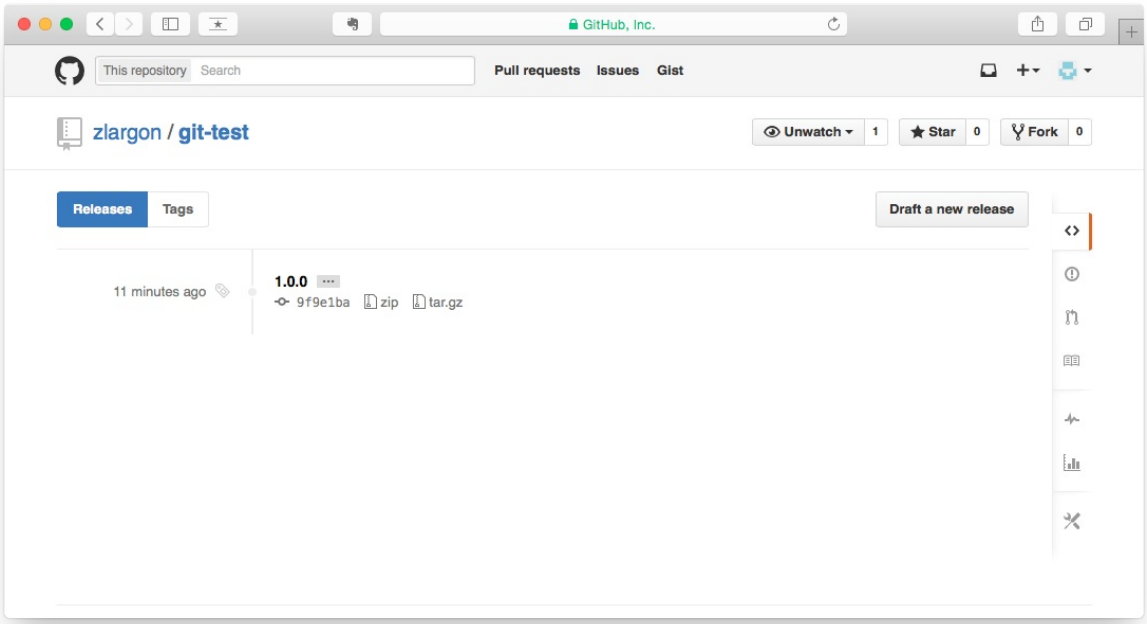
用法就跟 "上傳分支" 到 server 上是一樣的，只是把 `<branch name>` 換成 `<tag name>`

```
$ git push github 1.0.0
```



