



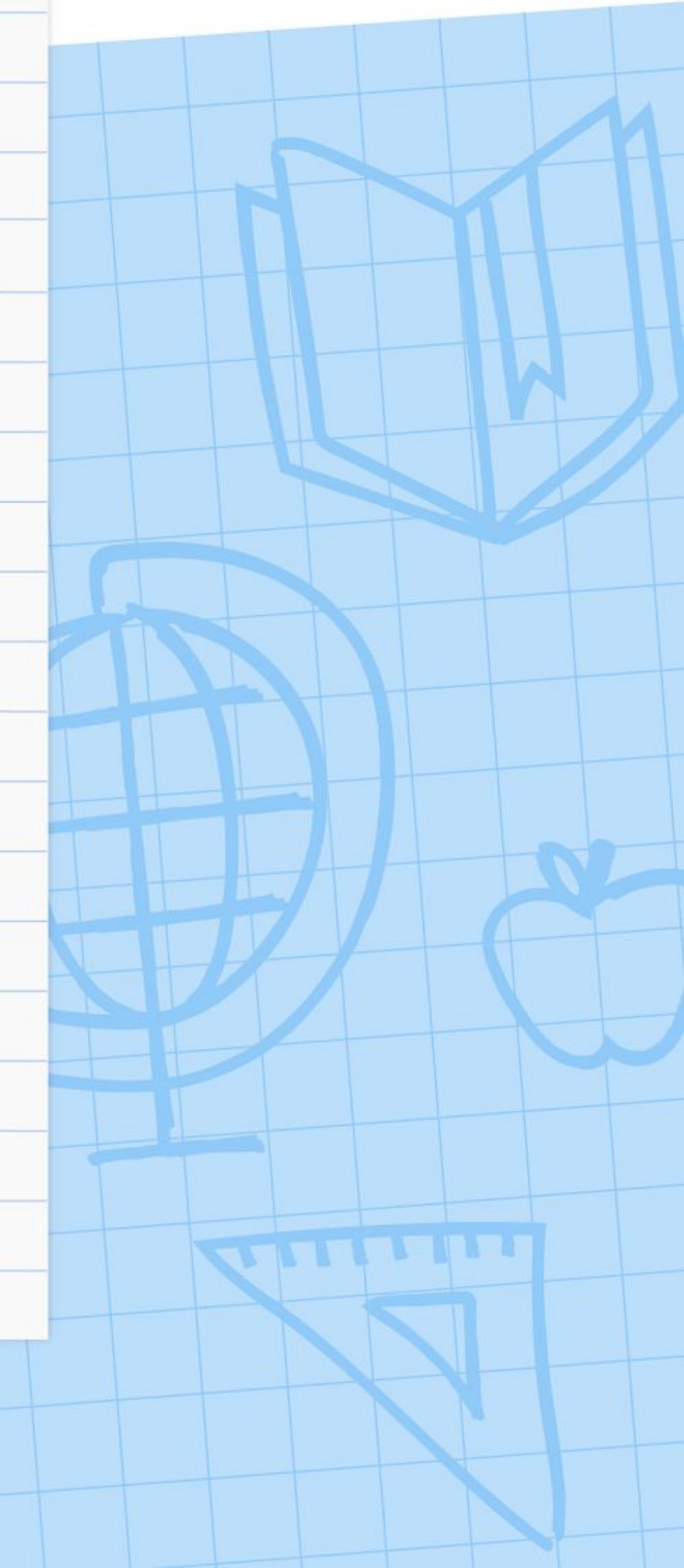
版本控管與高效工作流 - 1+2

對於專案的一舉一動得以控制的方式

讀書會主持人：顏榕嶙(Bernie)

日期：04/01 燕巢場

```
filterByOrg = study.lead_organization === filterByOrg .  
filterStatus = filterByStatus ? study.status === filterByStatus : true  
studyBatchStatus) {  
  
function filterStudies({ studies, filterByOrg = null, filterByStatus = null }) {  
  return studies.filter(study => filterByOrg === null || study.lead_organization === filterByOrg || filterByOrg === null || filterByOrg === study.lead_organization) .filter(study => filterByStatus === null || study.status === filterByStatus || filterByStatus === null || filterByStatus === study.status);  
}
```



