

Git 版本控制

資工系 101502520 黃瑞安
2014/12/20



為何需要版本控制？

自己一個人開發專案的時候，是不是...

- 常常東改改西改改，然後自己都忘了改了什麼，程式壞掉不知道怎麼辦？
- 常常在大幅修改前，先複製現在的做備份，然後搞得一堆備份版本又不知道哪個是哪個？
- 最後自己都不知道在搞什麼，搞得一團亂

多人共同開發專案的時候，是不是...

- 常常大家各自寫各自的，不知道怎麼整合？
- 常常整合卻又不小心把舊的版本整合進來？
- 最後大家又是搞得一團亂

版本控制怎麼解決這些問題？

Git 版本控制就像是一種高級的存檔機制。

開發到一個段落，就可以如存檔一般，提交這次的檔案變動，並附上提交訊息。

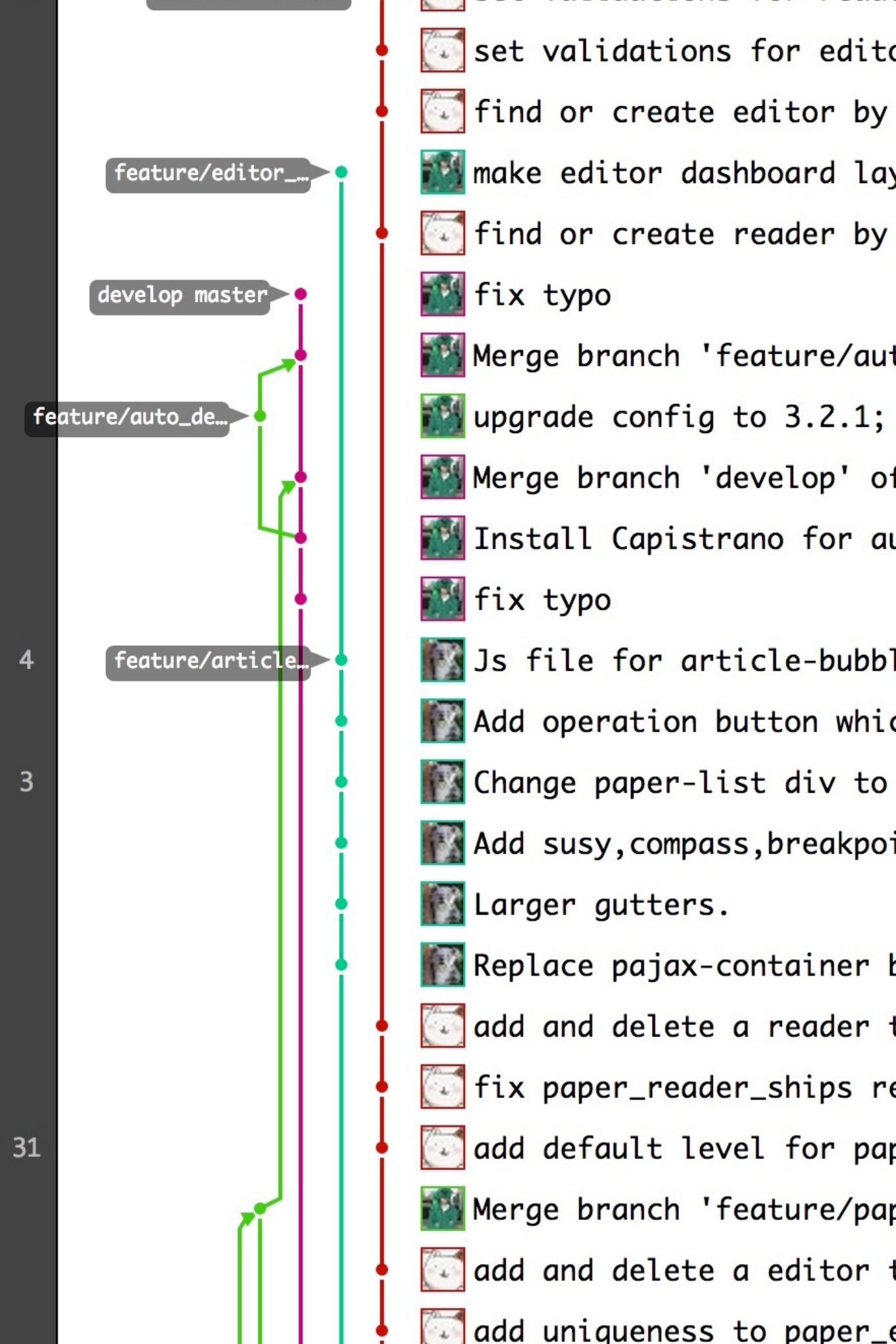
透過 Git，我們還可以十分清楚專案整體、各個分支的發展狀況

例如：

哪個分支合併了哪個分支
哪個夥伴最近做了什麼事情

除此之外，透過這個提交機制，Git 可以：

比較檔案變動的差異
回溯到先前的版本
創立新的分支，並輕鬆合併



在開始使用 Git 之前，須先設定

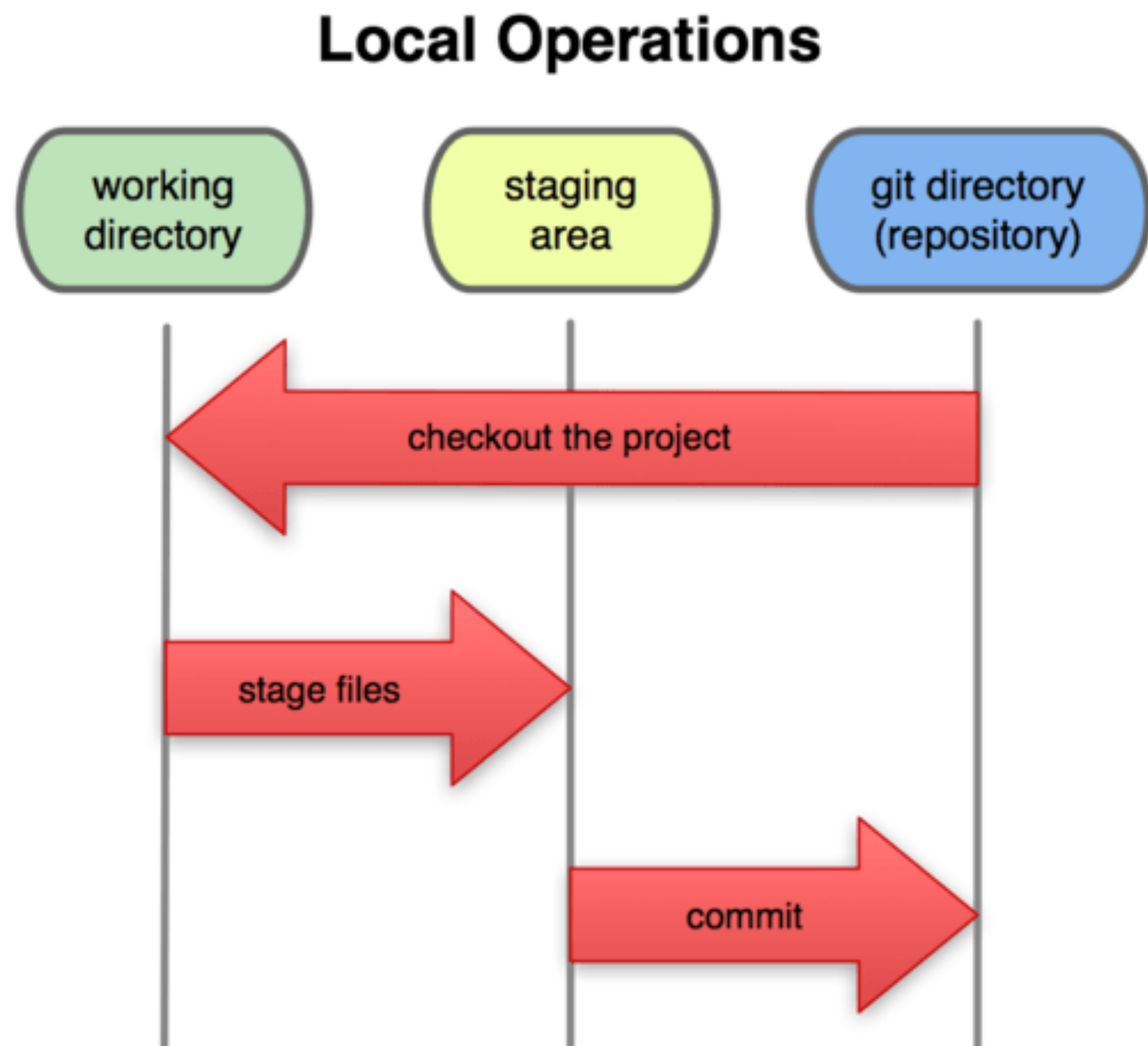
- 首先必須先設定提交要用的識別資料
 - `git config --global user.name "Your Name"`
 - `git config --global user.email "your@email.com"`
- 可以使用 `git config --list` 查看目前設定值
- 詳細請參考 Pro Git - 初次設定 Git

