git

一個分散式版本控制distributed version control軟體，即在每個開發者電腦中複製一份完整的數據庫以及完整歷史，因此無法連接網路時，仍可以進行軟體的分支及合併，而且系統的數據庫可以在多家電腦上備份，不需靠單一位置的備份，再透過其他機制來達到同步

git名詞

級別:

system所有使用者

global當前使用者的所有數據庫

local當前數據庫

區域:

Remote Repository遠端數據庫:

為了多人共用，配有專用的伺服器

Local Repository本地數據庫:

為了個人使用，配置在自己的機器

Index索引、staging area暫存區:

位於工作區和數據庫之間，保存下一次向數據庫提交快照的暫存區

Working Space工作區、Working Directory工作目錄:

保存目前正在處理檔案的目錄，Git相關的操作都會在這個目錄下完成

branch分支:

指向某個commit的指標(該commit的SHA-1值)，刪除分支並不會造成commit消失

Master:

在數據庫進行最初的提交後，Git會建立一個master分支，之後的提交在切換分支之前都在master分支裡

Integration主幹:

必須維持穩定，新分支會建立在它上面，可以隨時發布版本的分支，通常會將master當作Integration使用

Topic側枝:

為了開發功能、修復錯誤所建立的分支，完成作業後，要將Topic合併到Integration

HEAD:

指定分支branch的指定commit版本，在建立新數據庫時，Git會預設HEAD指向master分支的最新commit版本，可以藉著移動HEAD的指向，更新正在使用的分支、版本

Stash:

可以暫存工作、暫存區中尚未提交的修改內容或新增檔案，可以在事後再取出暫存的修改，應用到原先的分支或者其他的分支中，在checkout到其他的分支時，會從原來的分支移動到切換後的分支，但如果在切換後的分支有相同檔案，而且兩者有任何不同時，checkout會失敗(要先提交修改內容，或者將修改內容放到stash中暫時儲存後再checkout)

~n上幾代父提交(n默認1)

HEAD~ = HEAD~1 //上1代父提交

HEAD~~ = HEAD~2 //上2代父提交

^n第幾個父提交(n默認1)

HEAD^ = HEAD^1 //上1代第1個父提交

HEAD^2 //上1代第2個父提交

HEAD^^ = HEAD^1^1 //上2代第1個父提交

HEAD ^^^2 = HEAD ~2^2

SHA-1值

使用SHA-1演算法計算的結果，用於辨認不同的commit，只要檔案內容或提交信息不同，就會視為新的commit，產生新的SHA-1值

ORIG\_HEAD

當做一些比較危險的操作，例如merge、rebase、reset，Git就會把HEAD的狀態(最近一次危險動作之前的SHA-1值)放置在這裡，讓你隨時可以跳回危險動作之前的狀態

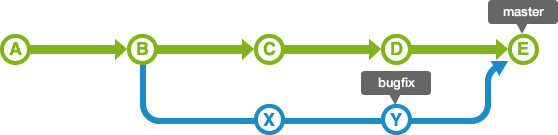
git概念

合併:

merge:修改內容的歷史記錄會維持原狀，但合併後的歷史紀錄比較複雜

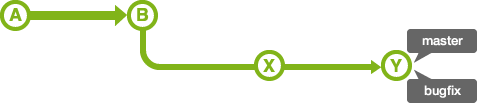
1. merge bugfix到master時，如果master有更改過:

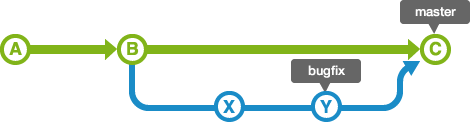
會產生一個合併提交(E)



2. merge bugfix到master時，如果master沒有更改過(fast-forward快轉合併):

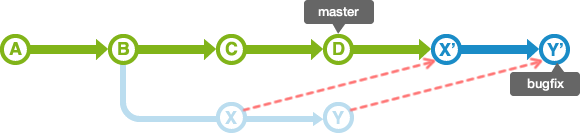
因為bugfix的歷史記錄包含master的歷史記錄，所以只要把bugfix移動到master，master就可以導入bugfix的內容



