寫給大家的 Git 教學

第三版，2012

Littlebtc (Hsiao-Ting Yu)

改寫自 Scott Chacon 的「Pro Git」電子書。

Part 1

Why Git

為什麼要版本控制？為什麼要用 Git？

這是程式設計師的日常

當你寫了一個新東西，卻發生了以下的悲劇…

- 忘了自己改了哪些地方
- 把舊的東西改壞了，改不回來
- 拿去跟別人的成果合併，卻兜不起來

→有沒有辦法解決？

有！「版本控制系統」

當你使用版本控制系統時：

- 忘了自己改了哪些地方→每次的更動都會被記錄下來
- 把舊的東西改壞了，改不回來→可以隨時退回到過去的版本
- 拿去跟別人的成果合併，卻兜不起來→提供機制處理多人協作的衝突

→「凡走過必留下痕跡」、「三個臭皮匠，勝過一個諸葛亮」，這就是版本控制的精神。

版本控制系統的演進

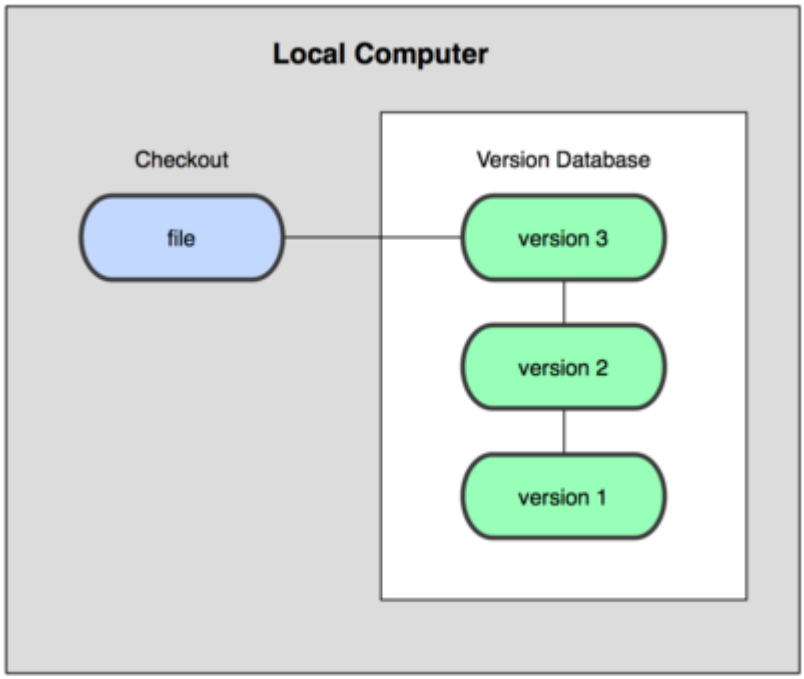
- 單機式 (rcs)
- 中心式 (CVS、Subversion)
- 分散式 (Git、Mercurial、Bazaar)

版本控制系統的演進

單機式

為了達成「凡走過必留下痕跡」：

- 本機中建立一個資料庫
- 記下每個檔案的版本變更



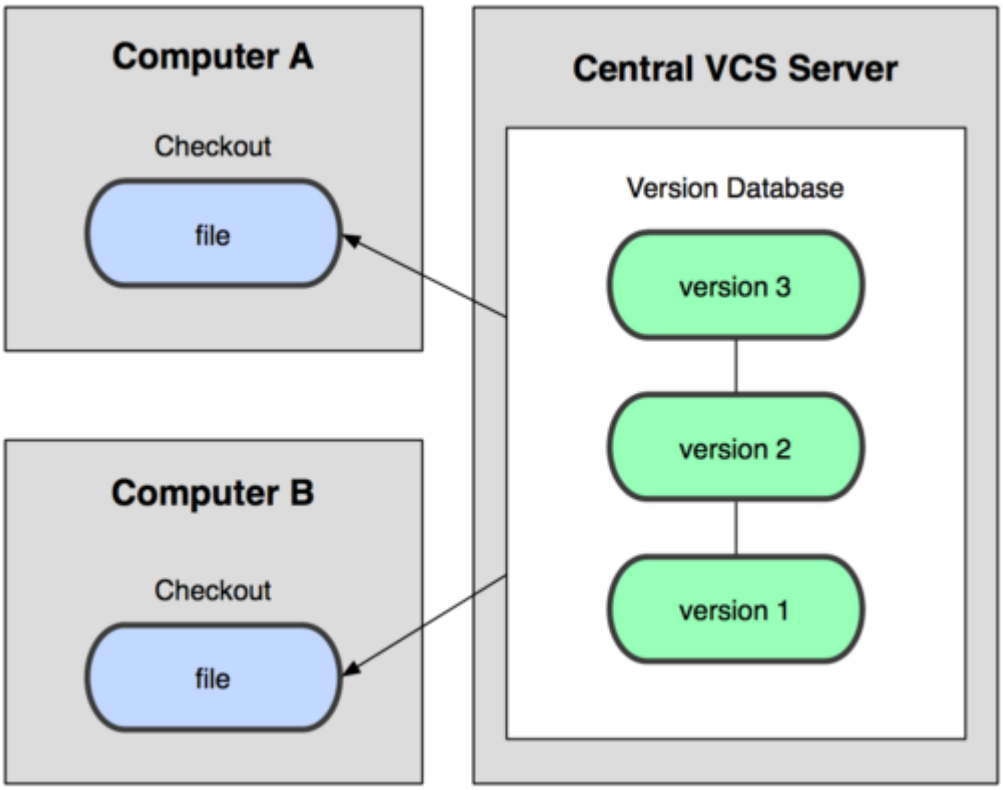
rcs (1982) 屬於此類（至今還有人使用！）

→問題：兩人以上協作時，如何同步每個人的版本資料庫？

版本控制系統的演進

中心式

為了讓三個臭皮匠可以一起作業：



- 版本資料庫放在中心統一控管
- 每個人從中心取出（Checkout）東西
- 修改完後將內容提交（Commit）回中心

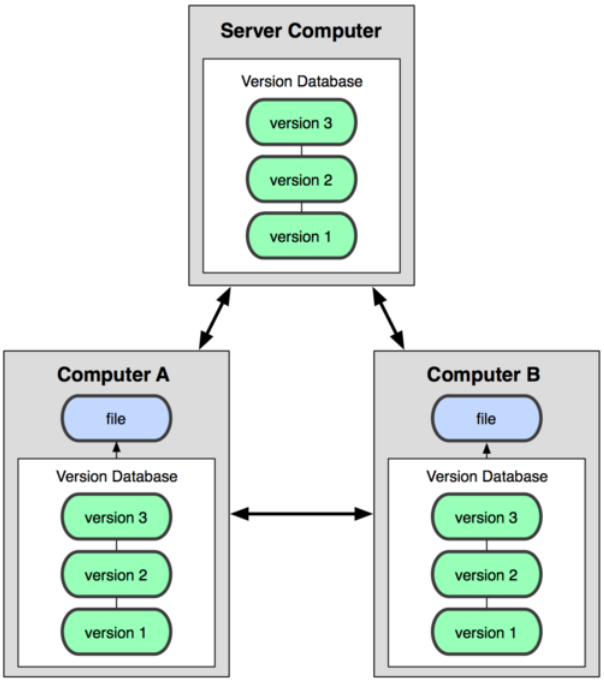
早期盛行CVS (1990)、現在主流為 Subversion (2000)

→中心一旦故障，大家的作業都會出問題，該怎麼辦？

版本控制系統的演進

分散式

…何不讓每個人都有一份完整的資料庫？



- 「大家都能獨立工作」
- 「Server 爛了？不要緊，拿到一份好的資料庫灌回去就可以全部復原！」

Git (2005) 是主流、Mercurial (2005) 跟 Bazaar (2005) 也很盛行

→又來了：怎麼解決每個人之間的同步問題？
引進非線性的開發模式！

Git 的歷史

簡言之：為了解決 Liunx 核心的開發問題，所以有了 Git

- 過去，Linux 核心開發沒有使用版本控制系統（主因是 Linus Torvalds 不喜歡中心式的系統）。
- 2002 年，開始改用專有的分散式版本控制系統 BitKeeper，讓自由軟體/開源社群不滿（自由的東西用非自由的開發工具，成何體統？）。
- 2005 年 BitKeeper 跟 Linux 開發團隊鬧翻，不再免費供應系統。Git 的開發自此開始。

Git 的強項

- 快
- 簡單
- 非線性開發
  （分支、合併、不受單一主線拘束）
- 完全分散式
- 處理超大資料的能力
  （Linux 核心的程式碼超過一千萬行！）

這些專案都在用 Git

- Linux Kernel
- Android
- GNOME
- KDE
- PHP (Since 2012)
- Ruby on Rails
- django