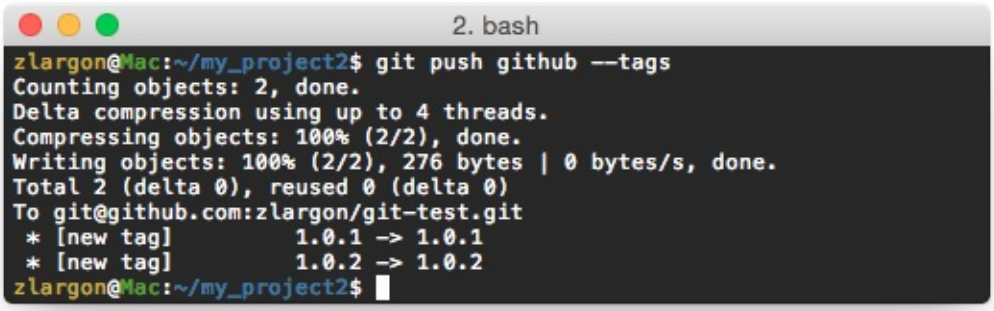
上傳成功後，可以從 Github 的 release 頁面看到我們的 tag

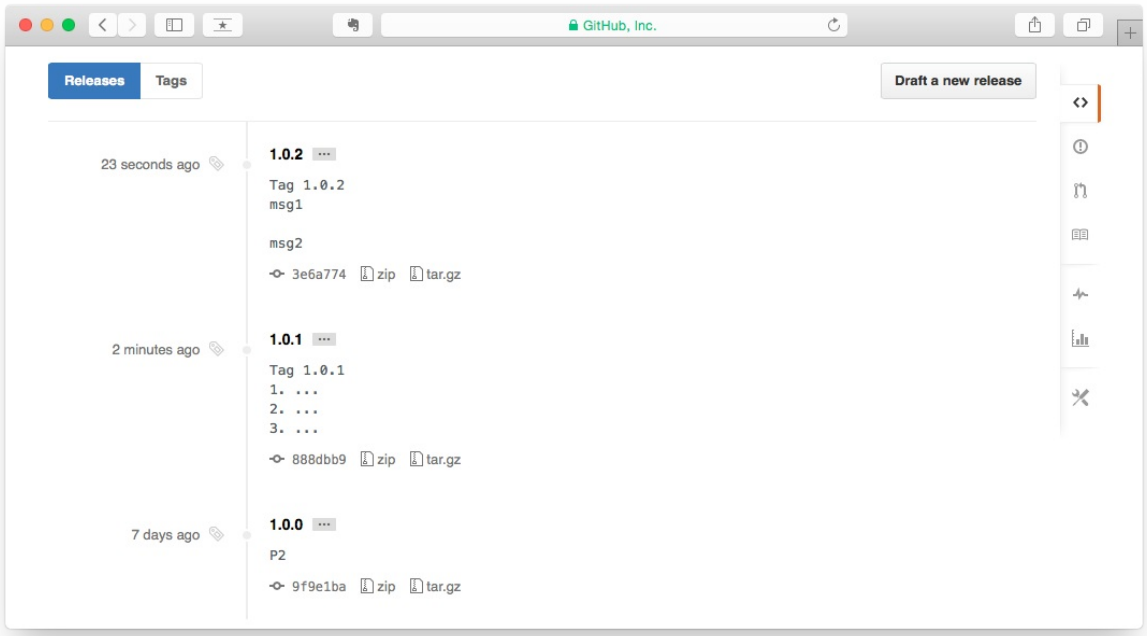




使用 `git push <remote name> --tags` 一次上傳所有的標籤

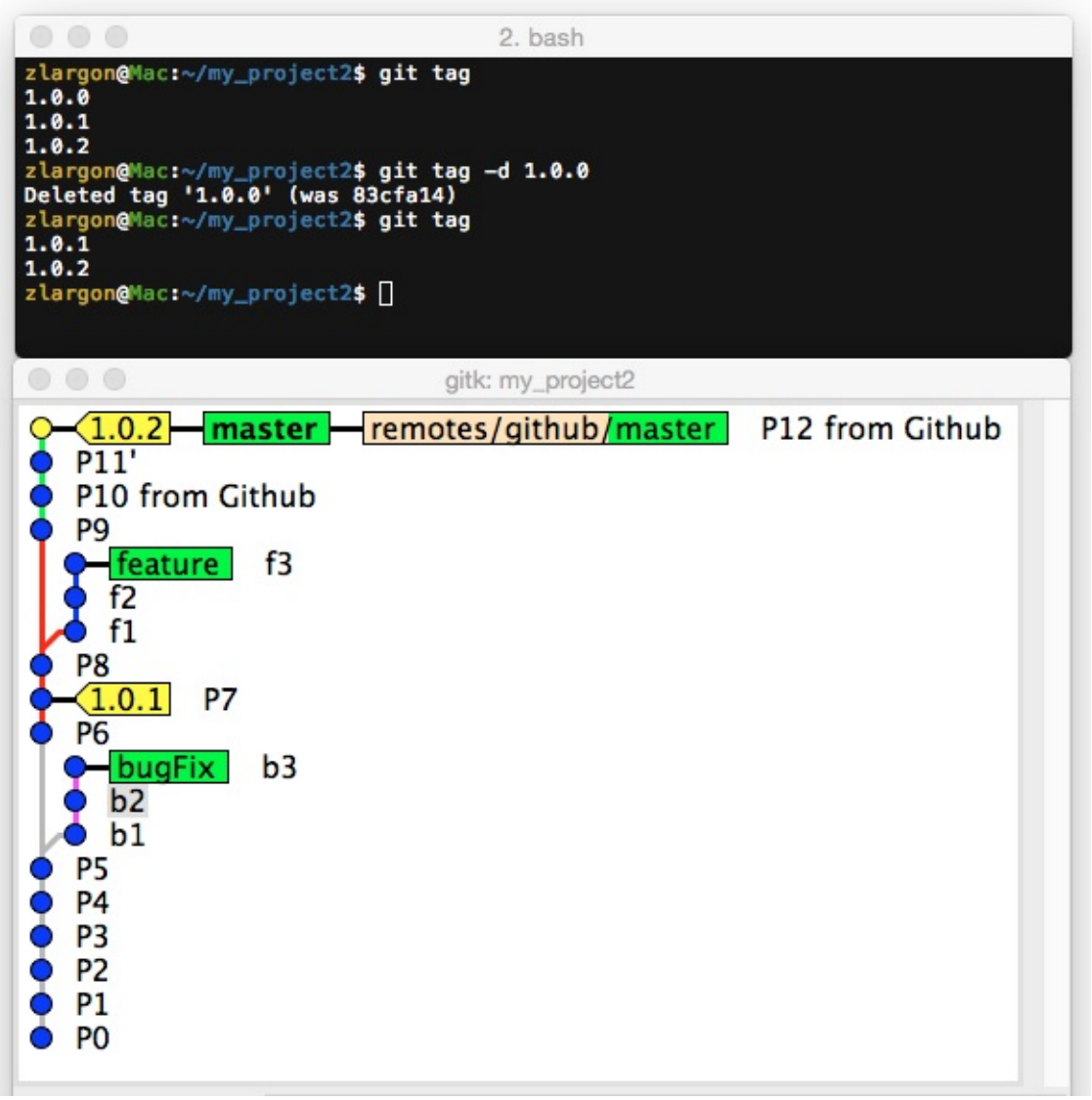
```
$ git push github --tags
```





使用 **`git tag -d <tag name>`** 刪除本機端的標籤

```
$ git tag -d 1.0.0
```



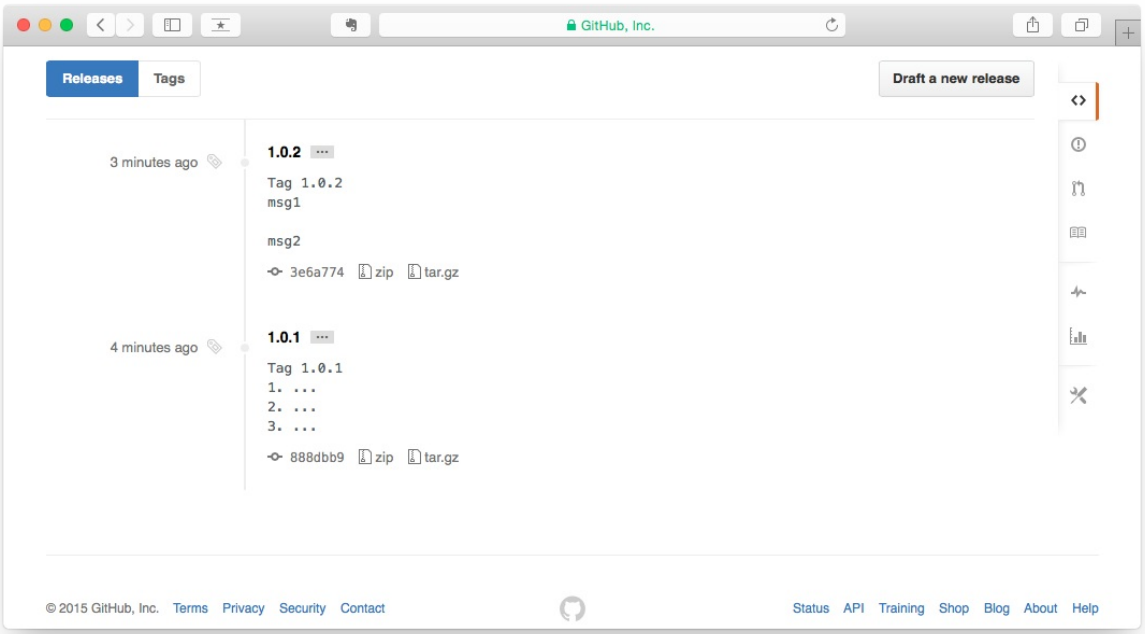
注意，這只會刪除本機端的標籤

所以 Github 上的標籤還是會存在

使用 `git push <remote name> :<tag name>` 刪除遠端的標籤

跟 "刪除遠端分支" 的做法都是一樣的

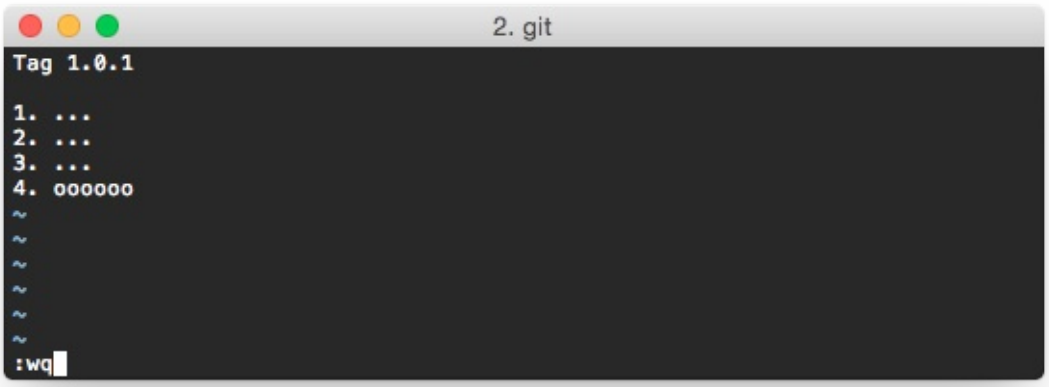
只是把 `<branch name>` 改成 `<tag name>`

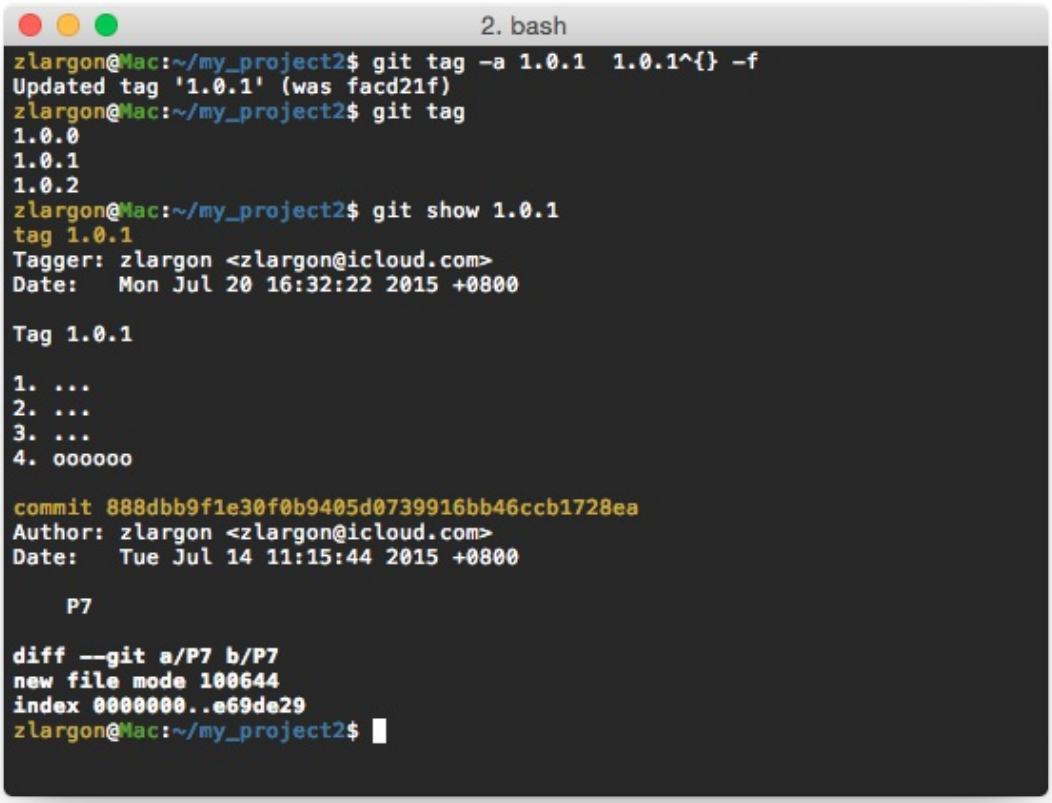


使用 `git tag -a <tag name> <tag name>^{} -f` 更新 **annotated tag** 的訊息

例如說我們要修改 1.0.1 的內容

```
$ git tag -a 1.0.1  1.0.1^{} -f          # 進入 vim 文字編輯模式
```



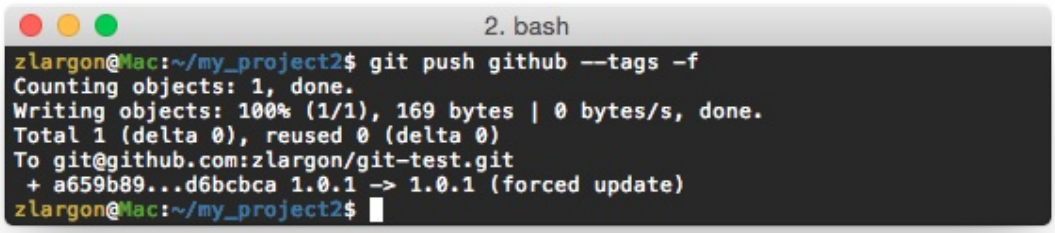