開始使用 Git

- 在專案的根目錄下執行 `git init` 初始化 git repository (建立 `.git` 資料夾)
- 在 `.git` 資料夾裡面有一些指標性的純文字檔，分別是：
 - `.git/HEAD`
 - `.git/refs/`
- `.git/HEAD` 是用來判斷現在是在哪個分支，或是在哪個提交上。
- `.git/refs` 資料夾裡面的檔案都是儲存分支或 tag 目前指向哪個提交。
如 `.git/refs/heads/master` 是本地的 master 分支目前指向的提交的 SHA。
- 理解這些指標對於操作 git 很有幫助，特別是 `checkout`、`reset` 指令。

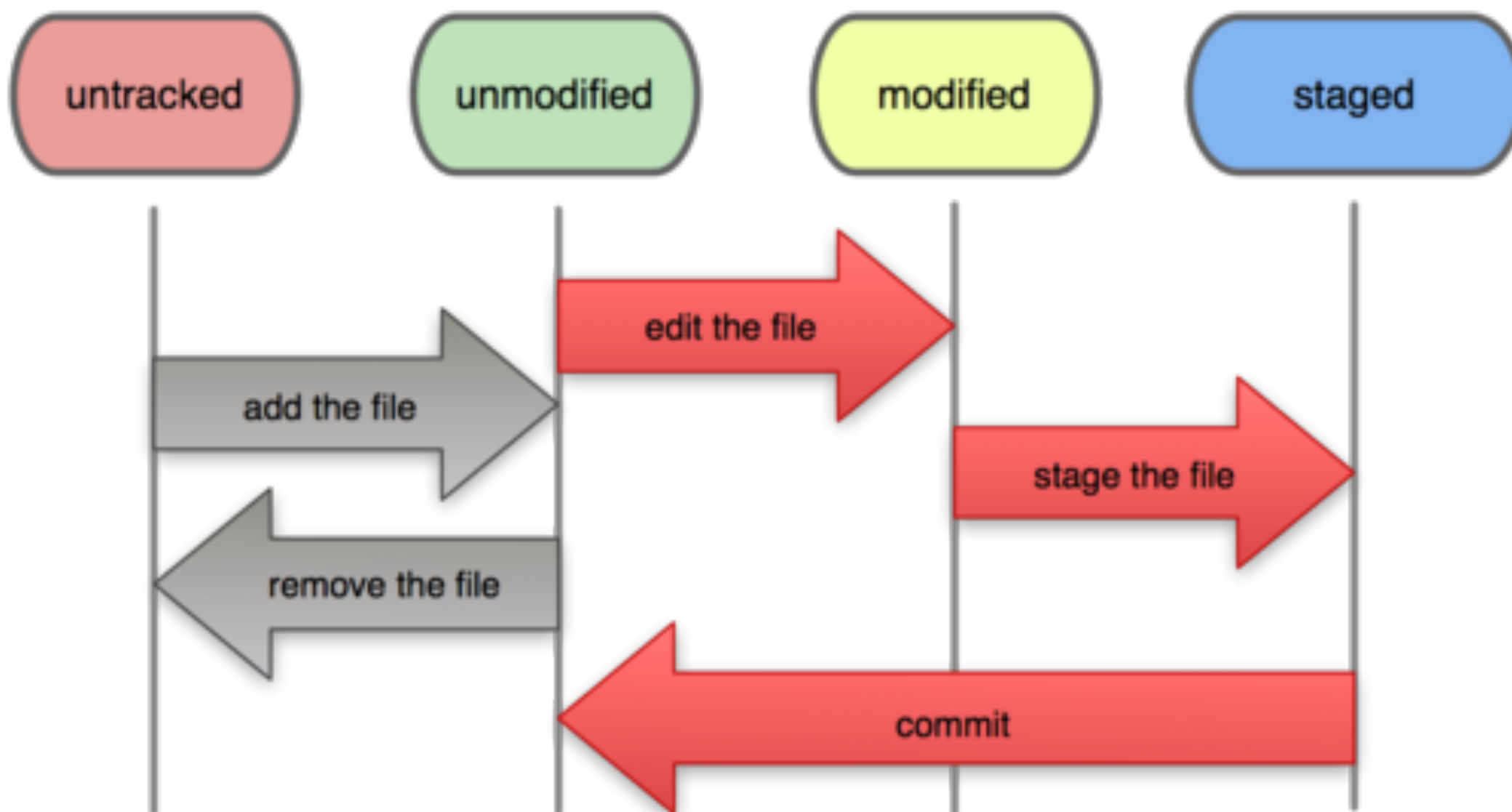
Git的三個操作區域

- working directory 就是目前工作的環境，也就是 .git 資料夾之外，我們直接用編輯器或 IDE 直接操作的區域。
- git directory(repository) 就是 .git 資料夾，git 會將所有的記錄都放在裡面。
- staging area 則是提供給我們方便操作的過渡區域。git 以一個 commit 提交作為記錄單位。你可以選擇哪些檔案要加入這次的提交，就把那些檔案先加入 staging area，之後就可以一併提交。
- 而已經提交到 git repository 的記錄可以隨時取回到 working directory