3.使用merge的non fast-forward參數，即使是可以fast-forward的合併也會產生一個合併提交(C)

rebase:修改內容的歷史記錄會接在要合併的分支後面，合併後的歷史記錄比較簡單，但比用merge更容易發生衝突

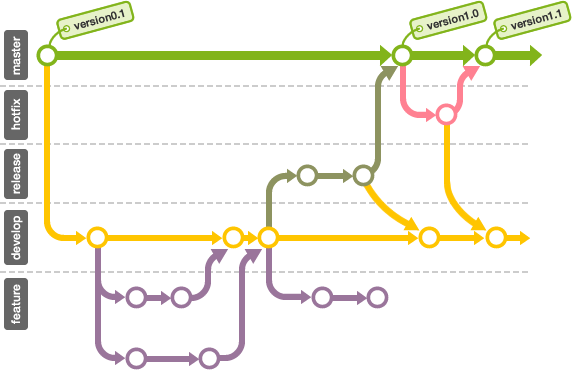
rebase bugfix到master，bugfix的歷史記錄會增加在master的後面(歷史記錄會被統一，形成簡單的一條線)，移動提交X、Y有可能會發生衝突，所以需要修改衝突部分



!執行Rebase時，master的位置不變(因此須將master merge到bugfix後，master的HEAD才會移動到bugfix的HEAD)



A successful Git branching model、Git Flow:



Master分支:

用來放穩定、隨時可上線的版本，只能從別的分支合併過來(開發者不會直接commit到這個分支)，通常會在commit打上版本號標籤

Develop分支

所有開發的基礎分支，當要新增功能(Feature分支)都是從這個分支切出去，新功能完成後，也都會合併回來

Feature分支:

在開發新功能或修復錯誤時，從develop建立的分支，操作基本上不需要共享(不需要在遠端數據庫建立分支)，完成開發後，合併回develop

Release分支:

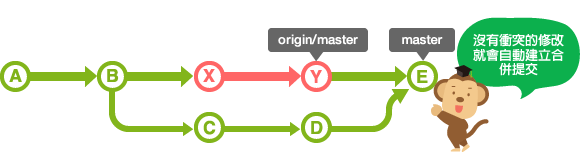
為了做發布前最後錯誤修復，從develop建立的分支(成員可以在這個分支裡進行最後的調整，而其他成員依舊可以在develop 支上進行功能開發，不會相互干擾而影響最後發布)，測試完成後，Release分支會同時合併到Master、Develop分支(確保release最新的錯誤修復同步到develop和master)

Hotfix分支:

當發布的產品需要緊急修改時，會從Master分支開一個Hotfix分支出來進行修復(從Develop建立可發布版本要花許多的時間)，Hotfix修復後，會同時合併到Master、Develop分支(確保Hotfix修復同步到Master和Develop)

conflict衝突:

本地的上次到下次push之間，如果有其他人更新了遠端數據庫(X、Y)，且本地沒有更新本地數據庫，那麼push會被拒絕(若沒有合併直接覆蓋歷史記錄的話，其他人push的修改內容會消失)，因此需要先pull(從遠端數據庫取得最新的修改歷史)，再合併(merge)



執行合併會自動合併修改部分，但兩者檔案中同一行都發生了修改，就會發生衝突，需手動合併

merge:

先修改衝突部分(==分隔上方是當前檔案衝突部分或本地數據庫，下方是合併檔案衝突部分或遠端數據庫)，再提交合併提交(git add <修改後檔案>、git commit -m "合併提交")

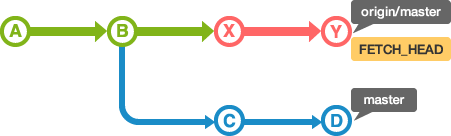
rebase:

先修改衝突部分(==分隔上方是當前檔案衝突部分，下方是合併檔案衝突部分)，再繼續rebase (git add <修改後檔案>、git rebase --continue)

遠端:

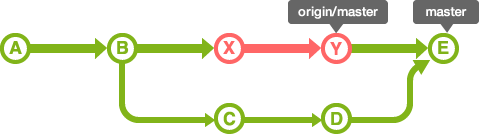
fetch:

取得遠端數據庫的最新歷史記錄(取得commit會導入自動建立FETCH\_HEAD分支)



pull = fetch + merge:

取得遠端數據庫的最新歷史記錄，並將遠端數據庫合併到本地數據庫(合併提交)



push:

將本地數據庫合併到遠端數據庫

Git會以fast-forward合併，以防止覆蓋以前的提交，如果發生衝突，push會被拒絕，需要先pull最新的遠端變更，再進行一次push

!基本上不能修改遠端數據庫共享的提交

標籤tag

指向某個commit的指標，用於標記特定的commit，可以搭配checkout、reset使用，通常用來標記發布版本，打上標籤的commit是固定的，不能隨意的變更位置，因為標籤不會移動

lightweight tag輕量標籤:

本地暫時性使用

不可變更的暫時標籤

可以添加名稱

annotated tag標示標籤:

用於重要的提交，例如用來增加發布提交上的註解或簽名

可以添加標簽者的名稱、email及日期

可以添加名稱

可以添加註解

可以添加簽名

語意化版本控制Semantic Versioning SemVer

良好的語意化版本控制，可以讓引用的人，清楚知道你開發的功能是否被修改或移除，而能夠決定是要拒絕更新，或者配合新版本，修正程式碼

主版號MAJOR:當你做了不相容的API修改

次版號MINOR:當你做了向下相容的功能性新增

修訂號PATCH:當你做了向下相容的問題修正

MAJOR.MINOR.PATCH

安裝此版本

^ MAJOR.MINOR.PATCH

保持MAJOR，MINOR、PATCH安裝最新版本

~ MAJOR.MINOR.PATCH

保持MAJOR、MINOR，PATCH安裝最新版本

約定式提交Conventional Commits

<類型type>(可選的作用範圍scope): <描述description>

[可選的正文body]

[可選的頁腳footer]

type(必選):commit的類別，例如:

feat:新增/修改功能(次版號MINOR)

fix:修復bug(修訂號PATCH)

docs:只修改文件

style:不影響程式碼含義的格式更改，例如空格，格式，缺少分號等

refactor:程式碼重構(既不是feat，也不是fix的程式碼變動)

perf:改善效能

test:新增/修改測試

build:影響構建系統或外部依賴項的更改，例如gulp、broccoli、npm

ci:對我們的CI配置文件和腳本的更改，例如Travis、Circle、BrowserStack、SauceLabs

scope(可選)commit影響的範圍，更改程式區段的位置

description (必選)commit的簡短描述，以動詞開頭，使用第一人稱現在時，比如change、第一個字母小寫、結尾不加句號

body(可選):Commit的詳細描述，可以分成多行，每一行不要超過72個字元，使用第一人稱現在時，說明程式碼變動的項目與原因，還有與先前行為的對比

footer(可選):必須包含一個符記token，並接着以: 或 #分隔，再緊鄰一個字串值

符記token:使用-作為空白字元，如Acked-by(有助於區分出頁腳與多段落的正文)，BREAKING CHANGE例外

頁腳的字串值可以包含空白與換行，在遇到下一組有效的符記/分隔時停止解析

重大變更(主版號MAJOR):必須作為提交中的類型或作用範圍的後綴(!緊鄰於:之前)，或是在頁腳中作為一個符記token(BREAKING CHANGE: )

git設置

設置您的用戶名和電子郵件地址(每個Git提交都使用此信息)

git config --global user.name "John Doe"

git config --global user.email johndoe@example.com

配置默認文本編輯器(Visual Studio Code)

git config --global core.editor "code --wait"

中文的檔名可以正確顯示

git config --global core.quotepath off

只要在專案目錄裡放一個.gitignore檔案，就可以設定Git忽略的檔案

忽略01.txt檔案

01.txt

忽略pro目錄下的02.txt檔案

pro/02.txt

忽略所有db目錄下附檔名是.html的檔案

/db/\*.html

忽略所有附檔名是.js的檔案

\*.js

git config --global alias.co checkout

設置縮寫git co等同git checkout

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

設置縮寫git l等同git log --oneline --graph

基本Git命令

創建數據庫:

git init初始化一個資料夾，建立全新數據庫，並開始跟踪現有目錄(在現有目錄中添加了一個隱藏的子文件夾，該子文件夾中包含版本控制所需的內部數據結構)

git clone遠端數據庫→工作區:

git clone <repository遠端數據庫url>

在現有目錄創建遠端已存在項目的本地副本(包括項目的所有文件、歷史記錄、分支)

git clone <repository遠端數據庫url> <directory新目錄名稱>

在新目錄創建遠端已存在項目的本地副本

查詢資料:

git status

顯示尚未被提交的檔案狀態，分別為:

(暫存)將要提交的檔案Changes to be committed

(工作區)被更動但尚未要提交的檔案Changes not staged for commit:先前就已經被add，但被修改modified或是刪除deleted的檔案

(工作區)未被追蹤的檔案Untracked files:完全新加入的檔案，不曾被add

git log

顯示數據庫的歷史提交記錄(上新下舊)

--reverse顯示提交日誌(上舊下新)