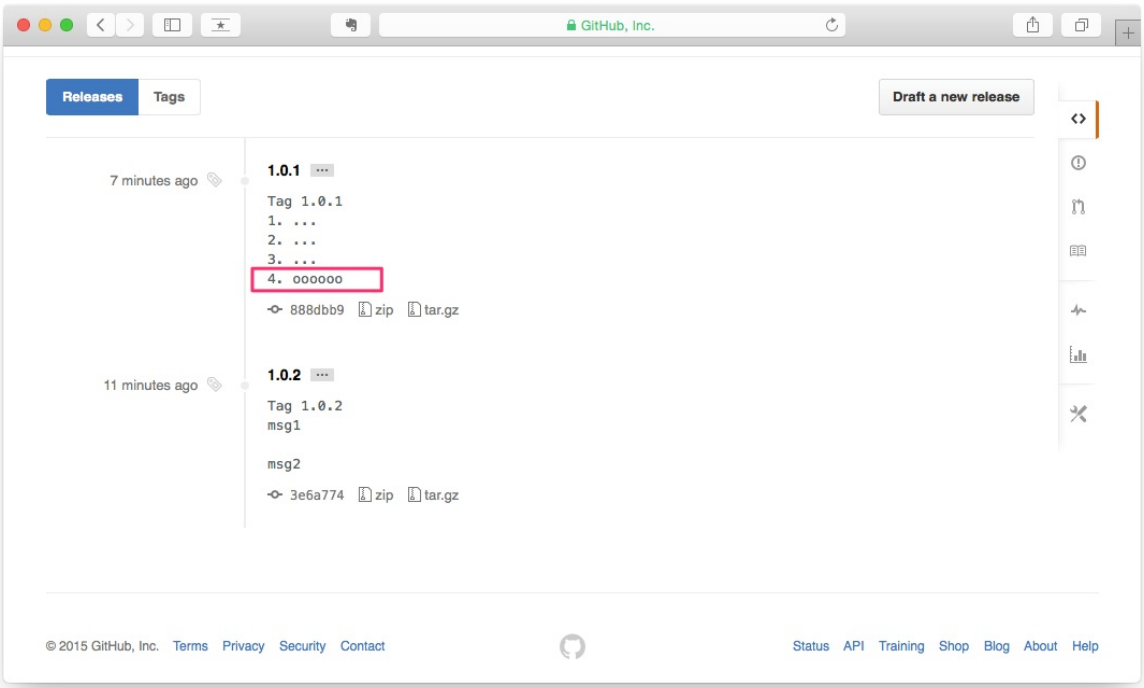
使用 `git push -f <remote name> <tag name>` 強制更新 **remote** 的 **tag**

我們剛才更新了 1.0.1 的訊息，現在要重新上傳到 server

但是會失敗，原因是 server 上已經有 `1.0.1` 的 tag 了

如果想要強制修改 tag 的內容，則可以加參數 `-f` 等同於 `--force`





本章回顧

- 使用 `git tag <tag name> <commit id>` 新增 lightweight tag

  > 省略 `<commit id>` 則會 tag 目前的 patch

- 使用 `git tag -a <tag name> <commit id>` 新增 annotated tag
- 使用 `git tag -a <tag name> <commit id> -m <message>` 新增 annotated tag 以及 message
- 使用 `git push <remote name> <tag name>` 指定上傳的標籤
- 使用 `git push <remote name> --tags` 一次上傳所有的標籤
- 使用 `git tag -d <tag name>` 刪除本機端的標籤
- 使用 `git push <remote name> :<tag name>` 刪除遠端的標籤
- 使用 `git tag -a <tag name> <tag name>^{} -f` 更新 annotated tag 的訊息
- 使用 `git push -f <remote name> <tag name>` 強制更新 remote 的 tag