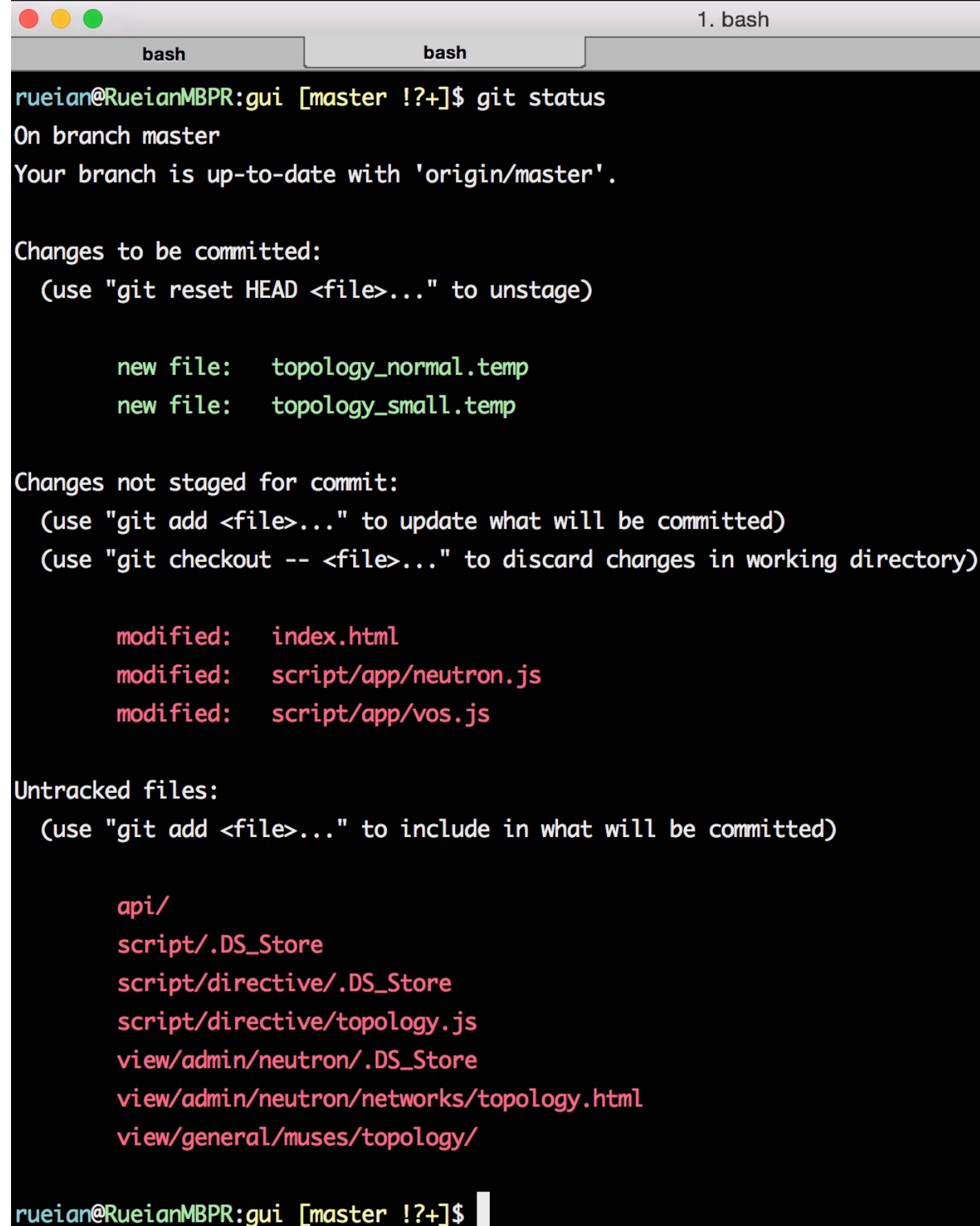
Git 的檔案狀態

File Status Lifecycle



查看狀態 git status

- 使用 **git status** 指令可以觀看目前狀態
- 以右圖為例：
 - 目前所在地分支為 master
 - 目前本地master與遠端origin/master記錄相符
 - 由上而下三個檔案列表區塊分別是：
 - 在staging area等待commit的檔案
 - 已經被修改但還沒被加入staging area的檔案
 - 還沒有被git追蹤的檔案
- **git status -h** 查看其他用法



```
rueian@RueianMBPR:gui [master !?+] $ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    new file:   topology_normal.temp
    new file:   topology_small.temp

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

    modified:   index.html
    modified:   script/app/neutron.js
    modified:   script/app/vos.js

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

    api/
    script/.DS_Store
    script/directive/.DS_Store
    script/directive/topology.js
    view/admin/neutron/.DS_Store
    view/admin/neutron/networks/topology.html
    view/general/muses/topology/

rueian@RueianMBPR:gui [master !?+] $
```

提交記錄 git add & git commit

- 將想要加入提交的檔案(一個或多個)，先加入 staging area
 - `git add <file/directory> <file/directory> ...` #加入檔案或資料夾
 - `git add -u` #加入所有已追蹤且有變動的檔案
 - `git add -A` #加入所有已追蹤且有變動與未追蹤的檔案
- 再將 staging area 內的檔案加入 commit 提交
 - `git commit`
 - `git commit -m "commit message"`

移除檔案 git rm

- 若想將已被 git 追蹤的檔案從 git repository 中移除可用 `git rm` :
- `git rm <file>` #將該檔案從 working directory刪除，並將刪除動作加入 staging area，等待提交。
- `git rm <file> --cached` #將刪除動作加入 staging area，等待提交，但不會真的將檔案從 working directory 內刪除。如此可以將某個必要的檔案取消追蹤。

忽略檔案 .gitignore

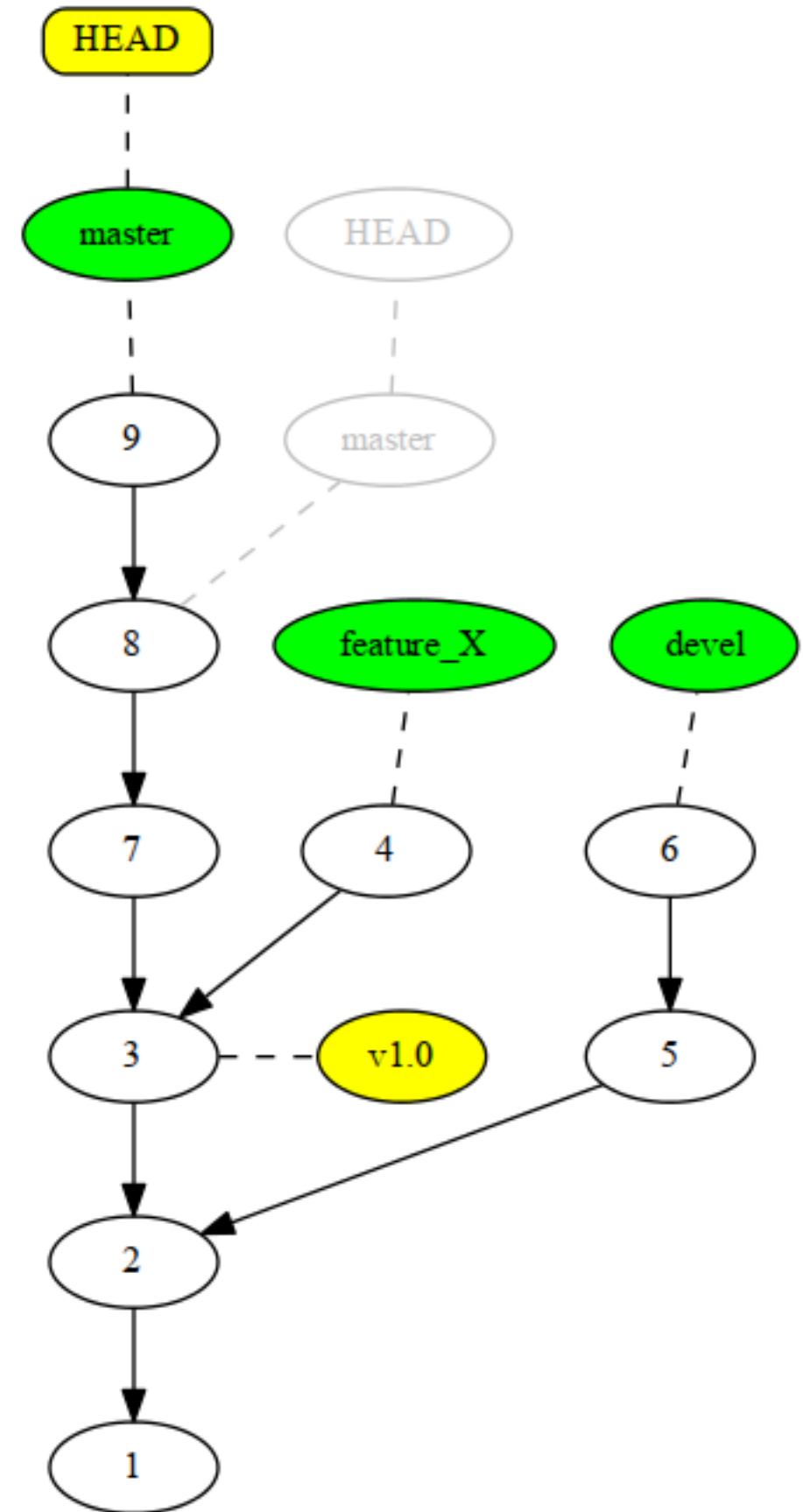
- 有些檔案並不希望被 git 追蹤，例如 IDE 的環境設定、含有機密的設定檔如資料庫帳號密碼等檔案，可以透過 .gitignore 檔案來設定。
- 也可以設定全域的忽略規則
 - `git config --global core.excludesfile '~/.gitignore'`
- 詳細請參考 Pro Git - 提交更新到儲存庫#忽略某些檔案