Part 2

Setup Git

設定屬於你的 Git 開發環境。

設定 Git

Git 的操作大部分都是透過命令列，但也有許多的圖形介面工具。

無論如何，以下的設定都是必要的：

- 安裝 Git 軟體
  這不用說了吧 XD
- 產生一個 SSH 金鑰
  為什麼？因為 Git 可以利用 SSH 進行使用者驗證
- 調整 Git 的組態
  設定姓名和電子郵件地址

有懶人包幹嘛不用？

GitHub 他們寫了三種平台下安裝 Git 的懶人包，以下提供連結：

- Windows：http://help.github.com/win-set-up-git/
- MacOS X：http://help.github.com/mac-set-up-git/
- Linux：http://help.github.com/linux-set-up-git/

「Add your SSH key to GitHub」這個步驟是 GitHub 的設定，可以跳過去 :)

Windows

照上面懶人包安裝後，您會把 Git for Windows 安裝好。

Git for Windows 有兩種不同的使用方式：

- Git Bash：一個 Bash Shell，你可以在裡頭以命令列方式使用 Git。
- Git GUI：一個簡單的圖形使用介面。

圖形介面工具推薦

- TortoiseGit
  免費，在檔案總管下使用，需先安裝 Git for Windows
- Git Extensions（免費）
- SmartGit（跨平台，非商業使用免費）

MacOS X

安裝完後在「工具程式」→「終端機」以命令列使用 Git。

（Mac 下另外建議您使用 iTerm 2 取代內建的終端機程式）

圖形介面工具推薦

- SourceTree
  免費，也支援 Mercurial 和 SVN
- GitX
  免費，有只能瀏覽歷史和 Commit 的原始版，和後人改過的全方位版
- GitHub for Mac（免費）
- Tower（USD $59）
- Xcode 4 內建 Git 支援（需要 Lion 以上）

Linux

在終端機中以命令列使用 Git。

圖形介面工具推薦（GNOME）

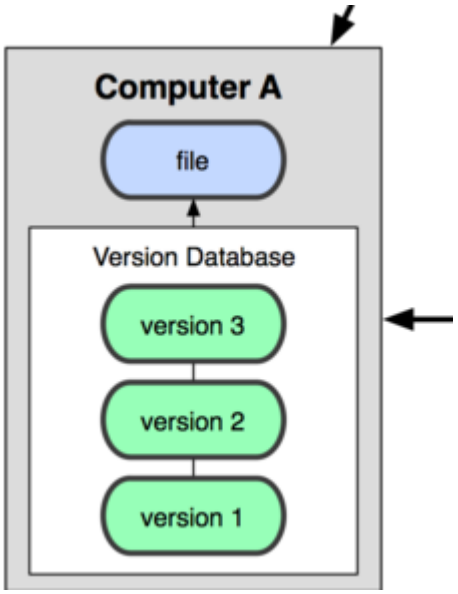
- RabbitVCS
  免費，Nautilus 檔案管理員下操作
  Ubuntu 12.04 內附的版本沒有 Git 支援，請參照官網設定 PPA 安裝
- gitg
  GitX 的 GNOME fork
- giggle

Part 3

Git Basics

介紹如何建立一個 Git 的 Repository，並在其中新增新的 Commit。

所有事都能單獨在本機進行！



- 你不只有一份完整的資料庫……
- 閱讀版本歷史、提交變更這些動作都可以在本機進行
- 不需要網路連線也能單獨工作！

建立 Repository

- 「Repository」（倉庫、套件庫）是 Git 對版本資料庫的稱呼。
- 下達 `git init` 後，會在該目錄裡建立一個 Repository。
- Repository 所需的檔案會放置在 `.git` 目錄之中。（因此，千萬不要誤砍了！）

為了方便解說，以下操作全部都在命令列下進行。
若使用圖形介面，請試著找出相同的使用方法吧！

建立 Repository

範例：建立一個空的目錄叫 playground，在那裡建立 Git Repository：

- `mkdir playground`
- `cd playground`
- `git init`

Tips: 其實也可以用 `git init playground` 一次建空目錄和 Repository

基本觀念：Commit

- 版本控制就是把當時 Repository 內所有檔案的現狀作記錄與控制。
- 而在 Git 之中，每一次的「現狀紀錄」稱之為 Commit。
- Commit 中也包含作者、時間、紀錄資訊（Log）、前後對應 Commit 等資訊，方便追蹤管理。

SHA: d1c1f204e69ca4dc1d44cea6b317938ecac5004e
Author: Hsiao-Ting Yu <sst.dreams@gmail.com>
Date: Sun Apr 01 2012 17:21:03 GMT+0800 (CST)
Subject: 「有病的 Spinner」 by medicalwei
Refs: 4.5.3
Parent: 408a9e6255ce269fa9d49558a10654260a627d5f

「有病的 Spinner」 by medicalwei

created app/assets/images/spinner-sumidagawa-63px.gif
changed app/assets/stylesheets/graph.css.scss
changed app/views/fengyuan_players/graph.html.erb

基本觀念：Staging

- 在您的目錄（稱為 Working Directory，工作中目錄）之中作業。
- 唯有放進 Staging Area（暫存區）裡的更動會被 Commit。
- 為什麼？(1) 多項目同時作業下，可以分開切成多個 Commit，方便管理。 (2) 不用擔心未完成或暫存的東西影響到 Commit。
- 就算是同一檔案，也可以只有某幾行加入 Staging 中！

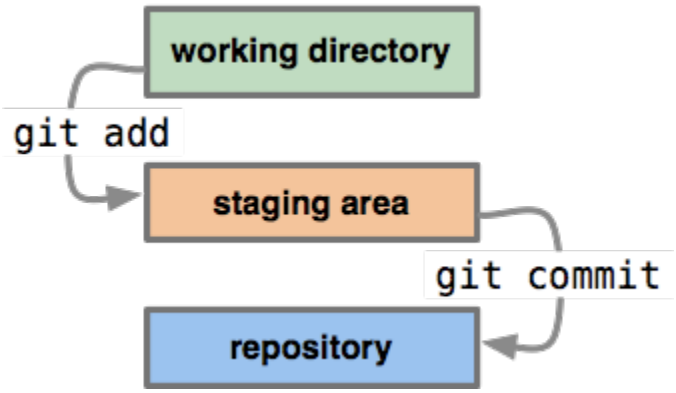
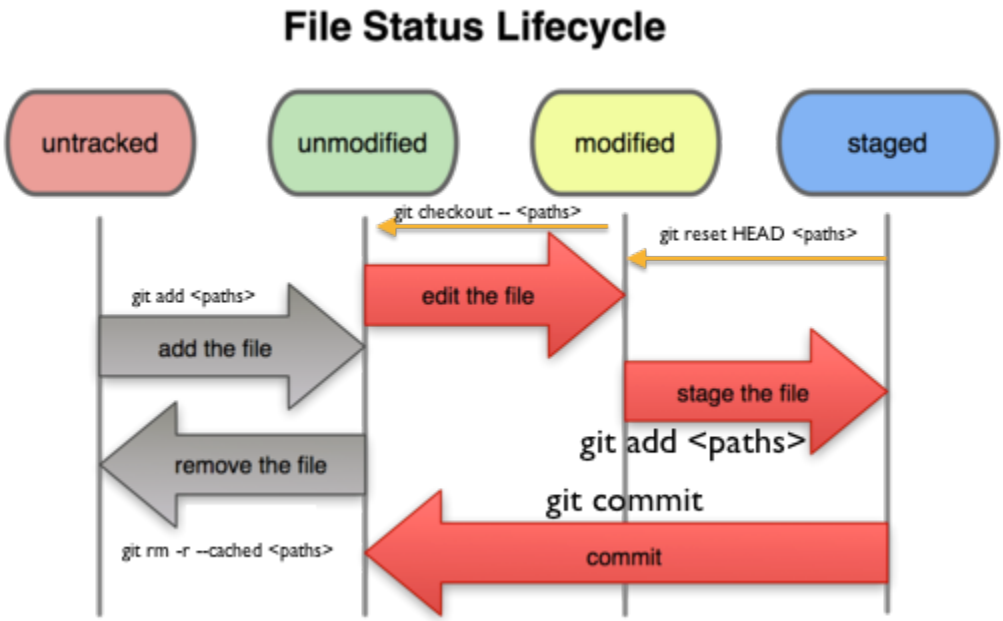


Diagram from Why Git is Better than X, MIT License

基本觀念：檔案狀態

- untracked：沒有納入版本控制範圍內的檔案。
- unmodified / modified：沒有變動 / 有變動還沒有 Staging 的檔案。
- staged：Staging Area 中的檔案。