子模組

通常我們在開發專案的時候，會在 Github 上找一些第三方開發的 library 或是 framework 來用

一般來說，要加入這些第三方的程式，最快的方法就是在我們的專案開一個資料夾，然後全部 copy 一份丟進去

往後如果這些第三方程式有更新的時候，我們就要把重新 copy 一次，把原來放置第三方程式的資料夾整個覆蓋掉

不過 Git 提供了一個方式，讓我們可以把第三方的程式與我們的程式完全拆開

此後第三方程式就可以直接更新

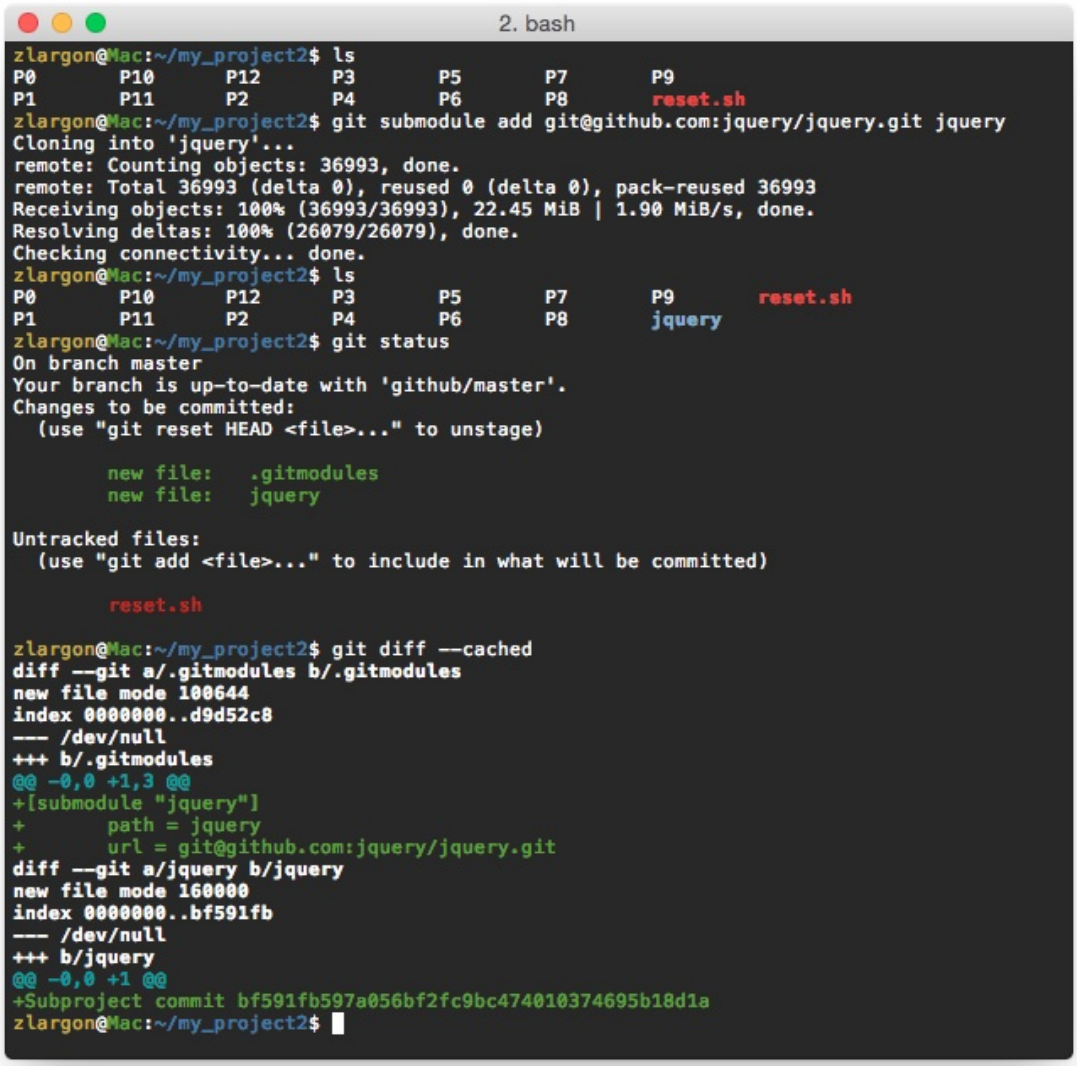
使用 `git submodule add <repo url> <project path>` 新增子模組

舉例來說，如果我們想要把 JQuery 的 source code 放到我們的專案中

https://jquery.com/

https://github.com/jquery/jquery

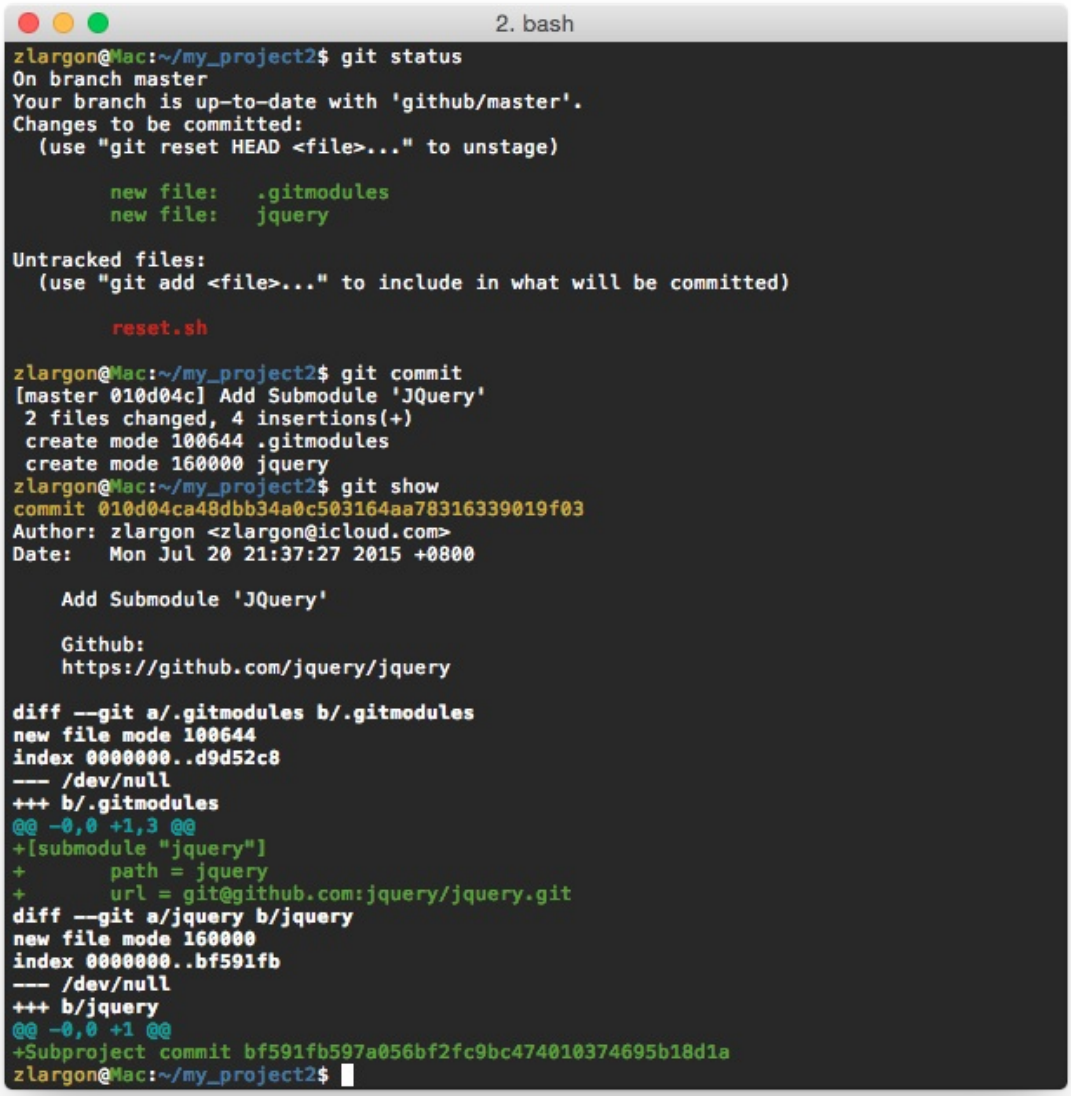
```
$ git submodule add git@github.com:jquery/jquery.git jquery
```