開立分支 git branch

- 在開發新功能的時候，我們可以開立一個新分支，這樣就不會影響到原分支的內容。當新功能開發好了，而且確定沒問題就可以合併回去原分支。
- 使用 `git branch <name>` 從目前的位置創立一個新分支出來，並使用 `git checkout <name>` 切換到該分支。
- 或是直接使用 `git checkout -b <name>`，創立新分支並直接切換過去。
- `git branch -a` 可以列出包含本地與遠端的所有分支
- `git branch -h` 可以看其他用法

建立標籤 git tag

- 標籤有輕量級(lightweight)和含附註(annotated)兩種。
- 輕量級標籤只是指到特定的指標。
- 含附註標籤則是完整 Git 物件(具備作者和日期等訊息)。
- 一般而言，建議使用含附註的標籤以便保留相關訊息。
- `git tag -a <name>` #建立含附註標籤
- `git tag <name>` #建立輕量級標籤
- 由於標籤是固定的，故常用來標記版本。如 v1.0
- 詳細請參考 Pro Git - 標籤



標籤是固定的，而分支是可變動的

查看記錄與變動 git log & git diff

- `git diff` 可以查看目前還沒被加入 staging area 檔案的變動
- `git diff --cached` 可以查看 staging area 的檔案變動
- `git difftool` 用外部差異比對程式查看檔案變動
- `git log` 可以查看該 branch 的提交記錄
- `git log -p` 可以查看該 branch 的提交記錄，包含變動
- `git log --graph` 可以查看 commit tree 圖表
- `git blame <file>` 可以查看單一檔案的變動記錄

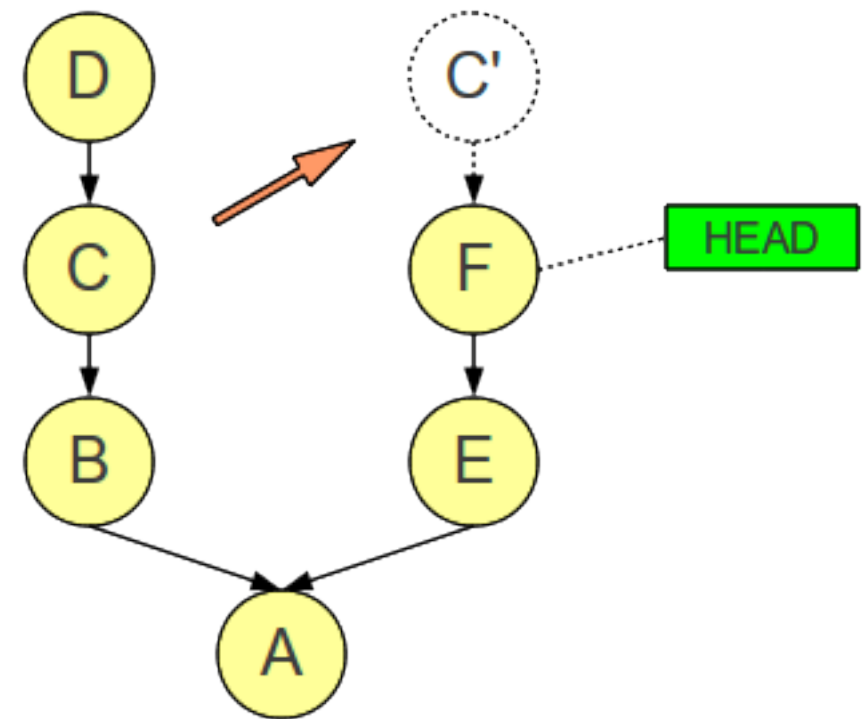
暫存變動 git stash

- 常常你開發到一半，而你想轉到其他分支上進行一些工作。但你不想只為了待會要回到這個工作點，就把做到一半的工作進行提交。這時候可以用 `git stash` 將尚未提交的變動暫存進堆疊。
- `git stash list` 可以查看在堆疊中暫存的變動
- `git stash apply` 可以將堆疊中暫存的變動套用回來
- `git stash drop` 可以將暫存的變動存堆疊中移除
- `git stash pop` 可以套用變動並從堆疊中移除
- 詳細請參考 Pro Git - 儲藏 Stashing

複製提交

git cherry-pick

- `git cherry-pick <SHA> <SAH>...`
- `cherry-pick` 可以複製一個或多個 commit，分別做成新的並套用在目前 HEAD 之後。
- `cherry-pick` 在套用提交時可能會遇到 conflict，將解決完衝突的檔案加入 staging 後可用 `git cherry-pick --continue` 繼續完成套用，或是用 `--abort` 放棄。