--oneline僅用一行顯示每次的commit(拿掉了作者、日期相關資訊，只保留驗證字串的七碼)

--graph以圖表方式顯示分支歷史

--decorate顯示包含標籤資料的歷史記錄

--author="作者名稱"顯示指定作者的提交

--committer="提交者名稱"顯示指定提交者的提交

--grep="提交信息"顯示包含指定提交信息的提交

-S "檔案內容"顯示包含指定檔案內容的提交

--since=<date>

--after=<date>

顯示指定時間之後的提交

--until=<date>

--before=<date>

顯示指定時間之前的提交

git log <檔案名稱>

顯示包含指定檔案的提交

gitk呼叫gitk (git內置GUI工具)，用圖形化介面確認歷史提交記錄

git blame <檔案名稱>

顯示文件各行最後修改的版本和作者

-L <start>,<end>

限制行數在<start>到<end>(兩者皆包含)

git reflog

顯示reflog信息(當HEAD有移動就會在Reflog中記錄)

本地操作:

git add工作區(未跟踪)→暫存區(跟踪):

git add file1.txt file2.txt

添加file1.txt和file2.txt至索引

git add \*.txt

添加所有.txt至索引

git add .

添加當前目錄中所有檔案至索引

git add –all

添加所有檔案至索引

git add –f

允許添加.gitignore中忽略的檔案

git commit暫存區→本地數據庫:

git commit -m “提交信息”

提交暫存區內容、提交信息至本地數據庫

git commit

無參數執行，將打開默認文本編輯器以編寫提交消息，第一行是主題(簡短描述)，其後留空行，其他所有內容都是正文(擴展描述)，建議用約定式提交

git commit --no-edit

不編輯提交信息

git rm <pathspec檔案路徑>

刪除文件，並從暫存區中刪除文件(等同rm + git add)

git rm –cached <pathspec檔案路徑>

取消git跟踪，並從暫存區中刪除文件

git rm –r <pathspec檔案路徑>

刪除文件，並從暫存區中刪除文件(遞迴刪除，可以刪除資料夾)

git checkout <檔案名稱>

恢復刪除(deleted)的工作區文件

git mv <oldname> <newname>

重命名文件

遠端操作:

git remote add <name遠端數據庫名稱> <url遠端數據庫url>

新增並命名一個遠端數據庫

git remote

顯示遠端數據庫列表

-v

顯示詳細的遠端數據庫列表(有遠端數據庫url)

git push本地數據庫→遠端數據庫:

git push <repository遠端數據庫名稱> <refspec指定分支名稱>

上傳本地指定分支到遠端指定分支(如果遠端指定分支不存在，就在遠端新增一個分支)，repository一般會使用origin，refspec可以是本地和遠端共用分支名稱或本地分支名稱:遠端分支名稱兩種格式，push、pull、fetch皆適用

–u設定upstream上游(只要做過一次，並且成功push出去，之後無參數的git push、git pull都會套用相同<遠端數據庫名稱> <指定分支名稱>)

git pull遠端數據庫→工作區:

git pull <repository遠端數據庫名稱> <refspec指定分支名稱>

將遠端指定分支合併到本地指定分支

git fetch <repository遠端數據庫名稱> <refspec指定分支名稱>

將遠端指定分支下載到本地指定分支

分支操作:

git branch

顯示分支列表(前面有\*就是現在的分支)

git branch <branchname分支名稱>

建立分支

git branch <branchname分支名稱> <start-point用作分支HEAD的commit>

建立具有指定HEAD的分支(start-point默認為當前分支HEAD)

git branch -d <branch分支名稱>

刪除分支(刪除的分支必須完全合併到其上游分支)

git branch -D <branch分支名稱>

強制刪除分支(刪除的分支不需完全合併到其上游分支)

git branch -m <舊名稱> <新名稱>

重新命名分支和相應的reflog

git checkout <branch分支名稱>

切換分支

-b

建立分支，並切換至新分支

git merge <commit分支名稱>

將指定分支合併到當前分支

--no-ff

即使可以fast-forward快轉合併，也會新增合併提交

git rebase <branch將當前分支合併至的目標分支>

將當前分支合併至目標分支

git rebase --continue

解決合併衝突後，重新啟動rebase

git rebase --abort

在中途停止rebase的操作，並還原(將HEAD重置為原始分支，如果rebase時，提供了<branch>，則HEAD將重置為<branch>；否則，HEAD將被重置為開始rebase的位置)