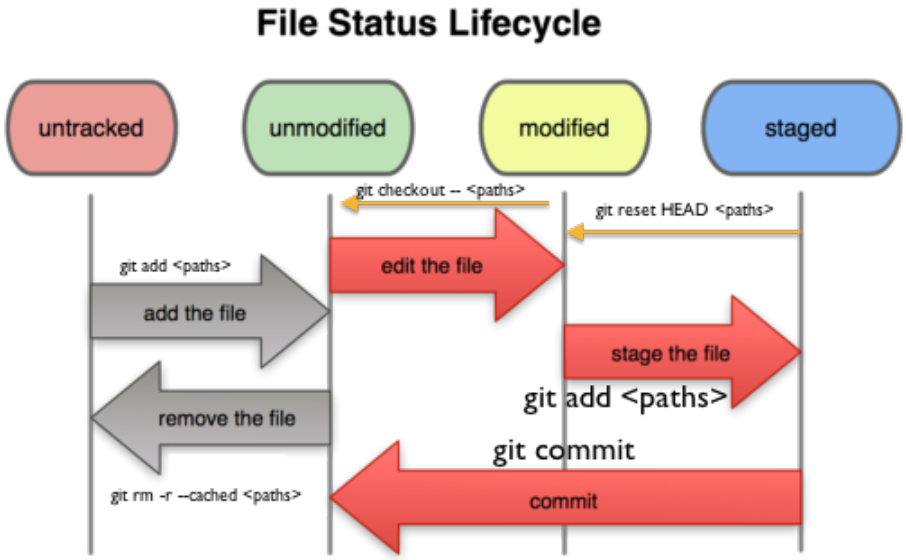


相關的指令（新增）

- `git add <paths>`: 把檔案新增或變動加入 Staging Area。
- `git status`: 檢視檔案狀態。
- `git commit`: 開啟編輯器，確認變動並輸入訊息後送出 Commit。
  指令中加 `-m 'Commit 訊息'` 就會直接 Commit
- `git commit --amend`: 「更動」上一次的 Commit
  將上次 Commit 跟新的更動合併為新的 Commit。

小秘訣: `git commit -a` 會將所有變更都加入 Staging Area 之後 Commit。（但還是建議您善用 Staging :P）

相關的指令（移除或補救）



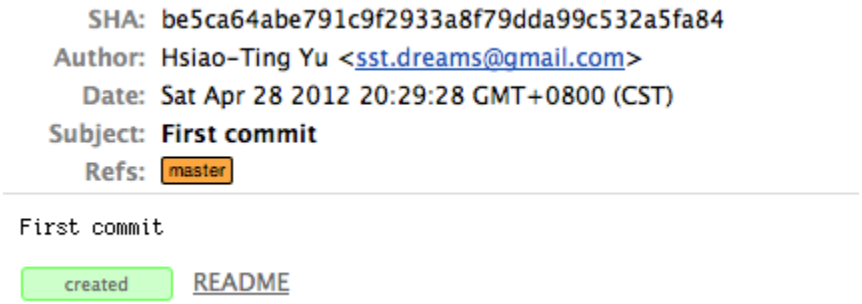
- `git reset HEAD <paths>`：把檔案中 Staging 的部份注銷掉。
- `git rm <paths>`：將檔案從版本控制中移除後，刪除檔案。
  加 `-r` 採遞迴方式，`--cached` 則不刪除原來檔案。
- `git checkout -- <paths>`：恢復檔案為未更動的狀態（對 Staging 的部份無效）。

實例：第一個 Commit

用 `touch README` 建立一個空的 README 檔案後：

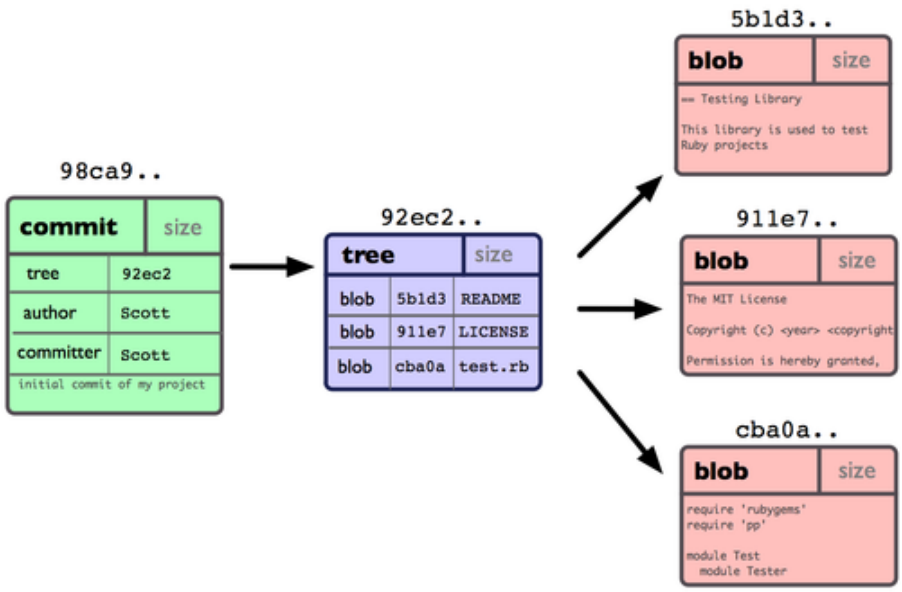
```
git add README
git commit -m 'First commit'
```

這樣就會建立一個 Commit，將 README 納入版本控制之中。



Commit 的內部結構

- 利用 `tree` 存放檔案一覽，將檔案存在 `blob` 裡。
- 所有的資料都以 SHA1 checksum 標記，防止損毀或中途更改。
- Git 針對每個檔案版本獨立紀錄，而非只紀錄之間的差異，更具便利性。



p.s.「我不要這個 Commit！」

有兩種方法：

- 如果你還沒 Push 給別人（後述），您可以直接把整個樹退回到前次 Commit 時的狀態：
  `git reset HEAD^`
- 如果你已經 Push 給別人，別人已經有您新的 Commit 了，因此上述的救法無效。您可以用 Revert，製造一個跟目前 Commit 差異完全相反過來的 Commit：
  `git revert HEAD`

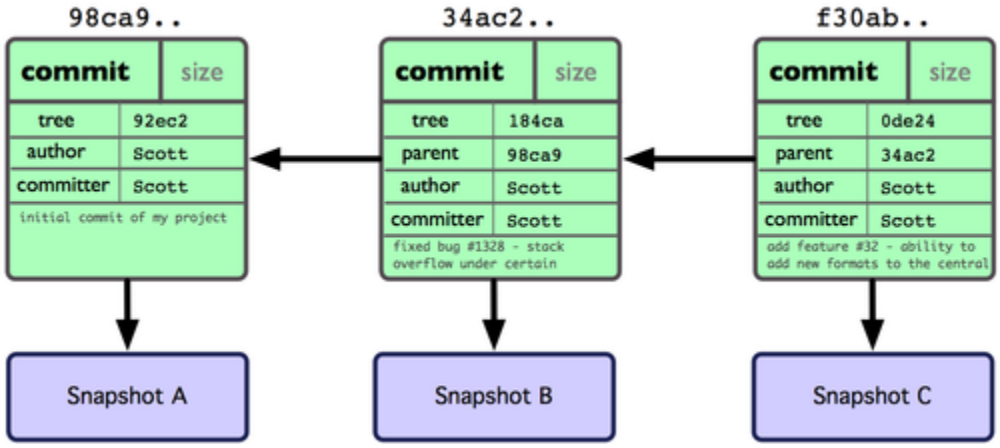
Part 4

Tree Management

介紹 Git 樹狀的歷史管理方式，Branch 和 Tag 的使用，
以及 Merge 和 Rebase 的概念。

當 Commit 超過一個時...