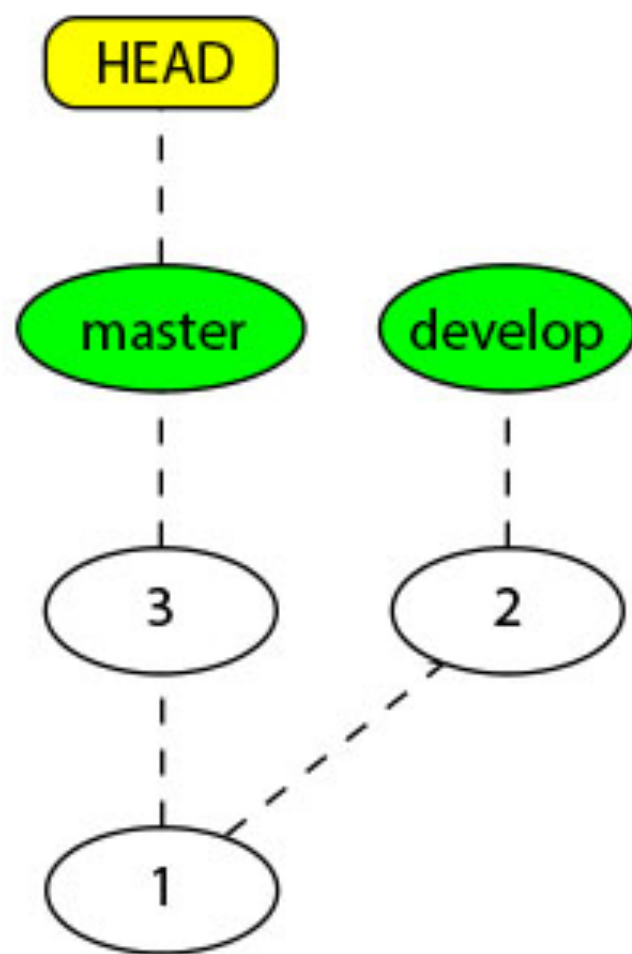
切換分支 git checkout

- `git checkout <branch/SHA>` 可用來將 HEAD (.git/HEAD) 指向某個位置，並將目前 working directory 覆蓋，用來切換分支或在 commit tree 裏面游走。
- 例如 `git checkout master` 後，.git/HEAD 的內容就是 `ref: refs/heads/master`
- `git checkout <SHA>`，將 HEAD 直接指向某個提交，進入 detached HEAD 狀態。這時候所做的提交不屬於任何分支，僅可使用 SHA 存取這些提交，最好用 `git branch` 或 `git checkout -b` 為這些 dangling commit 建立一個新分支。

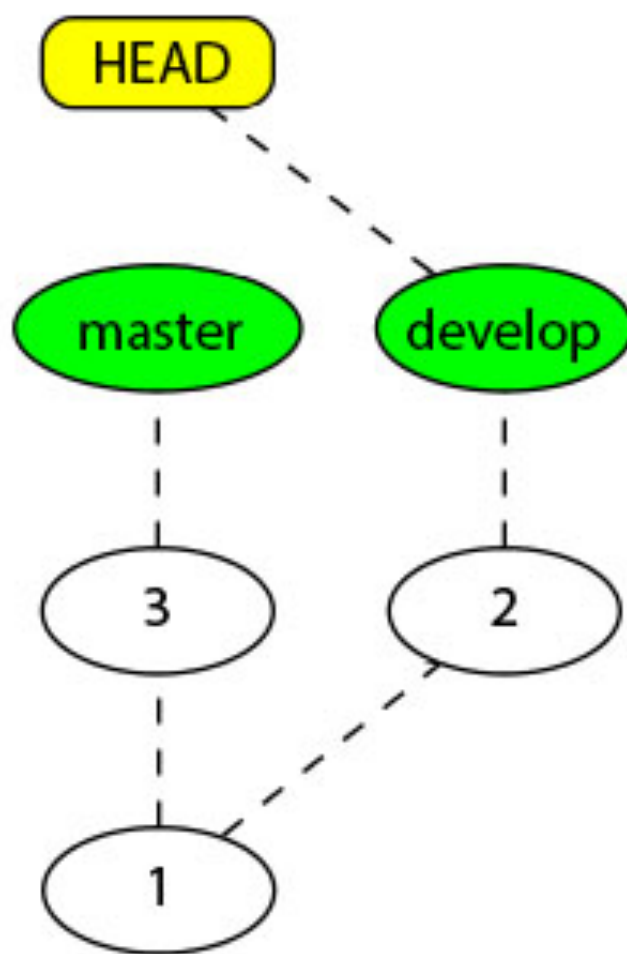
撤銷記錄 git reset

- `git reset <SHA>` 可將目前分支的 HEAD (.git/refs/heads/<branch>) 設定到某個提交。若在 detached HEAD 狀態則是設定 .git/HEAD。
- `git reset HEAD~2` 可將目前分支回到兩個提交之前，也就是撤銷最近兩次的提交，並將這兩個的變動放回 unstaged 狀態，可以修改後再透過 `git add` 加回 staging area。
- 加上 `--hard` 參數則不保留變動，直接覆蓋 working directory。
- `git reset` 不只可以用來撤銷先前的提交，被撤銷的提交也可以用 `reset` 還原。可用 `reflog` 查找已經被撤銷的提交的 SHA，再用 `git reset <SHA> --hard` 即可還原。