目錄, 其之一

- 版本控制概論
- GitHub, 你的夥伴
- 基礎操作指令
- 分支與共用
- 衝突解決



目錄, 其之二

- Git與工作流
- 進階常用指令
 - 重新整理
- 次要存放庫
- 本質





Google Developer Groups
On Campus • National Kaohsiung Normal University

1

版本控制概論

從零開始的版本疊代

```
filterByOrg = study.lead_organization === filterByOrg ? true : false
filterStatus = filterByStatus ? study.status === filterByStatus : true
filterPatchStatus = filterPatchStatus ? study.patch_status === filterPatchStatus : true

function filterStudies({ studies, filterByOrg = false, filterByStatus = false, filterPatchStatus = false }) {
  return studies.filter(study => filterByOrg || filterStatus || filterPatchStatus)
}
```

已經當作伏筆被講很久了

很多次，之前的課程中我總是會提到GitHub以及類似的版本控管工具。

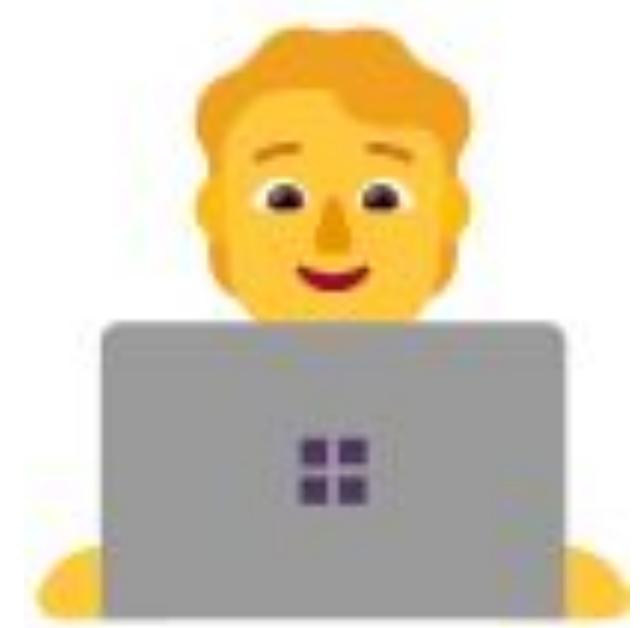
我們作為一個軟體工程師要有所謂的專案思維，因此如何維護自己開發的成果也就相當重要。

我們從為什麼要做版本控管開始，



開發模式

現在的各位都是軟體工程師了



我們跳過一些繁文縟節，直接跳到今天的重點。

版本控管的程序往往在啟動專案前就開始了，因為我們通常會先有一個專案架構以供參考，在 GitHub 上我們也會建立一個遠端存放庫以供使用。

這樣子的好處在哪裡呢？在我們每一次對專案進行改動之後，這些改動就能夠被同步的放上遠端，之後在任何地方都可以用了。



原始的方式

我們不是原始人，我們有最新科技阿



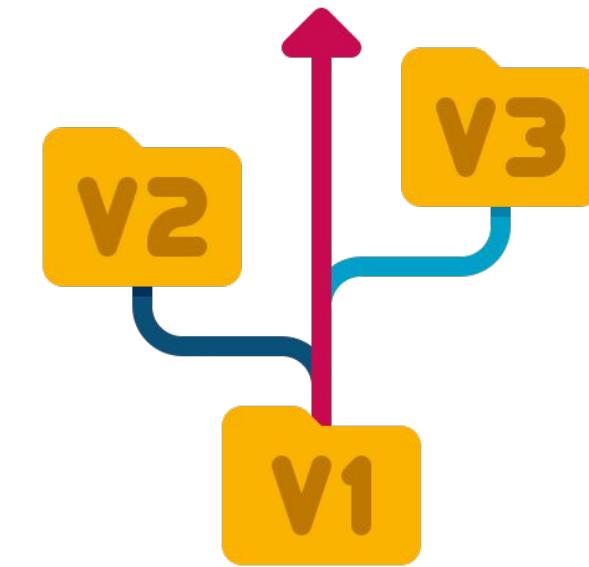
在我們認識版本控管的手段之前，我們來想想一般對檔案的保存會進行哪些種類的操作，如果不希望進度遺失，大部分人會習慣主動備份，把它們給獨立的保存起來。