Git 會新增兩個檔案，一個是 .gitmodules，裡面會紀錄所有 submodule 的路徑和 url

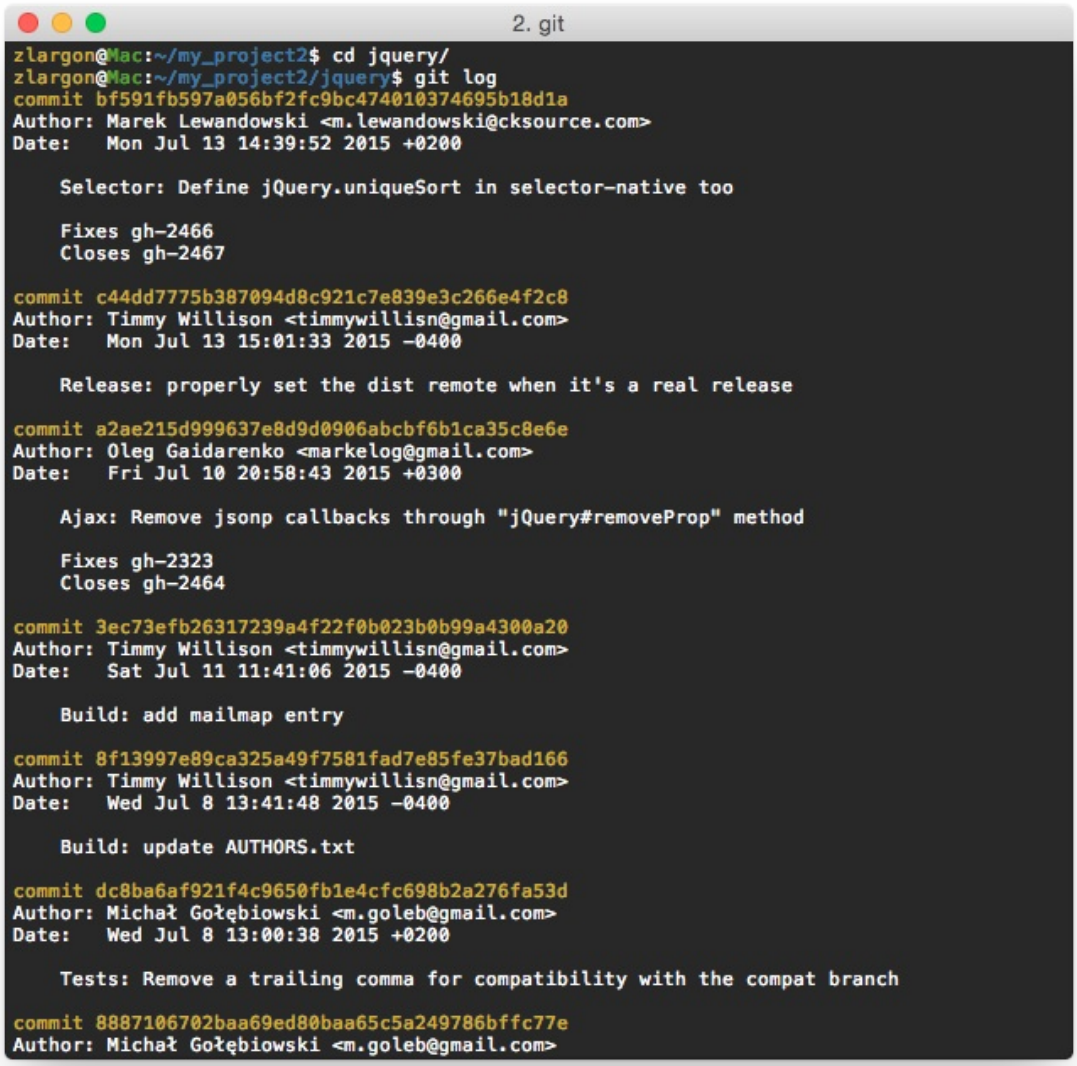
另外一個則是特殊的檔案，紀錄 submodule 的 commit id

這兩個檔案之後必須要提交

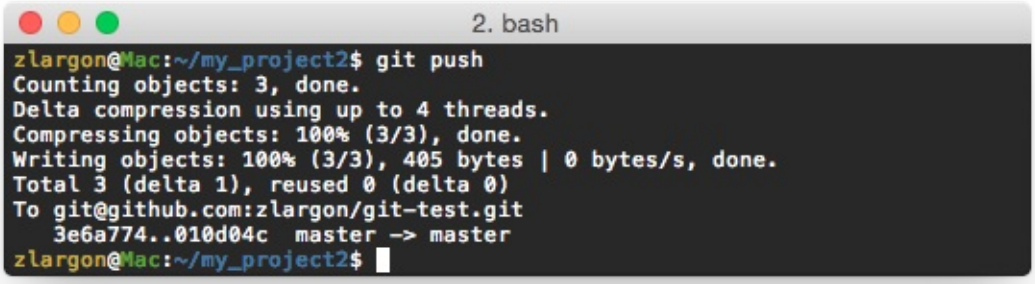


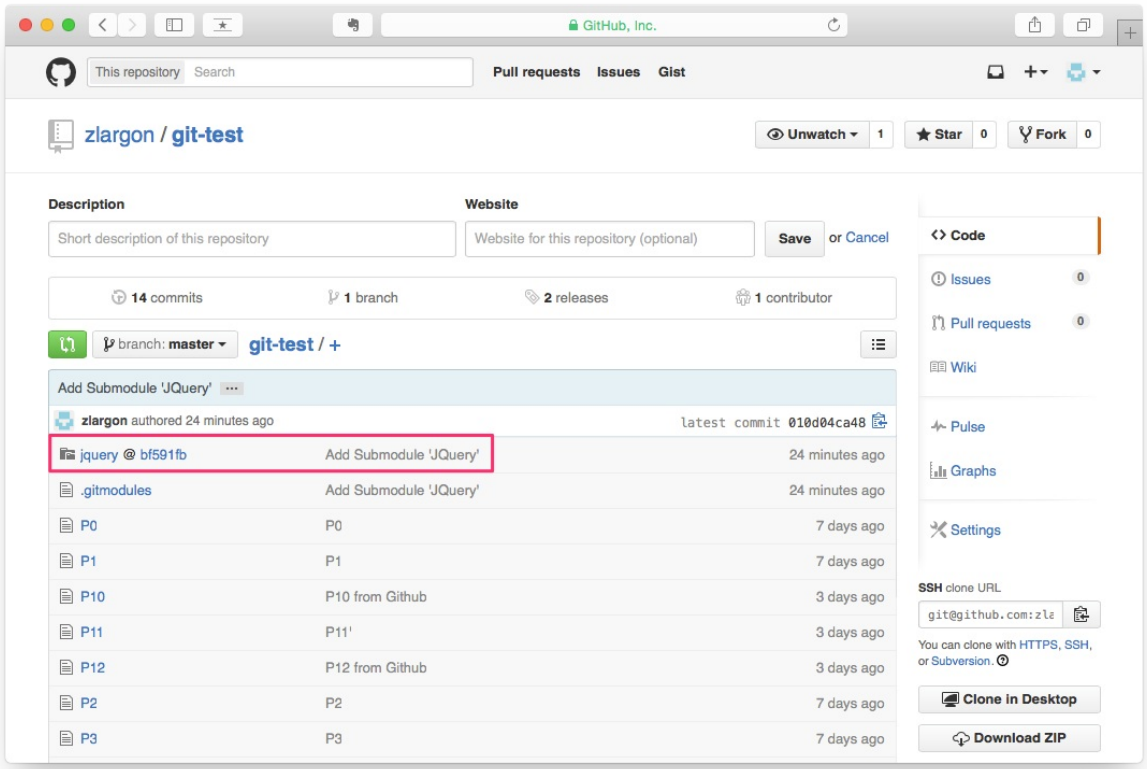
Submodule 本身也是一個 Git Project

當我們進到 jquery 底下的時候，他底下也有一個 `.git`，可以用 `git log` 來查看 jquery 的提交紀錄



接著我們把 patch 上傳到 Github





到 Github 頁面，會看到一個特殊檔案 "`jquery @ bf591fb`"