- Git 會用樹狀的方式紀錄 Commit 的繼承關係。
- 其實踐方式就像是 Linked List，在每個 Commit 中記下他的 Parent。
- 而 Git 允許 Commit 的分岔和合併，因此可以進行複雜的樹狀操作。



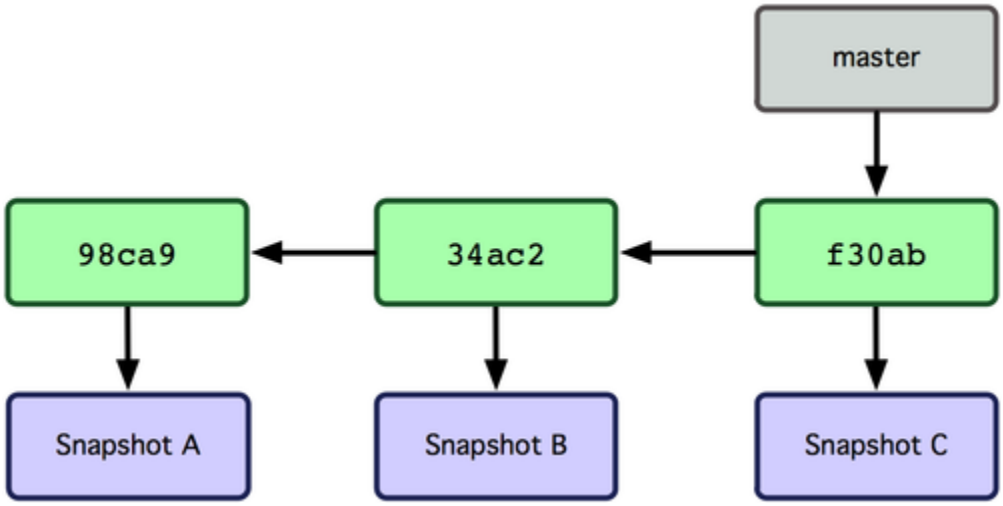
Tag 與 Branch

在 Git 中，Tag 和 Branch 是指向某一特定 Commit 的指標：

- Branch 是會動的：使用中的 Branch，指向的位置會在新 Commit 出現後自動移動。每一個 Repository 都會有一個預設的 Branch 叫 `master`。
  - Git 會使用 HEAD 指標去紀錄目前所使用的 Branch。
  - 可以隨時新增、切換和刪除 Branch（因為它只是指標！）
  - 分岔的 Branch 可以透過 Merge 和 Rebase 去合併。

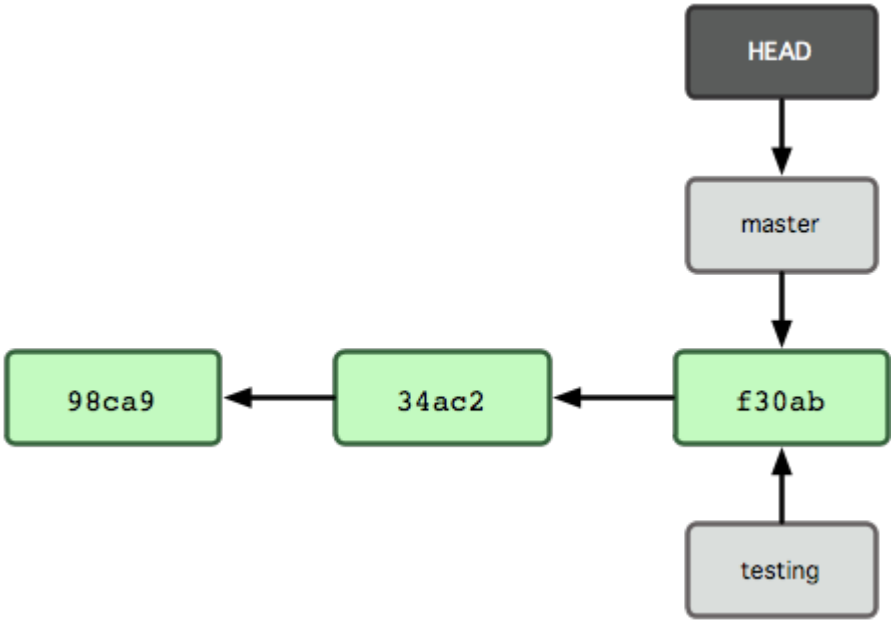
- Tag 是不會動的：對某一個特定的 Commit 加上標記，指向位置不會隨新 Commit 出現更動。用於標記釋出版本或里程碑。

實例一：我在哪裡？



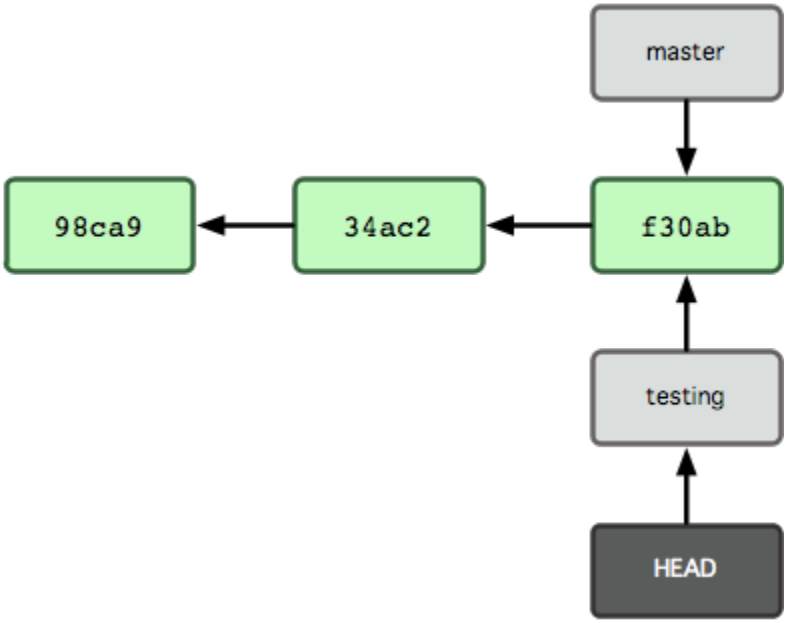
一個擁有三個 Commit 的 Repository。這時只有一個預設的 `master` Branch。

實例一：我在哪裡？



`git branch testing`
建立一個新的 Branch 叫 `testing`，指向目前所在的 Commit。

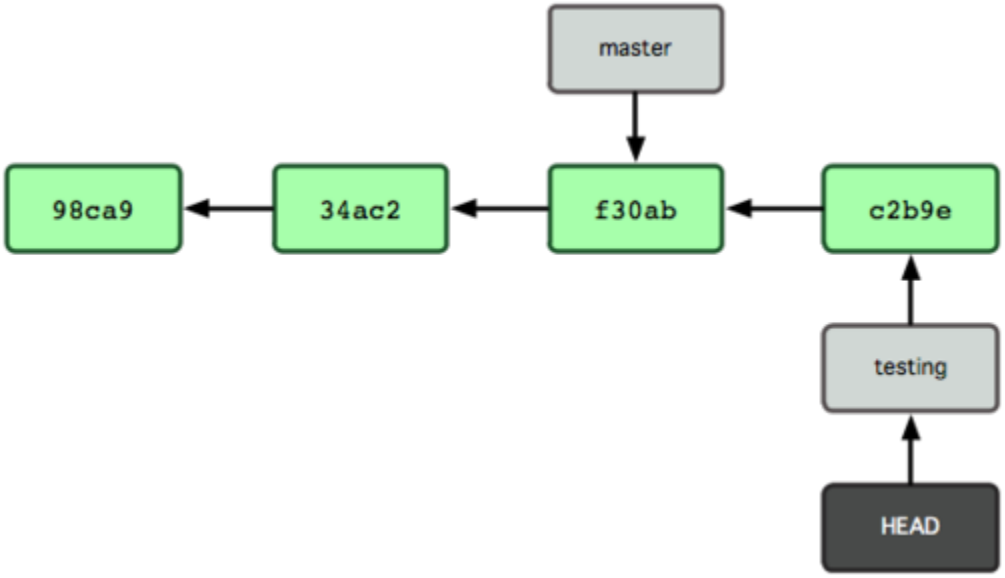
實例一：我在哪裡？



`git checkout testing`
切換到 `testing` Branch：HEAD 會變成指向 `testing`，然後切換到該 Branch 指向的 Commit。

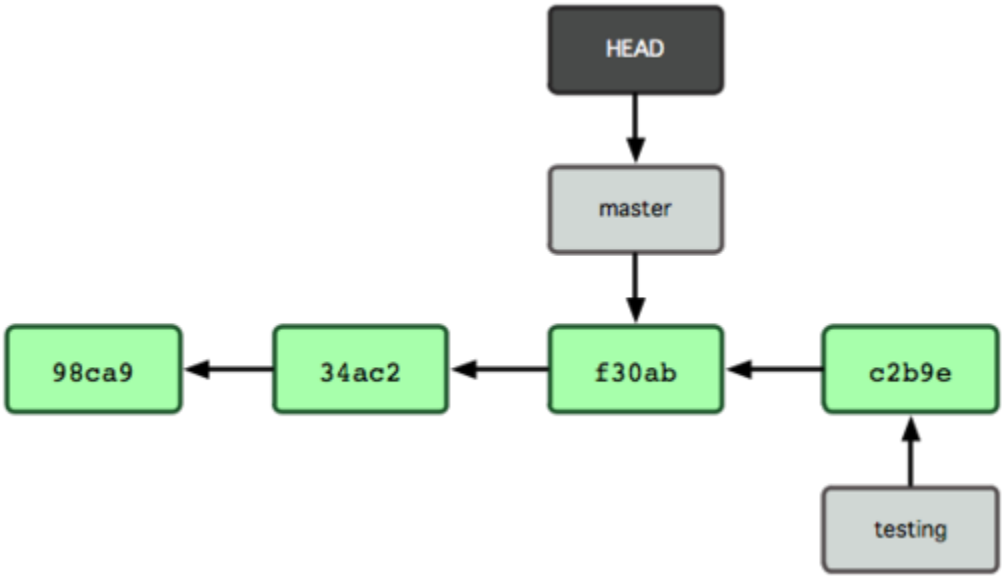
小秘訣：`git checkout -b testing` 可以一次做完開新 Branch + 切換