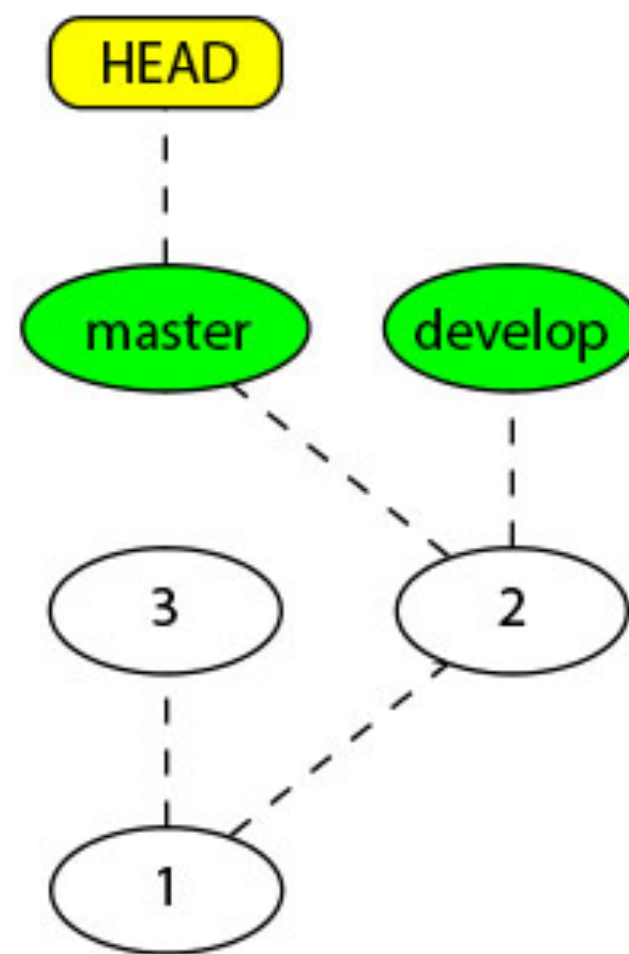
git checkout 與 git reset 的差異



原先



git checkout develop



git reset 2

取消尚未提交的變動 git reset & checkout

- 若提供檔案路徑給 **reset** 或 **checkout**，將不會去改變 HEAD 或分支的 HEAD。可用來將 staging 的檔案取回或將未加入 staging 的檔案的變動取消。如 **git status** 的操作提示：

```
rueian@RueianMBPR:git-test [master !]$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   test.md

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)

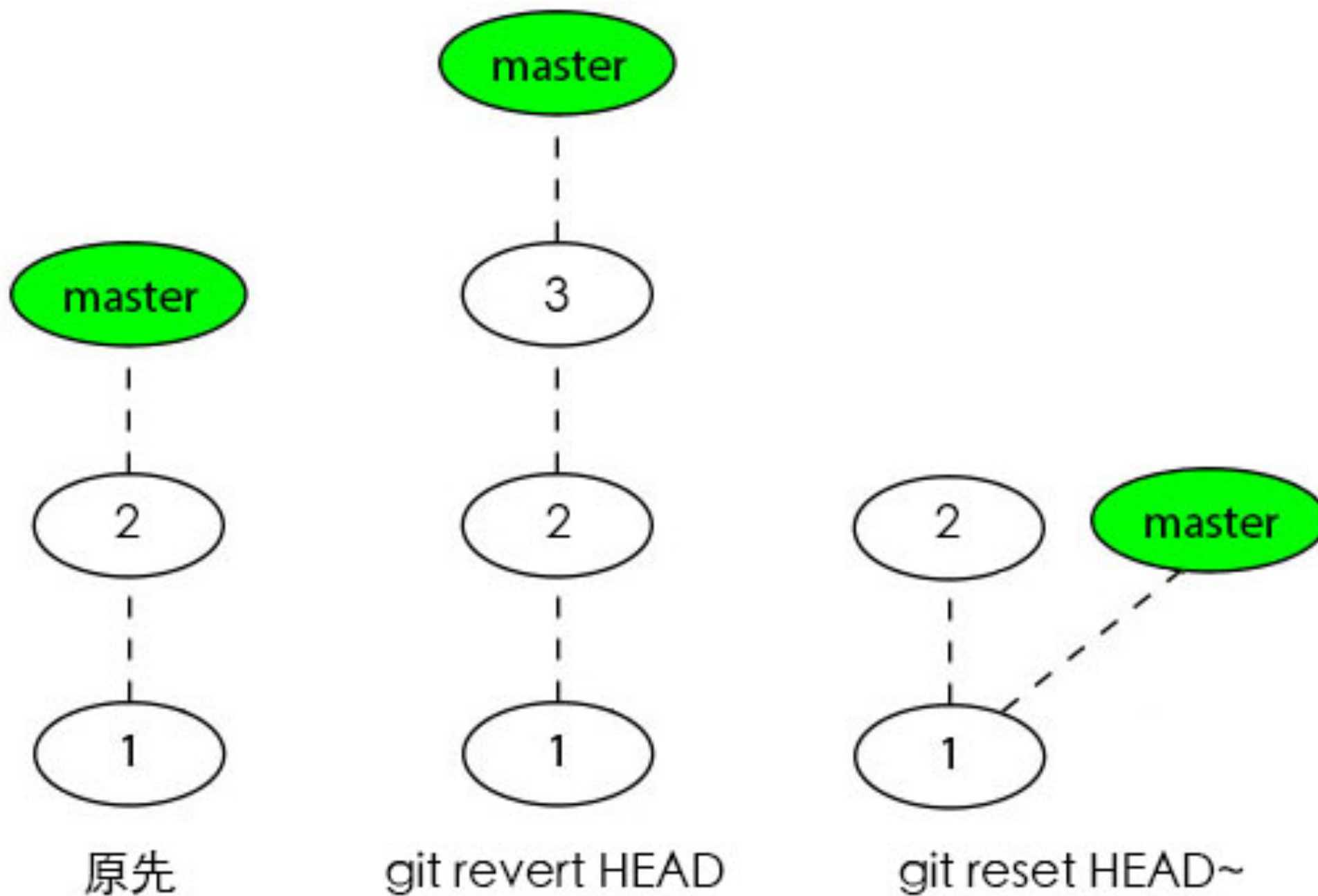
    modified:   README.md

rueian@RueianMBPR:git-test [master !]$
```

回朔記錄 git revert

- `git revert` 可以回朔指定的 commit，並將這個回朔動作做成一個新的提交。例如原本的提交記錄是 A->B->C，在 `git revert B` 之後會變成 A->B->C->D，個新的 D 就是這次的 revert 動作。
- `git revert HEAD~2` 是回朔到最近三個提交之前。
- `git revert` 常常需要處理 conflict，將衝突檔案處理完並用 `git add` 加入 staging 後，可透過 `git revert --continue` 繼續，或是可以用 `git revert --abort` 放棄這次 revert