git stash

將已跟蹤的暫存、工作區檔案儲存在stash中

-u

將未跟蹤的暫存、工作區檔案也一同儲存在stash中

git stash list

列出當前stash的儲存列表

git stash apply stash@{<revision>}

將指定stash的儲存套用在當前分支上

git stash drop stash@{<revision>}

刪除指定stash的儲存

git stash pop stash@{<revision>}

將指定stash的儲存套用在當前分支上，並刪除

標籤操作:

git tag

顯示標籤列表

git tag -n

顯示標籤列表和註釋(未註釋的標籤顯示提交信息)

git tag <tagname標籤名稱>

在當前提交添加標籤(輕量標籤)

git tag <tagname標籤名稱> <commit指定提交>

在指定提交添加標籤(輕量標籤)

git tag -a <tagname標籤名稱>

添加帶註釋的標籤(標示標籤)，啟動編輯器添加標籤的註釋

git tag –a -m "註釋" <tagname標籤名稱>

添加帶註釋的標籤(標示標籤)，直接添加註釋

git tag -d <tagname標籤名稱>

刪除標籤

改寫提交:

git commit --amend

創建新commit來替換當前分支的最後一次commit(用於增加最後一次提交漏掉的檔案、修改最後一次提交的內容、註解)

git revert <commit指定提交SHA-1值或HEAD相對父代位置>

取消指定提交(用於安全取消過去發布的提交)



--no-edit

不編輯提交信息

git reset <mode> <commit指定提交SHA-1值或HEAD相對父代位置>

將當前HEAD重置為指定提交狀態

<mode>包括:

--soft

修改HEAD的位置，不修改索引、工作目錄(commit拆出來的檔案在暫存區)

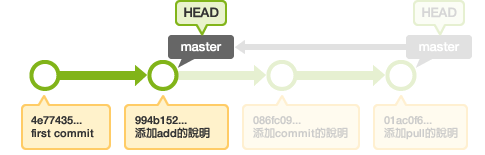
--mixed(默認)

修改HEAD的位置、索引，不修改工作目錄(commit拆出來的檔案在工作區)

--hard

修改HEAD的位置、索引、工作目錄(commit拆出來的檔案被丟棄)

git reset --hard HEAD~~

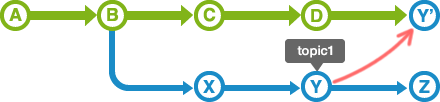


git reset --hard ORIG\_HEAD

還原到reset前的狀態

git cherry-pick <commit指定提交SHA-1值或HEAD相對父代位置>

從其他分支複製指定提交，並導入現在分支

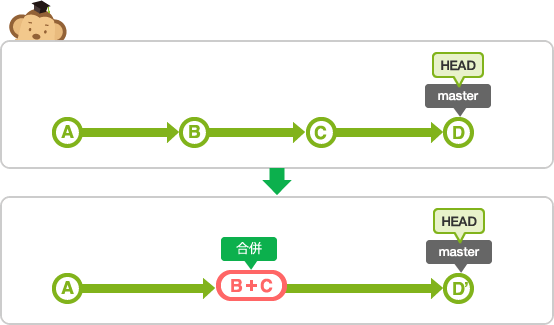


--no-commit

複製指定提交不會直接導入，會先待在暫存區

git rebase -i <commit指定提交SHA-1值或HEAD相對父代位置>

進入Rebase的交互模式(從指定提交(不包含)開始把多個commit合併成一個commit)，執行後開啟編輯器視窗(上舊下新)，將pick修改squash儲存後，編輯合併後commit的提交信息



pick(默認)

保留commit

reword(修改提交信息)

保留commit，並重新編輯該commit的提交信息

edit(修改提交內容、提交信息)

rebase執行到該commit時會暫停，可以在此commit基礎上進行内容修改后，執行git add、git commit –amend(重新編輯該commit的提交信息)，然後執行git rebase –continue繼續未完成的rebase

squash(合併提交內容，修改提交信息)