實例一：我在哪裡？



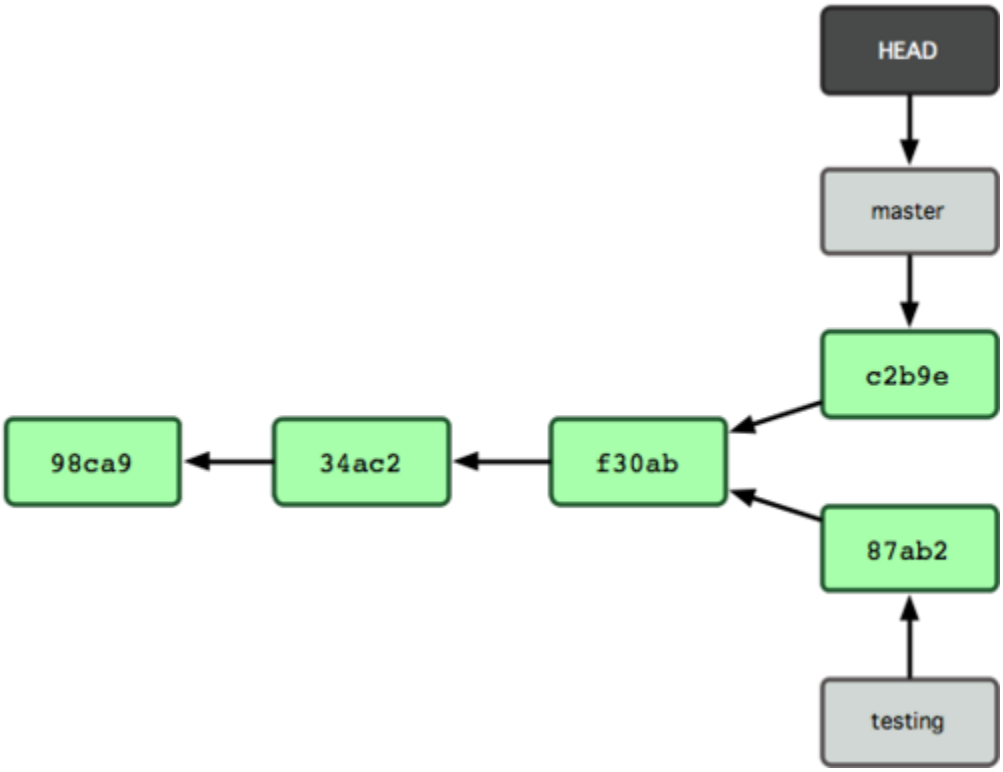
如果在這時新增新的 Commit，會移動 testing Branch 指向的位置，而不會移動 master。
（也就是只有 HEAD 指到的 Branch 會動）

實例一：我在哪裡？



用 `git checkout master` 切換到 `master` 和其對應的 Commit

實例一：我在哪裡？



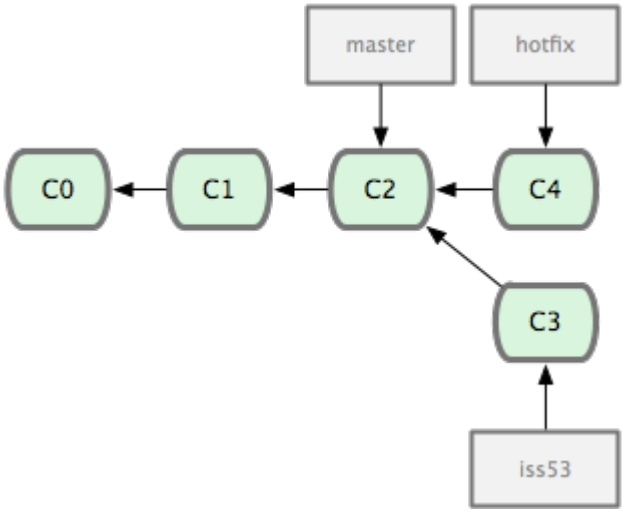
在 master 下進行 Commit 後，由於同一個 Commit 有兩個分支，產生分岔！

Branch 岔開之後呢？

- 可以透過 Merge「合起來」
  - 假如 Merge 的目標只是這個 Branch 往後推的幾個 Commit，只要把 Branch 往後推就好，這樣叫 Fast-forward merge。
  - 無法使用 Fast forward 的狀況，Git 會嘗試用策略去解決並合併分支。簡單、問題較少，但合併次數一多，會讓樹看起來很亂 :P。

- 或透過 Rebase「接上去」
  複雜，容易產生衍生問題，但結果較漂亮
  已經 Push 給別人的東西千萬不要用 Rebase！

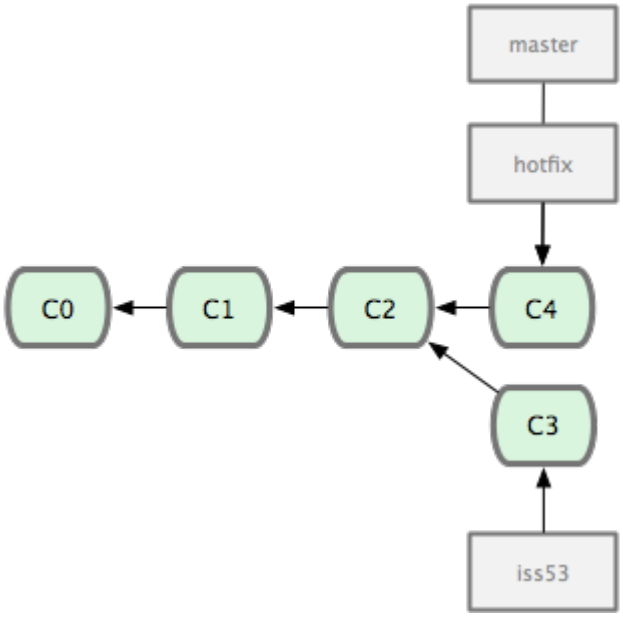
實例二：Merge



- `iss53` 用來修正 Issue 53，但還沒寫完
- `hotfix` 是為了解決現在 `master` 上迫切的問題
  已經寫好了，我想合併進 `master`

實例二：Merge

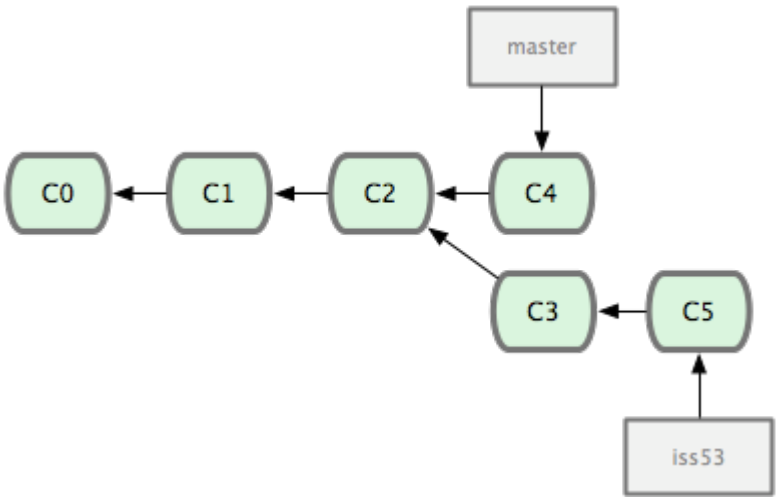
```
git checkout master
git merge hotfix
```



- 切回 `master` 後，「把 `hotfix` 裡的變更合併進來」
- 因為合併的對象是後面幾個 Commit，可以透過 Fast Forward 把 Branch 往後推解決！ :)
- 完成後，我就可以把 `git branch -d hotfix` 把不需要的 Branch 刪掉了