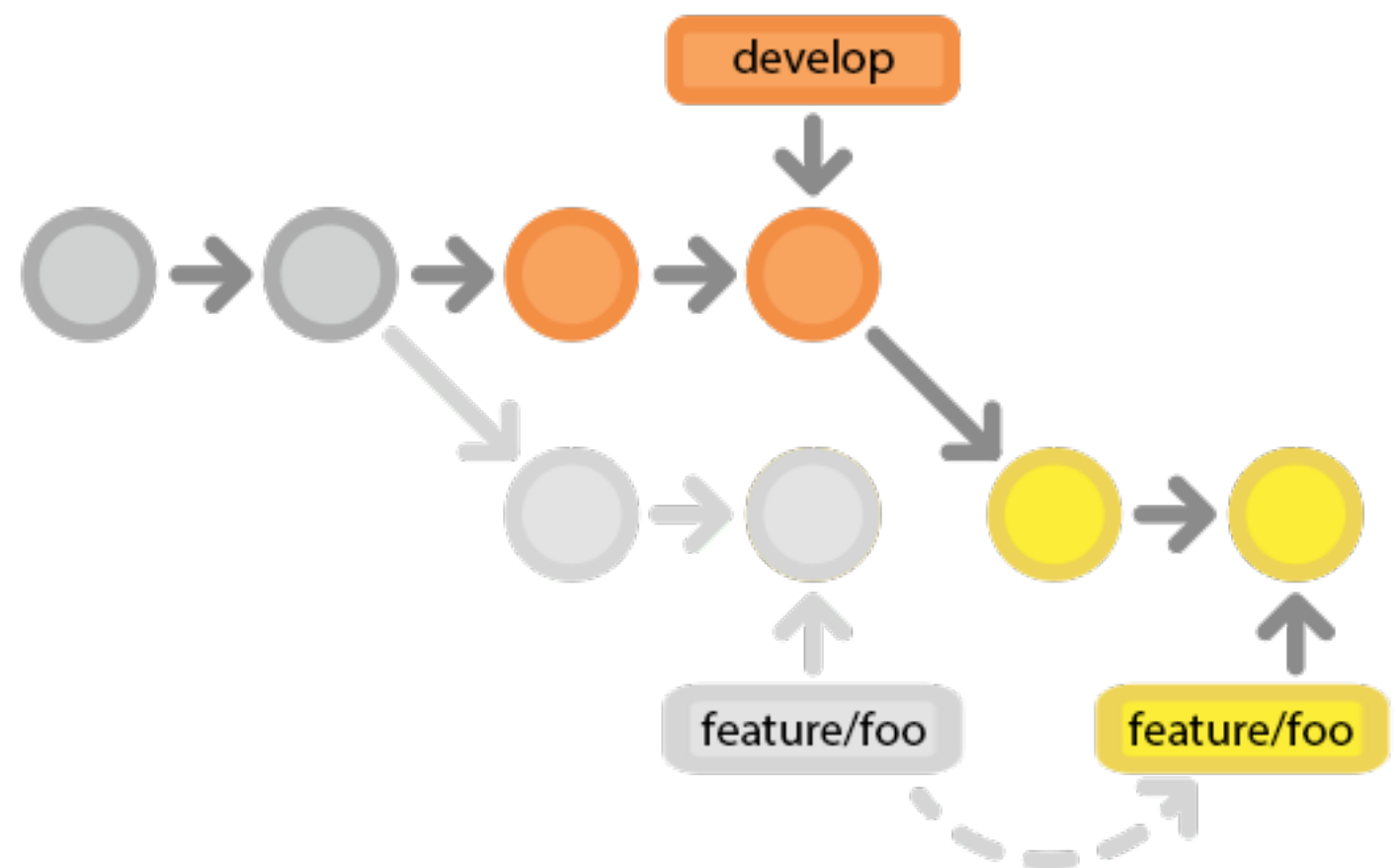
git revert 與 git reset 的差異



重寫歷史 git rebase

- **rebase** 用於重新套用提交。在將新分支合併回原分支之前，若原分支也有更新的話，最好將這個新分支基於原分支 **rebase**，並把衝突在這時解決掉。這樣可避免負責原分支的人在把新分支合併進來時要幫你處理衝突。
- **rebase** 更重要的功能是可以決定如何重新套用這些提交。使用 **git rebase -i HEAD~3** 可以進入互動模式重新套用前三次的提交。在互動模式中，可以做到重排、合併、分拆、移除提交以及修改提交等功能。
- 詳細請參考 Pro Git - 重寫歷史

git rebase 示範圖



改寫最近一次提交 `git commit --amend`

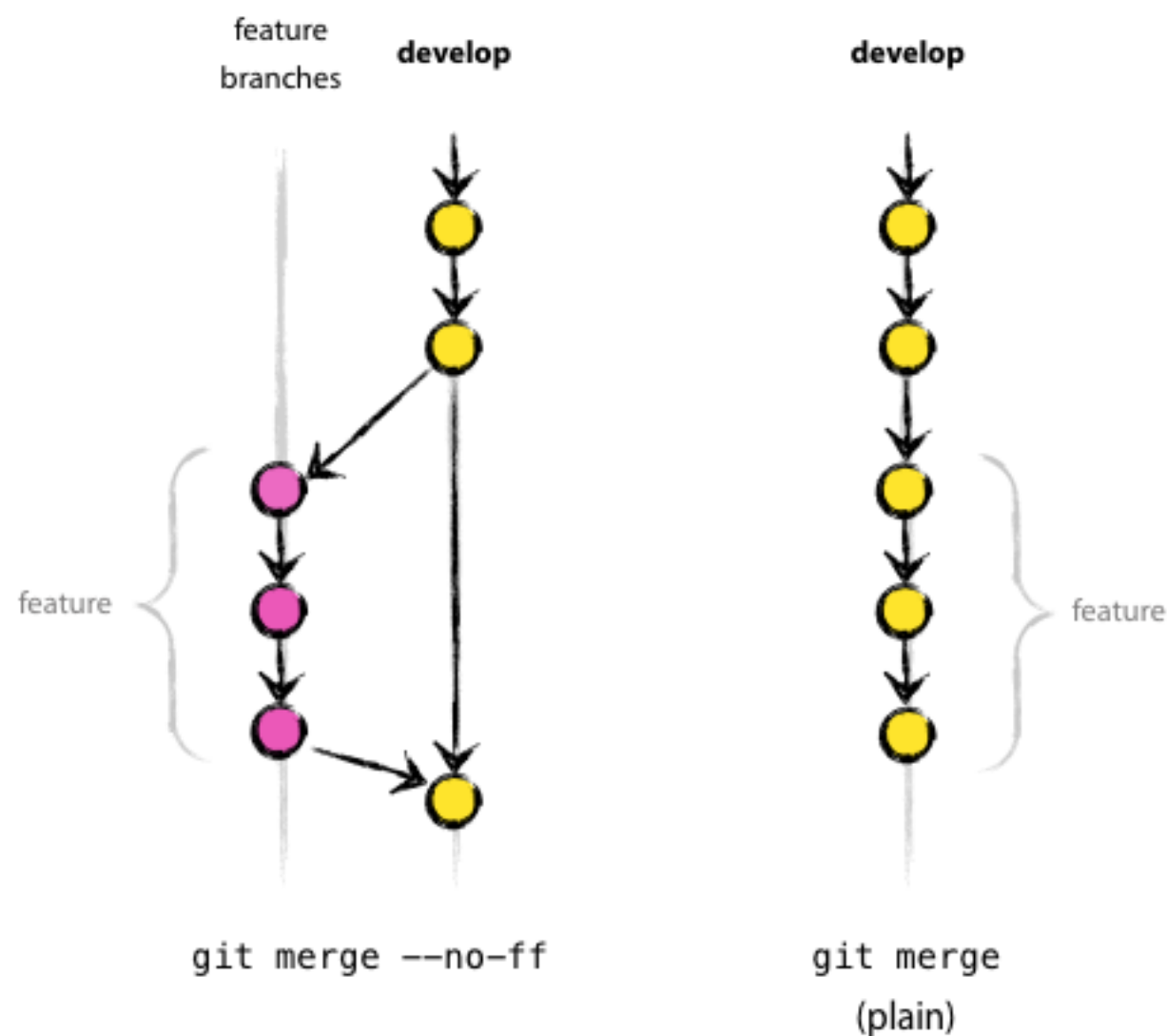
- 除了透過 `git rebase -i HEAD~`，用 `git commit --amend` 也可以改寫最近一次的提交記錄。
- `git commit --amend` 會將 staging area 和最近一次提交合併做成一個新的提交接在上一個提交之後。若只是要改寫提交訊息，則 staging area 裡面不需要有東西。
- 注意，`git rebase` 以及 `git commit --amend` 所產生的提交都會有新的 SHA 值，視為與先前完全不同的提交。若已經將先前提提交推上遠端倉庫的話，最好不要再改寫歷史，以免造成其他開發者的困擾。

利用 git reflog & git fsck 復原記錄

- 當使用 `reset`, `rebase`, `commit --amend` 等指令，改變了提交記錄之後，卻後悔了，怎樣可以復原呢？任何在 Git 提交的變動幾乎都是可復原的。
- git 會在每次 HEAD 改變的時候留下紀錄，可以使用 `git reflog` 或 `git log -g` 查看 HEAD 變動記錄。
- 我們可以透過 `git reflog` 查到想要回復到的提交的 SHA，再使用 `git reset --hard <SHA>` 回到該提交或用 `git merge <SHA>` 合併進來，甚至可使用 `git branch <name> <SHA>` 開立一個與改變記錄前相同新分支。
- 若在 `git reflog` 裡面找不到想要還原的提交的 SHA，可以使用 `git fsck --full` 查找未被指向的 dangling commit 的 SHA，透過相同的方法一樣可以還原。
- 詳細請參考 Pro Git - 維護及資料復原