點連結，會直接跳到 Jquery 的 Github 頁面

使用 `git clone --recursive <repo URL>` 下載包含子模組的專案

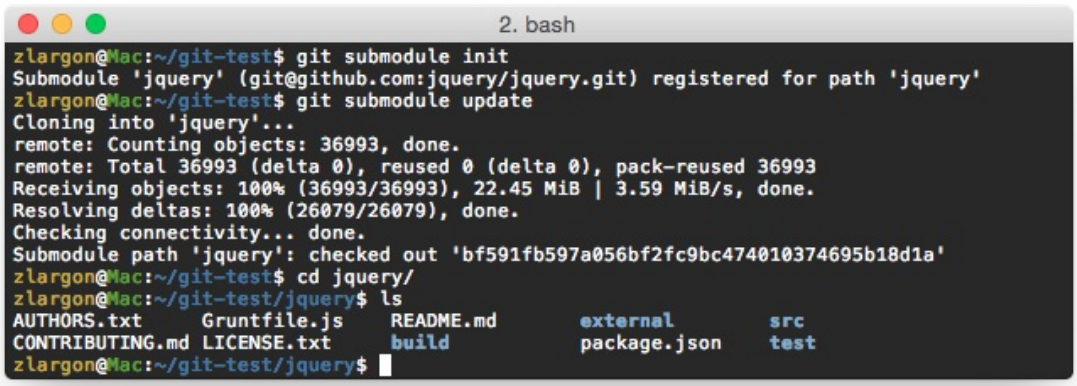
我們現在用 git clone 下載，git-test project

```
$ git clone git@github.com:zlargon/git-test.git
```

我們會發現 jquery 的資料夾是空的

Submodule 必須要另外執行兩個步驟，才會被 clone 下來

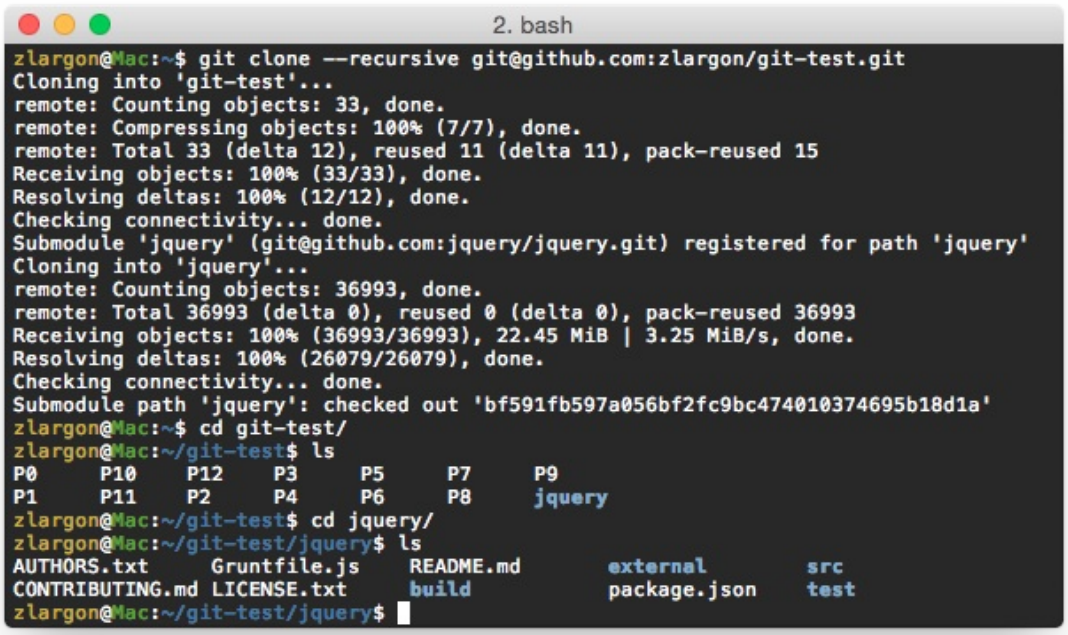
```
$ git submodule init
$ git submodule update
```



除此之外，可以在 `git clone` 的時候，加上參數 `--recursive`

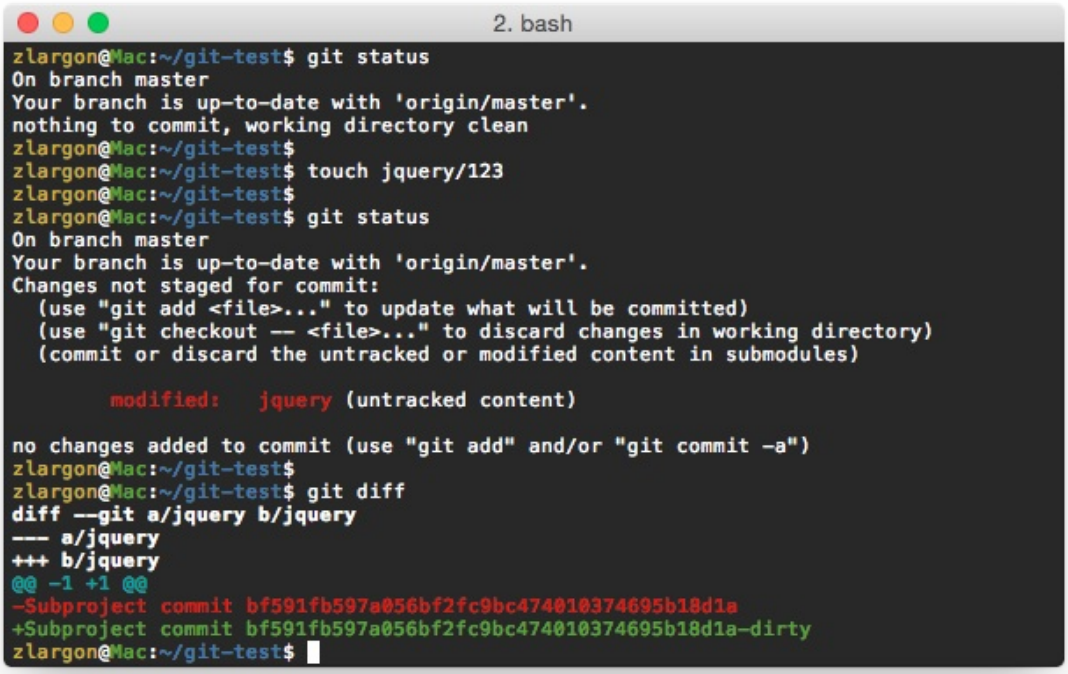
他會自動幫我們把全部的 submodule clone 下來

```
$ git clone --recursive git@github.com:zlargon/git-test.git
```