實例

二：Merge

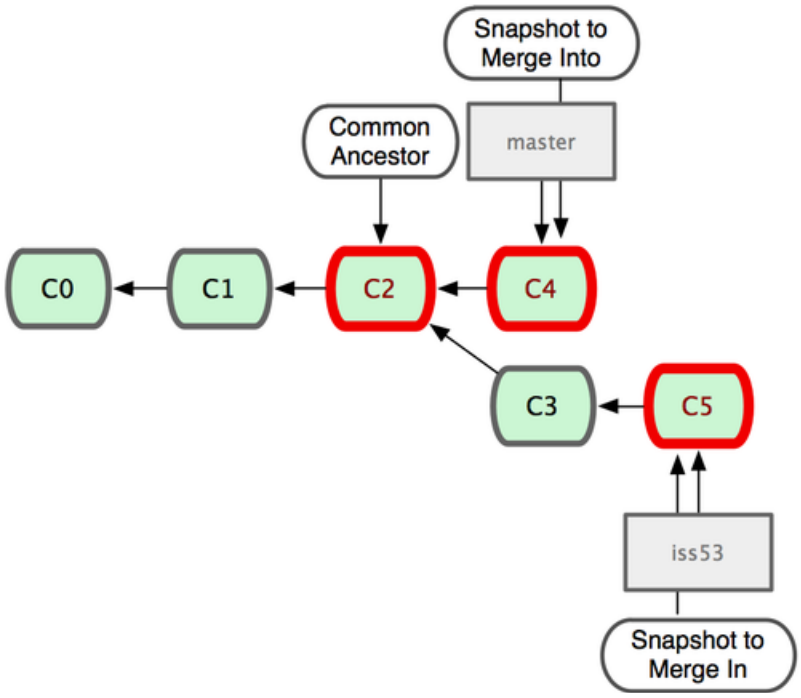


- 這時我把 iss53 寫好了，要 Merge 回去
- `git checkout master`
  `git merge iss53`

- 但，這種 Merge 沒辦法透過 Fast Forward 解決…不過所幸它們有共同的祖先，可以做 3-way Merge。

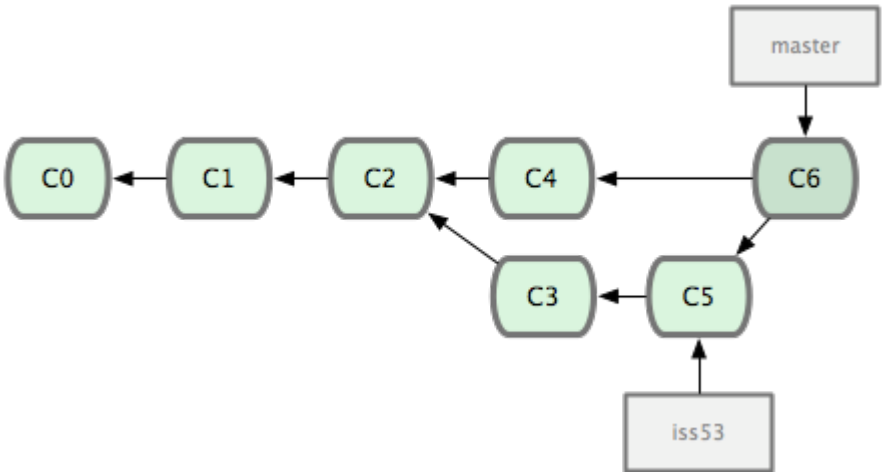
實例二：Merge

- 3-way Merge 的實作：比較祖先和合併對象間（C2 與 C4 間和 C2 與 C5 間）的變動，將這些變動合併起來



實例

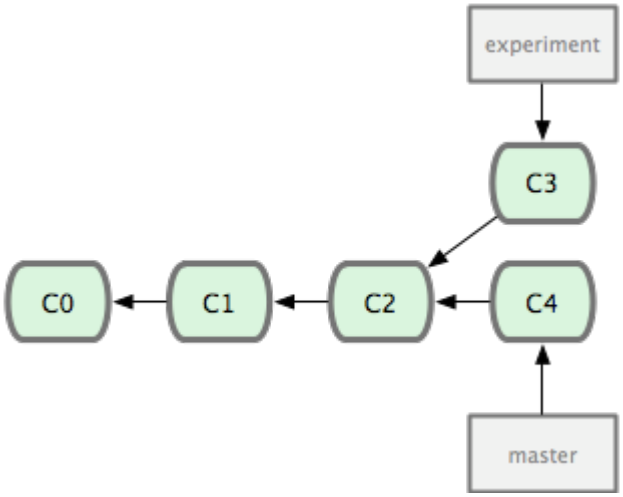
二：Merge



合併的結果會產生新的 Commit C6，C4 和 C5 是其共同的 parent。至此完成分支的合併。

實例

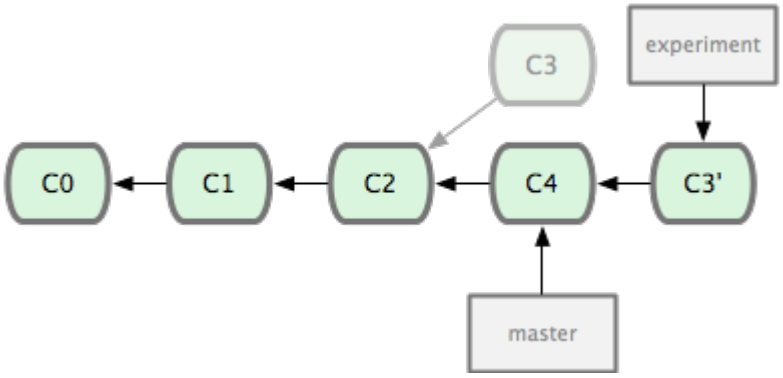
三：Rebase



我想把 experiment 的變更丟進去 master，但又不想用 Merge 製造額外的分支，要怎麼辦？

實例

三：Rebase




```
git checkout experiment
git rebase master
```

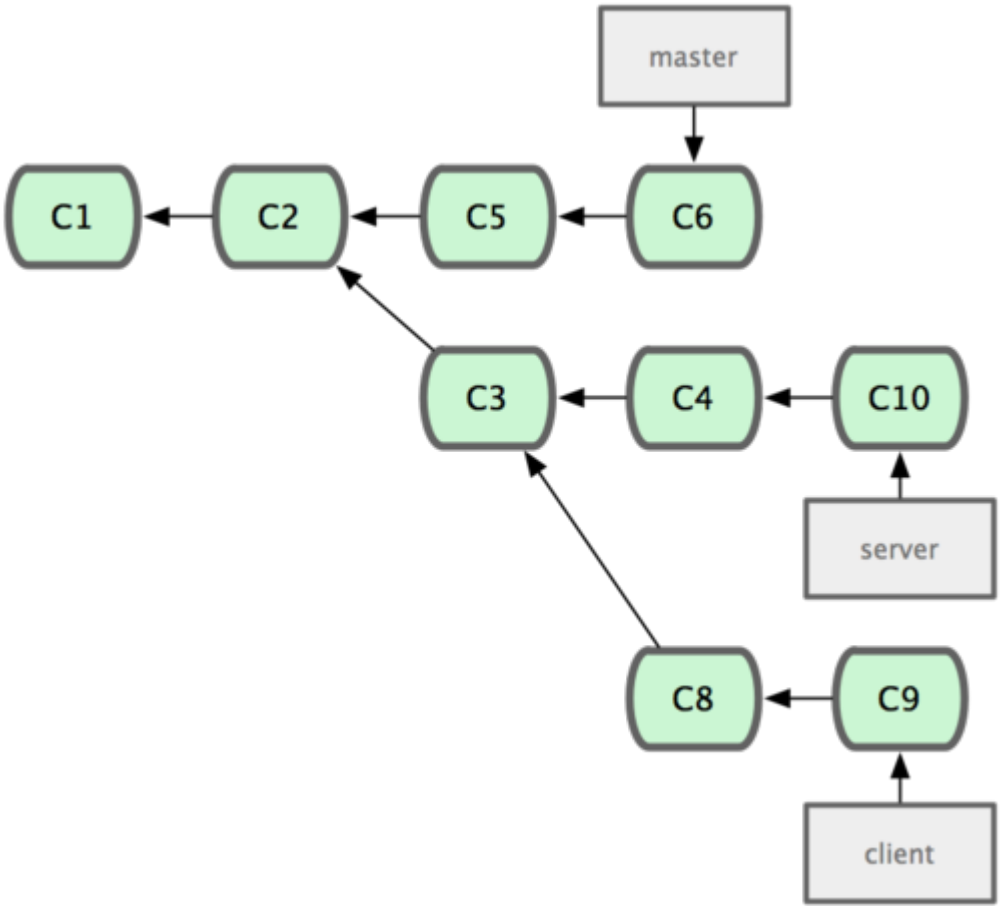
把 C3 的變更「蓋到 master 裡面去」

接下來就只要切回 master，把 experiment 合併好就搞定了 :)