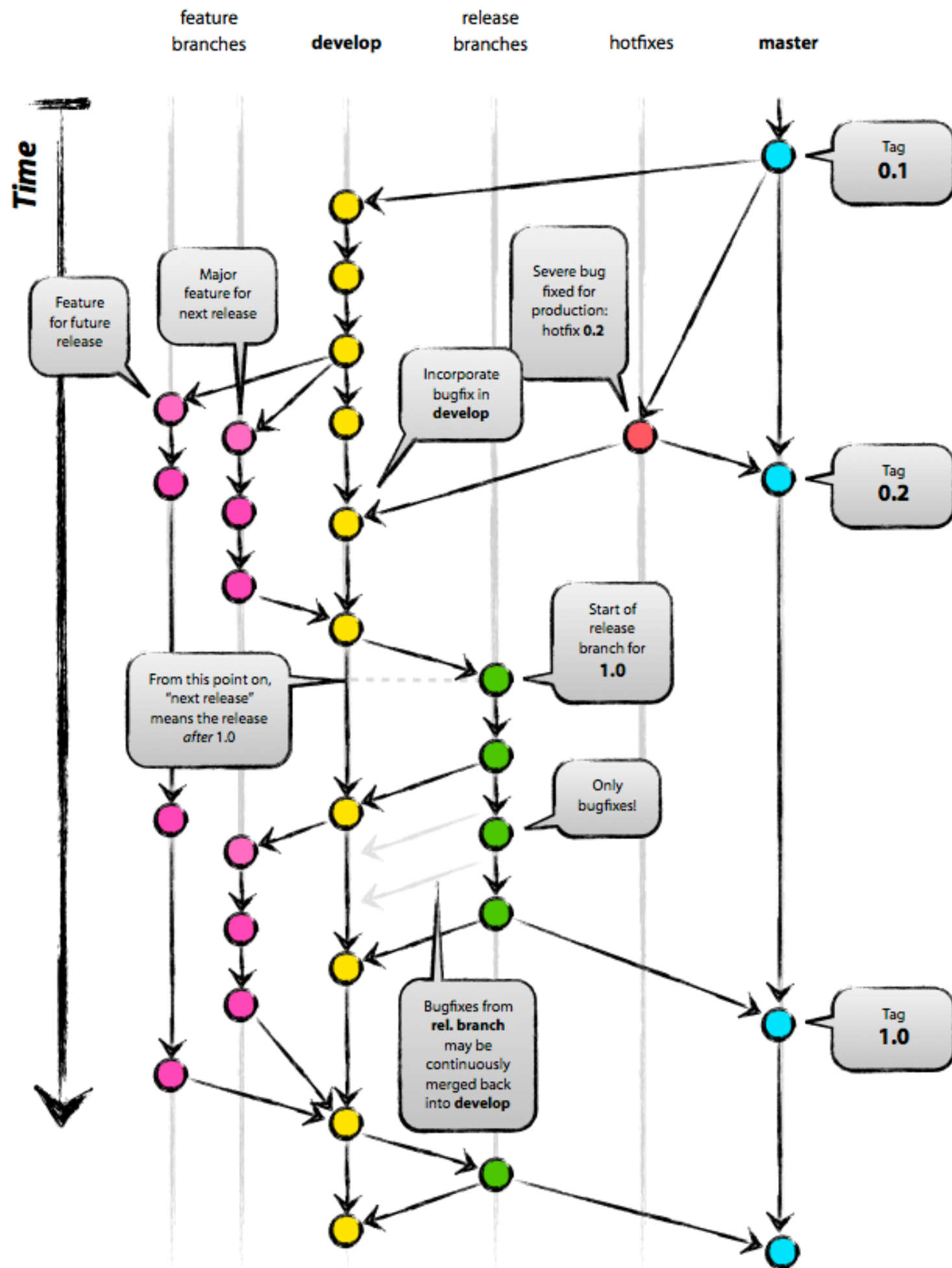
合併分支 git merge

- 在新分支開發完成時，可以用 **git merge** 將新分支合併回原分支。
- 在合併的時候時可以加上 **--no-ff** 參數，來取消 fast forward。這樣可以保證透過一個 merge commit 將新分支合併回原分支，而不是直接將所有提交直接接在原分支之後。



建議流程 git flow

- git flow 是一套不錯的 git 使用流程。若無特殊規範或考量可以採用這套流程。
- 主要分支
 - master: 永遠處在可上線的狀態
 - develop: 主要的開發分支
- 支援性分支
 - feature: 開發新功能都從 develop 分支出來，會合併回 develop
 - release: 準備要 release 的版本，只修 bugs。從 develop 分出來，會合併回 master 和 develop
 - hotfix: 用來修正線上版本的 bug。從 master 分出來，會合併回 master 和 develop



新增遠端倉庫 git remote

- 我們可以將本地的 git repository 放到遠端倉庫，進行代碼托管。如此一來可以多人同時開發，也可以避免意外導致資料遺失。目前 Github 與 Bitbucket 等服務商有提供免費代碼托管。若想自行架設可考開源的 Gitlab。
- `git remote -v` 可以列出目前專案設定的遠端倉庫
- `git remote add <name> <url>` 可以新增遠端倉庫
- `git remote -h` 可以查看其他用法

推送至遠端 git push

- 在新增完遠端倉庫之後，就可以將本地的更新推送上去
- `git push -u <remote> <branch>`
- `git push` 若加上 `-u` 參數，除了會將分支推上去外，還會將遠端的分支設成成本地分支追蹤的 upstream。
 - 例：`git push -u origin develop` 會將本地的 develop 推上 origin，並將本地 develop 設定追蹤遠端 origin/develop。
- 若本地的分支與遠端有發生分歧的情形，例如在 `rebase` 之後，將會無法直接 `push` 回遠端。可以使用 `-f` 參數強制覆寫遠端的分支。
 - 例：`git push -u gitlab feature/foo -f`
- `git push -h` 可以查看其他用法

複製遠端倉庫 git clone

- `git clone <url>`
- `git clone` 是用來將遠端的 repository 完整複製到本地，包含完整的 git 記錄，也會自動設定好 remote，不用再手動新增。
- `clone` 完之後就可以開始在上面進行開發，再透過 `git push` 即可更新回遠端。
- `clone` 之後可以使用 `git checkout <remote-branch>`，git 會幫你在本地建立一個新分支，並將 remote 對應的分支拉進來設定為追蹤的 upstream。或是可以手動完成：
 - `git checkout -b <branch> --track <remote>/<branch>`

更新本地 git fetch & git pull

- `git fetch <remote>` 更新儲存在本地裡的遠端分支資訊，並不影響到本地分支。
- `git pull` 其實是 `git fetch` 之後加上 `git merge`，也就是在 `git fetch` 之後，再將對應的遠端追蹤分支合併進你當前的本地分支。
- 要注意的是，如果你本地的分支與遠端的分支有分歧，合併進來的話將會產生 merge commit，建議可以使用 `git pull --rebase`，這樣的話就用 `rebase` 而不是 `merge`。

選擇提交

- git 有許多方法可以選擇某個特定的提交或是範圍。例如可以直接使用 SHA 指定特定的提交。
- 或是用 “~” 或 “^” 來往前選擇相對提交，例：
 - HEAD^2 是指第二個父提交，只對 merge commit 有用
 - HEAD~3 是指往前三個的提交
 - HEAD@{2} 是指上上次 HEAD 所指的提交
- 透過 “A..B” 可以選擇在 B 分支中卻不在 A 分支中的提交。
- 透過 “A...B” 可以選擇不在 B 分支也不在 A 分支的提交。
- 詳細請參考 Pro Git - 選擇修訂版本

參考資料與延伸閱讀

- Git 網站
- Git 手冊
- Pro Git 電子書
- GitHub Cheat Sheet PDF
- Git Concepts Simplified by Sitaram Chamarty