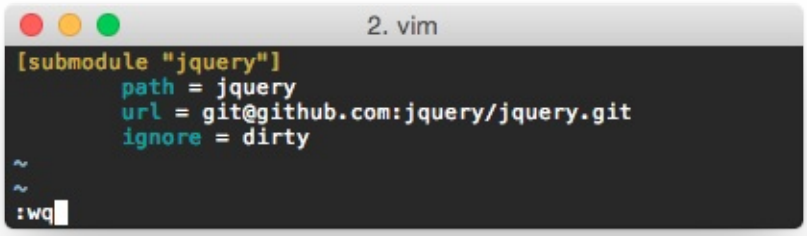


設定忽略 Submodule 內部的改動

當 jquery 內部有改動的時候，外部的 `git status` 會出現 jquery 被修改的提示



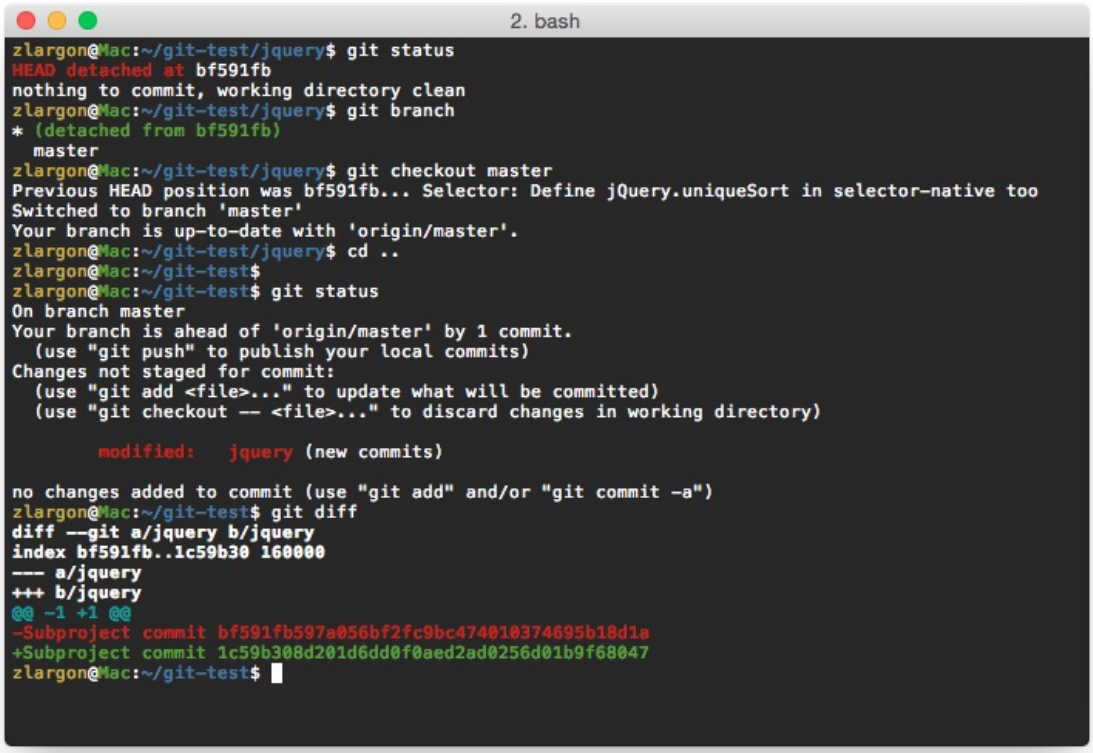
如果希望不要出現這些提示的話，可以在 `.gitmodules` 加上設定 `ignore = dirty`