為什麼「已經 **Push** 給別人的東西不能 **Rebase**」？

Rebase 過的東西跟還沒 Rebase 時的東西可能混在一起，整個亂掉…

實例四：複雜一點的 Rebase

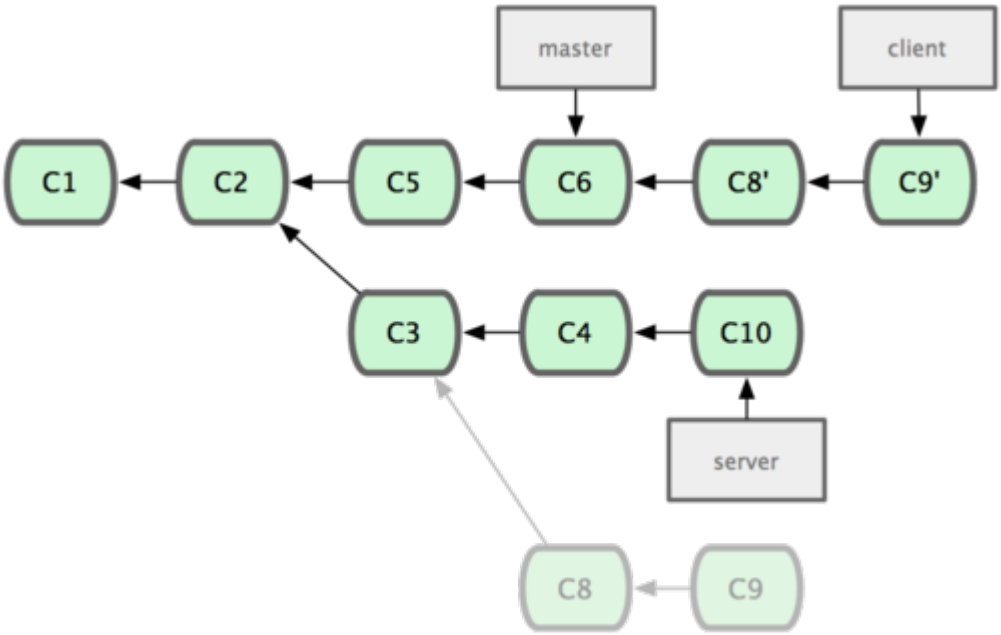


因為 server 寫得爛爛的，我想把 client 的變更蓋到 master 上而非 server 上…

實例四：複雜一點的 Rebase

辦得到！

```
git rebase --onto master server client
```



- 檢查 client，由 client 和 server 的共同祖先和其比較看作了哪些變動，把那些變更重新在 master 上面作過一遍
- 一樣，剩下的就是 Merge 而已了

Conflict!!

當在 3-Way Merge 時發現雙方更動同樣一塊內容時，就會發生衝突。

- Git 遇到衝突時會停下來，把未衝突到的地方給 Staging 起來，把衝突到的地方標示起來。
- 這時可以透過 `git status` 找出衝突的檔案（unmerged）
- 打開那些檔案，手動把衝突之處修正後，重新 Commit 就能解決衝突。

Branch 的其他指令

- `git branch`: 顯示目前所有的 Branch，會用星號標示目前所在者
  以 `git branch -v`同時顯示各 Branch 所在的 Commit
- `git branch --merged`: 顯示已合併到目前 Branch 的其他 Branch
  這樣可以把沒用到的 Branch 找出來刪掉…
- `git branch --no-merged`: 顯示未合併到目前 Branch 的其他 Branch

Tag

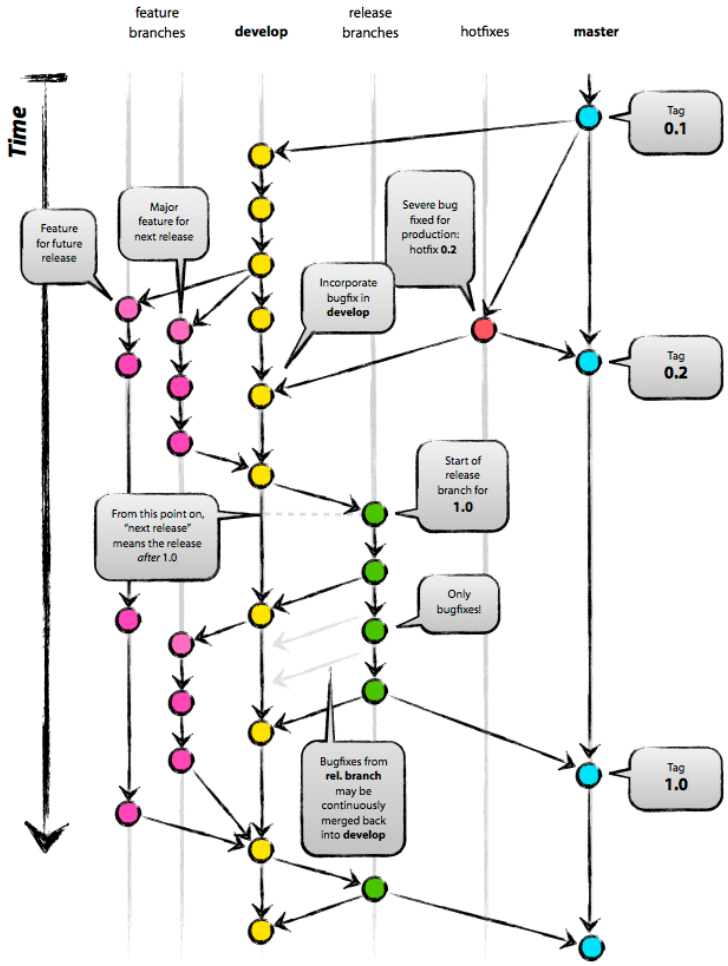
- `git tag v1.0`: 把現在的 Commit 加上名為 `v1.0` 的 Tag
- `git tag v1.0 abcde`: 對代碼為 `abcde` 的 Commit 加上 `v1.0` Tag
- `git tag -d v1.0`: 刪除名為 `v1.0` 的 Tag

Git 另有功能較為強大的 Annotated Tags，可以在 Tag 上加注作者資訊和認證機制。

p.s. Push 時，用 `git push --tags` 才會把 Tag 給一起 Push 上去

推薦的分支方式

- 「每個 Branch 都分開處理一項事情」
- 將測試中和穩定的版本分開為不同的 Branch
- 將需要很長時間才能完成的獨立項目拆成 Branch
- A successful Git branching model — 文中歸納了 Git 用於軟體開發時的分支最佳實踐。有人基於這個概念設計了 git-flow 工具。



p.s.「我想要乾淨的樹」

在進行 Branch 切換，或著 Rebase 的時候，會要求您的 Working Directory 必須是乾淨（沒有變動）的，但改到一半的東西又放棄不了？該怎麼辦？

→使用 Stashing 把變動放到外太空

- `git stash`：把目前的變更（包含 Staging 的部份）放進新的 Stash
  Stash 可以有很多個，會以堆疊的方式儲存（想成放進桶子裡）
- `git stash apply`：把最新的 Stash 中的變更取出。
- `git stash list`：檢視目前的 Stash 堆疊。

Part 5

Remote + Collaboration

介紹透過 Remote 連到其他的 Git 遠端，並示範 Git 協作的具體做法。

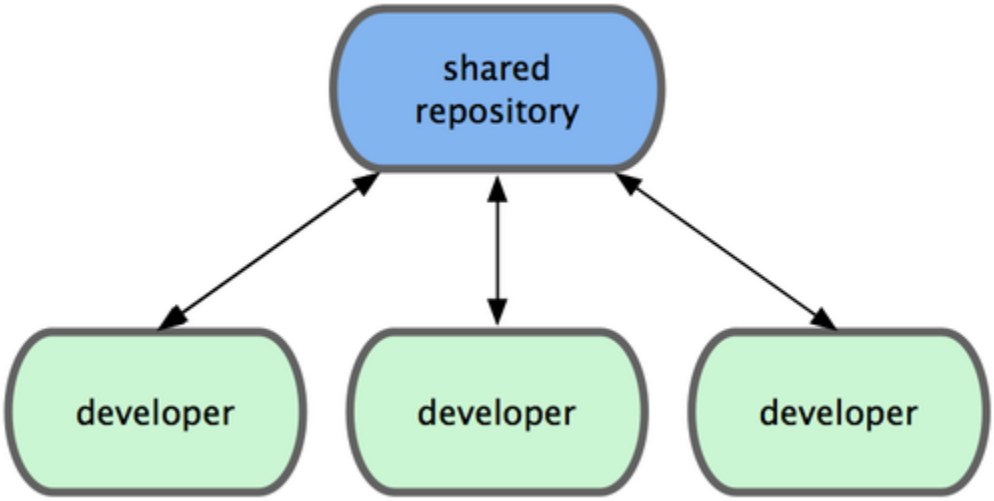
Remote（遠端）

- 讓您的 Repository 同步到其他的伺服器中
- 當多人協作時，需要藉此整合大家的東西
- Git 允許附加多個遠端，因此可以做複雜的同步工作

以下將先介紹多人協作的架構方式與 Git 伺服器的選擇後，介紹 Remote 的使用方式。

多人協作：單一中心式

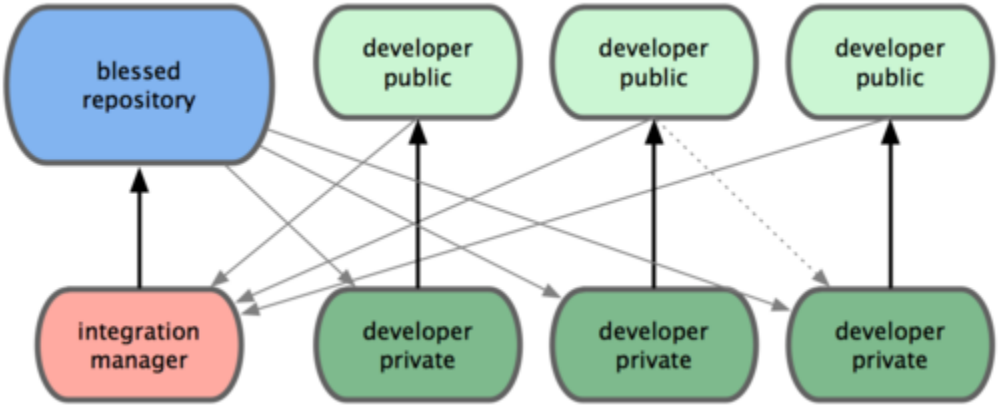
- 類似中心式版本控制系統，所有人跟一個共同的遠端 Repository 進行同步。
- 但跟中心式相比，遇到衝突時，可以透過 Merge 和 Rebase 來解決，不會有卡住的情形。
- 適合人數跟規模較小的情況。