不過真的會認真地幫每一個備份都編上版本號、標籤以及更新歷史的人少啊，很少啊，畢竟人類是怠惰的生物，並不是所有人都這麼有毅力。

因此，如果有一個協助我們控制版本的工具的話那該有多好，不需要自己手動。

認可：這是一個版本

到底版本是甚麼意思？



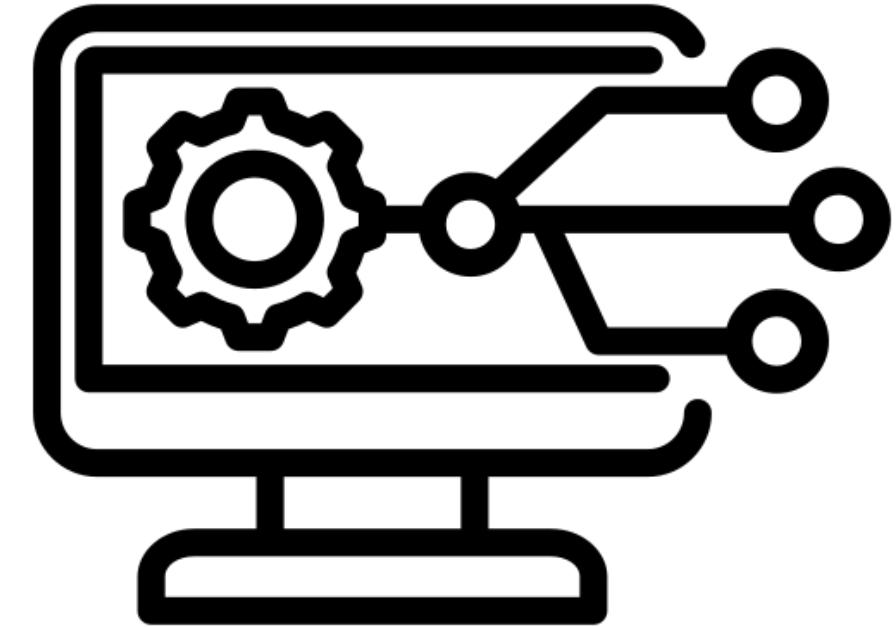
講到這裡，我們還沒有實際的去想過所謂的版本到底意義是什麼，而且是怎麼被決定的。

以一個概念來啟發：每一次新增、更改或者刪除專案檔案時，這個專案當前的狀態就與初始狀態有所不同了，我們可以在不同的時間點中將專案狀態給保存，以便之後更改。

這樣子的保存其實也就是認可當前狀態可以作為一個可用版本存在，因此在專案開發中，判斷什麼時候要進行認可非常重要。

版本控制的形式

唯一真理 v.s. 睽生平等



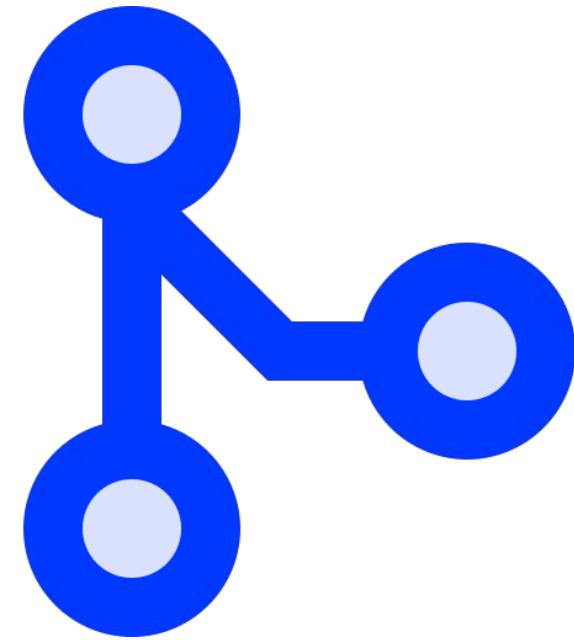
實際上我們有兩種可以進行版控的方式，分別是集中式和分散式VCS，前者是將一切版本資訊給存放在一個伺服器裡，所有人只要連線到伺服器就可以進行修改、查詢（例如SVN）。