將該commit合併至上一個commit(自動合併兩者的提交信息)，可以重新編輯合併後commit的提交信息

$ git log --oneline

27f6ed6 (HEAD -> master) add dog 2

2bab3e7 add dog 1

ca40fc9 add 2 cats

1de2076 add cat 2

cd82f29 add cat 1

382a2a5 add database settings

bb0c9c2 init commit

$ git rebase -i bb0c9c2

//編輯器視窗

pick 382a2a5 add database settings

pick cd82f29 add cat 1

pick 1de2076 add cat 2

pick ca40fc9 add 2 cats

pick 2bab3e7 add dog 1

pick 27f6ed6 add dog 2

//編輯器視窗修改後(squash向上合併入pick)

pick 382a2a5 add database settings

pick cd82f29 add cat 1 // ca40fc9併入1de2076併入cd82f29

squash 1de2076 add cat 2

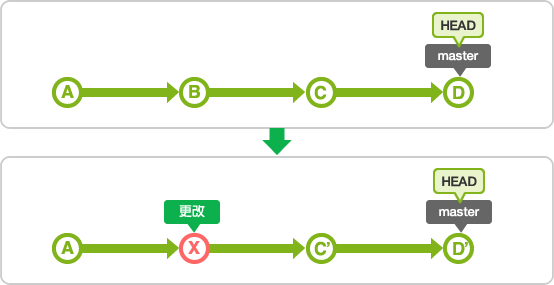
squash ca40fc9 add 2 cats

pick 2bab3e7 add dog 1 // 27f6ed6併入2bab3e7

squash 27f6ed6 add dog 2

git rebase -i <commit指定提交SHA-1值或HEAD相對父代位置>

進入Rebase的交互模式(從指定提交(不包含)開始修改commit，可以搭配git reset，將一個commit分割成多個commit，或在舊有commit間新增commit)，執行後開啟編輯器視窗(上舊下新)，將pick修改edit儲存後，在此commit基礎上修改内容后，執行git add、git commit –amend(重新編輯該commit的提交信息)，執行git rebase –continue繼續未完成的rebase(有衝突需修改檔案)



$ git log --oneline

27f6ed6 (HEAD -> master) add 04

1de2076 add 03

cd82f29 add 02

382a2a5 add 01

bb0c9c2 add 00

$ git rebase -i head~4

//編輯器視窗

pick 382a2a5 add 01

pick cd82f29 add 02

pick 1de2076 add 03

pick 27f6ed6 add 04

//編輯器視窗修改後(edit是要修改的commit)

pick 382a2a5 add 01

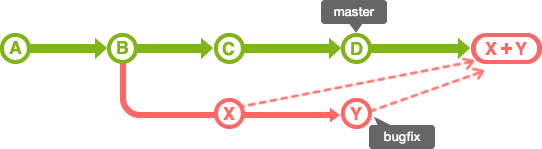
edit cd82f29 add 02

pick 1de2076 add 03

pick 27f6ed6 add 04

git merge --squash <commit分支名稱>

將指定分支合併到當前分支，同時合併提交



電子郵件:

git format-patch <開始的commit SHA-1值>..<結束的commit SHA-1值>

在當下目錄將開始(不包括)至結束的commit依序做成多個更新檔Patch

git format-patch -n

在當下目錄將最後n次的commit依序做成多個更新檔Patch

-o <dir>

在指定目錄儲存更新檔Patch

git am <Maildir>

將指定目錄的更新檔套用在現有專案上

Git命令

git status -s以短格式輸出，分別為:

M :updated in index

A :added to index

D :deleted from index

R :renamed in index

C :copied in index

M:work tree changed since index

D:deleted in work tree

R:renamed in work tree

C:copied in work tree

DD:unmerged, both deleted

AU:unmerged, added by us

UD:unmerged, deleted by them

UA: unmerged, added by them

DU unmerged, deleted by us

AA unmerged, both added

UU unmerged, both modified

?? untracked

!! ignored

git commit -a -m 'msg'工作區(已跟蹤)→本地數據庫 將工作區中所有上次提交之后已跟蹤檔案(所有暫存區有紀錄的文件)的修改提交(已在工作區刪除的文件將從暫存區中刪除)

git ls-files顯示工作區和暫存區中已追蹤文件的信息

git diff比對工作目錄與索引之間的差異

git diff –cached/staged將當前的索引狀態與當前分支的最新版進行比對(不會去比對工作區的檔案，而是比對索引與目前最新版之間的差異，有助於在執行git commit之前找出變更的內容)

diff --git a/01.txt b/01.txt //git對哪兩個檔案(前舊後新)進行比對

index ed1b3c2..94bbb12 100644 //git在做這次比對時的標頭資訊Header Line，會標示許多關於此次差異比對的額外資訊

--- a/01.txt //兩個比對的版本中比較舊的版本

+++ b/01.txt //兩個比對的版本中比較新的版本

@@ -1 +1,2 @@ //開始行數,總行數(前舊後新)

add01 //文本內容

+add02