多人協作：整合管理員式

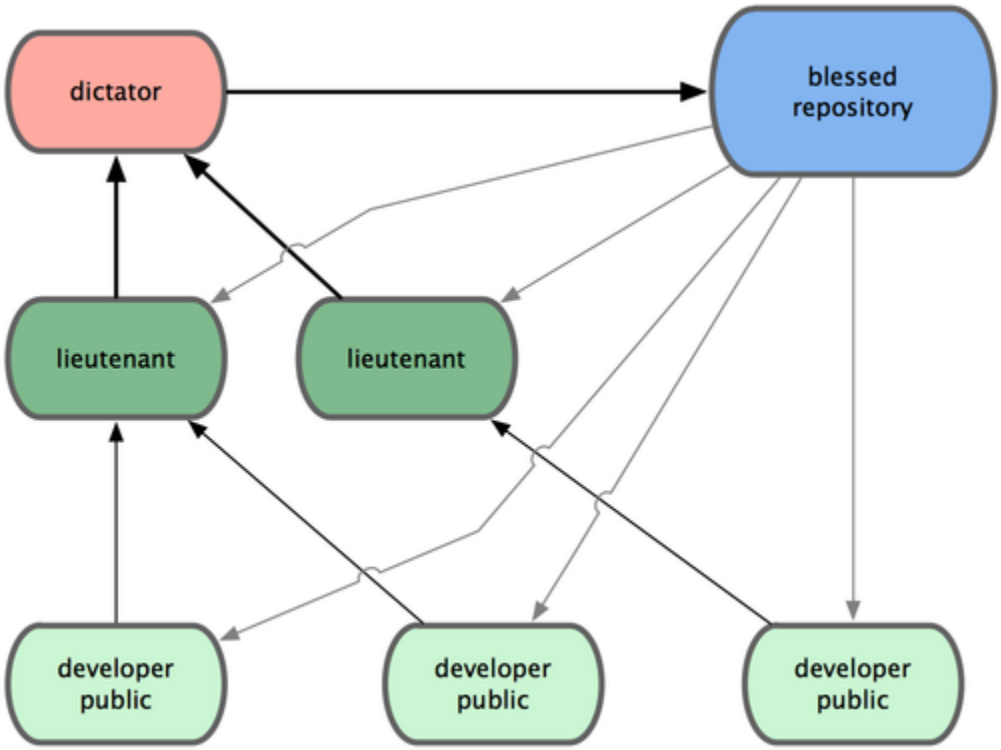
- 有一個「正統」（Blessed）的 Repository 來存放最終版本
- 每個開發者把自己的貢獻整理好後，傳到自己公開的 Repository。
- 由整合管理員決定是否每個開發者改的東西合併到正統裡。

這種模式就是 GitHub 或 BitBucket 等系統的 Fork 與 Pull Request 機制：開發者可以把正統版 Fork 下來改，改好後送 Pull Request 給整合管理員。



多人協作：司令官與附手

- 當專案規模超大時，就需要兩層甚至以上的組織。
- 每個開發者的成果由附手彙整後，交由司令官作最後整合。



Git 遠端的選擇

- 網路上的現有服務：GitHub、Bitbucket、⋯
- 自行架設：Gitolite、Gitosis、⋯
- 檔案系統（NFS、隨身碟⋯）
  - ~~Dropbox~~（雖然可以，但就失去 Git 的價值了）

GitHub

- 主打社群網路功能的 Git 托管服務
- 具有強大的群組、Fork / Pull Request 和其他附加功能，方便多人協作與溝通
- 公開 / Open Source 的專案免費，私密專案則要購買收費服務

https://github.com/

Bitbucket

- 早期只支援 Mercurial，近期開始支援 Git
- 類似 GitHub，也有 Fork / Pull Request 的機制
- 公開專案和五人以下協作的私密專案免費，另有收費服務

https://bitbucket.org/

自行架設

假如您有一個自己的 Linux 主機，您可以使用 Gitosis 或 Gitolite 架設一個自己的 Remote Server，請參考 Pro Git 上的說明：

- Gitolite（推薦）
- Gitosis

或著，如果您有 NFS 或隨身碟，可以直接透過檔案系統作為 Remote。

Remote Protocol

Git 提供以下四種 Remote 的協定：

- Local：以本機路徑作為 Remote 目標，適用於架設於 NFS 時。
  `/paths/to/remote.git` 或 `file:///opt/git/project.git`
- SSH：連接網路的協定，可讀可寫。可以透過 SSH Key 進行認證。
  `username@server.tld:project.git`
  通常會使用 SSH，因為其方便且安全性佳。

Remote Protocol

Git 提供以下四種 Remote 的協定：

- Git：連接網路的協定，唯讀，但速度很快
  `git://server.tld/project.git`
- HTTP(S)：連接網路的協定，可讀可寫。速度較慢，但適用於公司行號內有防火牆，只有 HTTP(S) Port 對外開放的環境
  `http://server.tld/project.git`

Cloning

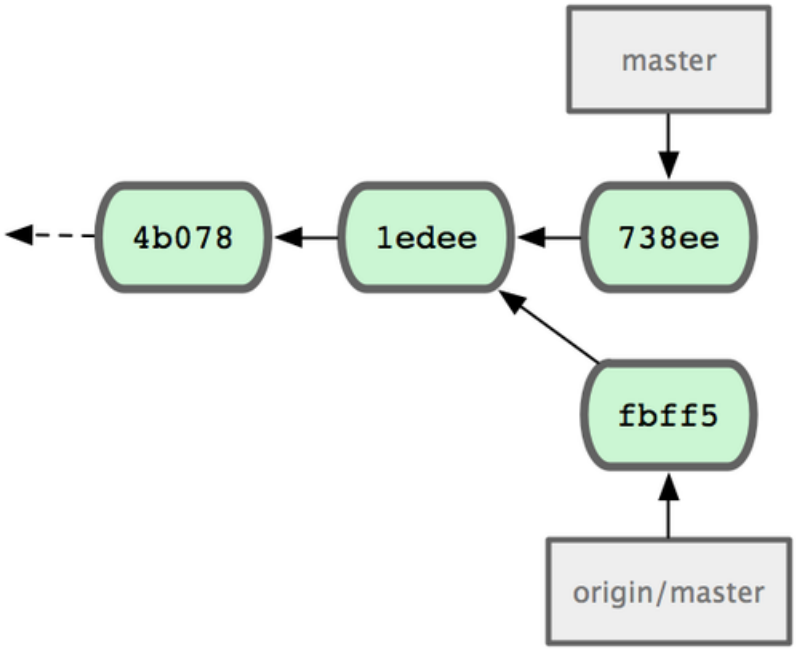
```
git clone git@github.com:littlebtc/my-project.git
```

這行指令會：

- 把程式碼從指定的地方抓下來
- 在抓下來的 Repository 中設定好一個叫做 `origin` 的 Remote 指向該處

Remote Branches

- 在 Remote 中 Repository 的狀態會被記錄下來
- 因此 Remote 中也會有 Branch，前面會加上 Remote 的名字來作識別
- `origin/master` 就是指 `origin` 這個 Remote 中的 `master` Branch



Remote 的相關指令

- `git fetch`：將 Remote 更新到最新的版本。
- `git pull`：進行 Fetch 後，將 Remote Branch 的變動合併進本機的 Branch。
- `git push`：將本機的 Branch 變動合併到 Remote Branch。

實例

在未設定 Remote 的 Repository 中，新增 Remote 後 Push:

- `git remote add origin git@github.com:littlebtc/my-project.git`
- `git push -u origin master`

其他 Git 的教學資源

- Pro Git
- Git Ready
- ihower 的 Git 心得文