修改完記得要提交

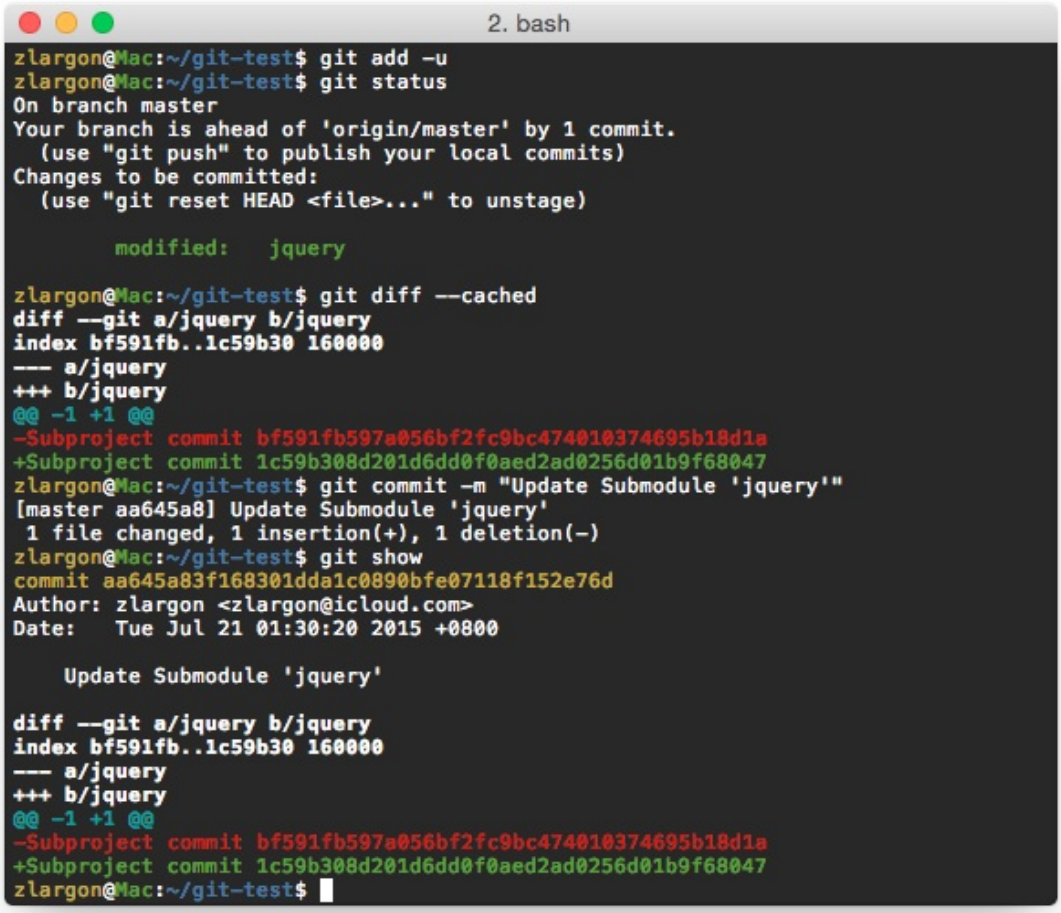
升級 / 切換 Submodule 版本

到 submodule 的資料夾下面 checkout 你想要的 patch



切換完畢後，回到上層做 `git status` 會看到 `jquery (new commits)`

這時候只要 git add，並且提交出去就可以了



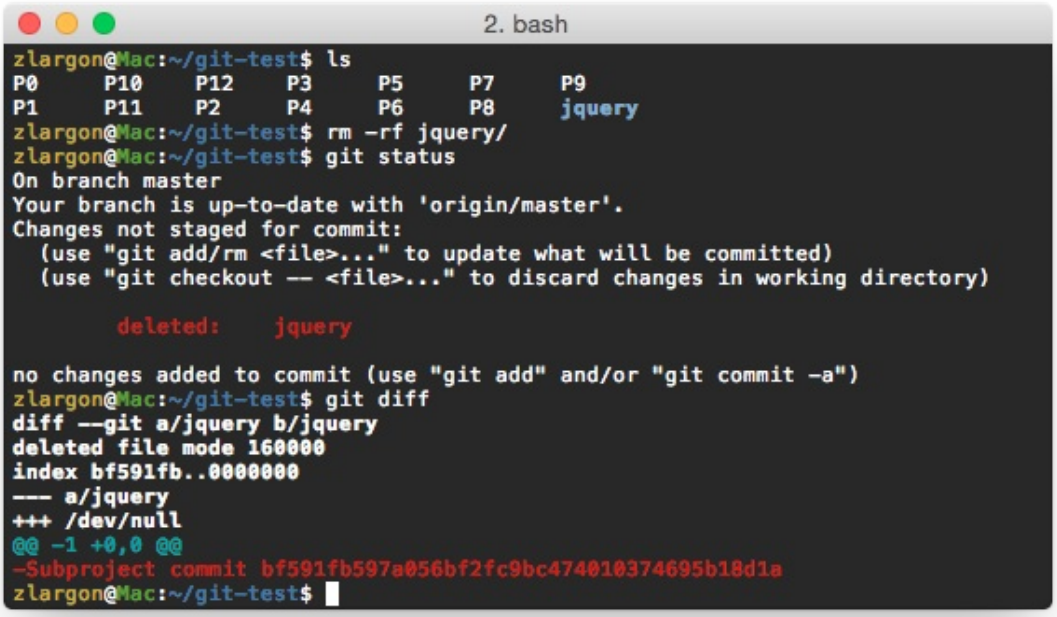
刪除 Submodule

Git 目前還沒有任何指令可以下指令把 submodule 刪除，不曉得爲什麼 @@"

希望以後會有類似 `git submodule rm` 之類的指令

1. 刪除 submodule 的資料夾

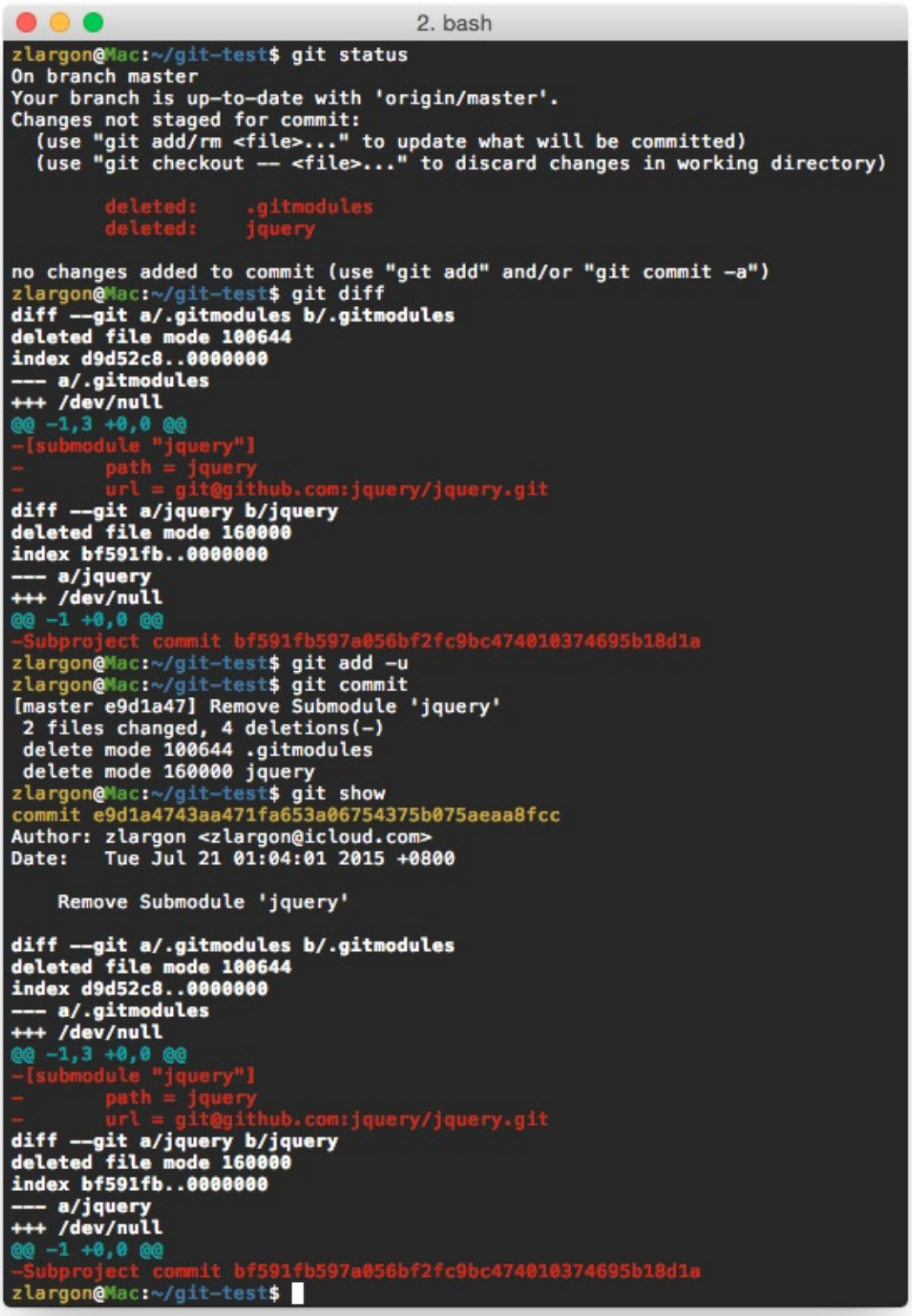
```
$ rm -rf <submodule path>
```



2. 刪除 `.gitmodules` 裡面 submodule 設定

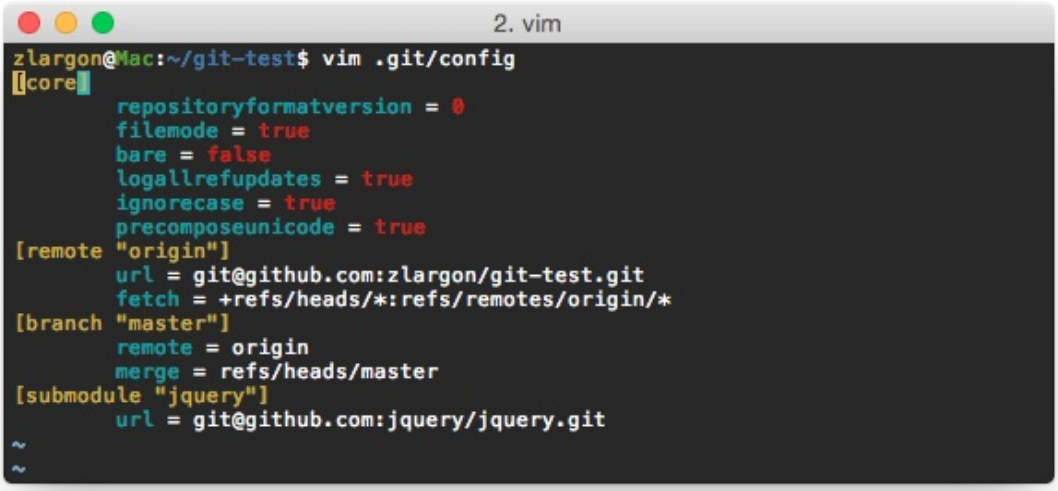
```
[submodule "jquery"]
    path = jquery
    url = git@github.com:jquery/jquery.git
```

3. 提交 patch



4. 刪除 `.git/config` 裡面的 submodule 設定

```
[submodule "jquery"]
    url = git@github.com:jquery/jquery.git
```



5. 刪除資料夾 `.git/modules/<module name>`