另一種比較常見，分散式的核心價值就是在每一台裝置上都有獨立的版本歷史，並能夠相互的追蹤進度，適合個人開發，這種實現方法主要就是透過今天要講的Git來達成。

那麼兩者的優缺點怎麼說？主要還是看情況吧，至少我們不希望每次開發一個新的專案都要重新處理中央導向的問題...

GIT

一步一步推進， GIT是好朋友



我們使用的是Git這個框架來做版本控制，與GitHub本身並沒有直接聯繫，這個語言原先只是作為一個可以被其他前端包裝的後端而開發的，後來之前成為一個完整的框架。

總得來說它是一個協助用的工具，如果沒有Git的發明，現在的人或許還要手動進行備份、認可等等動作，因此做人要感激。

希望各位已經在電腦安裝好Git的環境了，因為我們即將要開始對他進行操作。



Google Developer Groups
On Campus • National Kaohsiung Normal University

2

GitHub, 你的夥伴

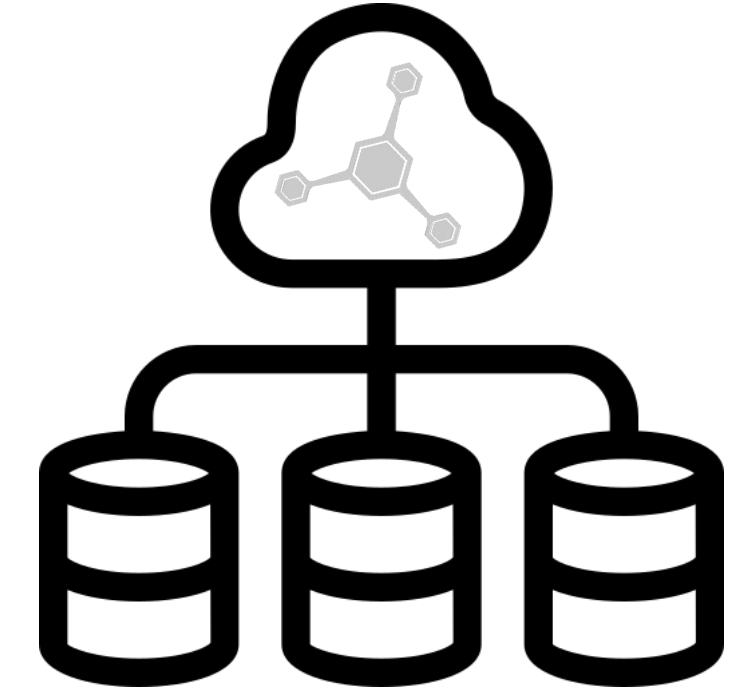
還得是遠端存放庫欸

```
const filterByOrg = study => study.lead_organization === filterByOrg;
const filterStatus = filterByStatus ? study.status === filterByStatus : true;
const matchStatus = filterStatus || !filterByStatus;

function filterStudies({ studies, filterByOrg, filterByStatus }) {
  return studies.filter(study => filterByOrg(study) && matchStatus(study));
}
```

GIT 的 HUB

所有專案的好去處， Git大廳一般的存在



雖然可能有些意外，但GitHub並不是由Git開發者所開發的，這是一個讓人可以存放檔案的網路倉庫，內部架構是用Git框架所實現，而且重點是完全免費（但你可以買Pro啦）。

這東西在我們的整個版本控管流程中擔任著一個Remote的角色，當然我們很快就會講到了。

要先簡單解釋的話呢，我們在自己本機的資料存放位置會是一個資料夾，在裡面進行的更動也還是在資料夾裡，透過Git與GitHub的聯繫我們可以在雲端也放一個完全一致的位置，然後透過指令來進行兩者的同步，這種東西我們叫他...



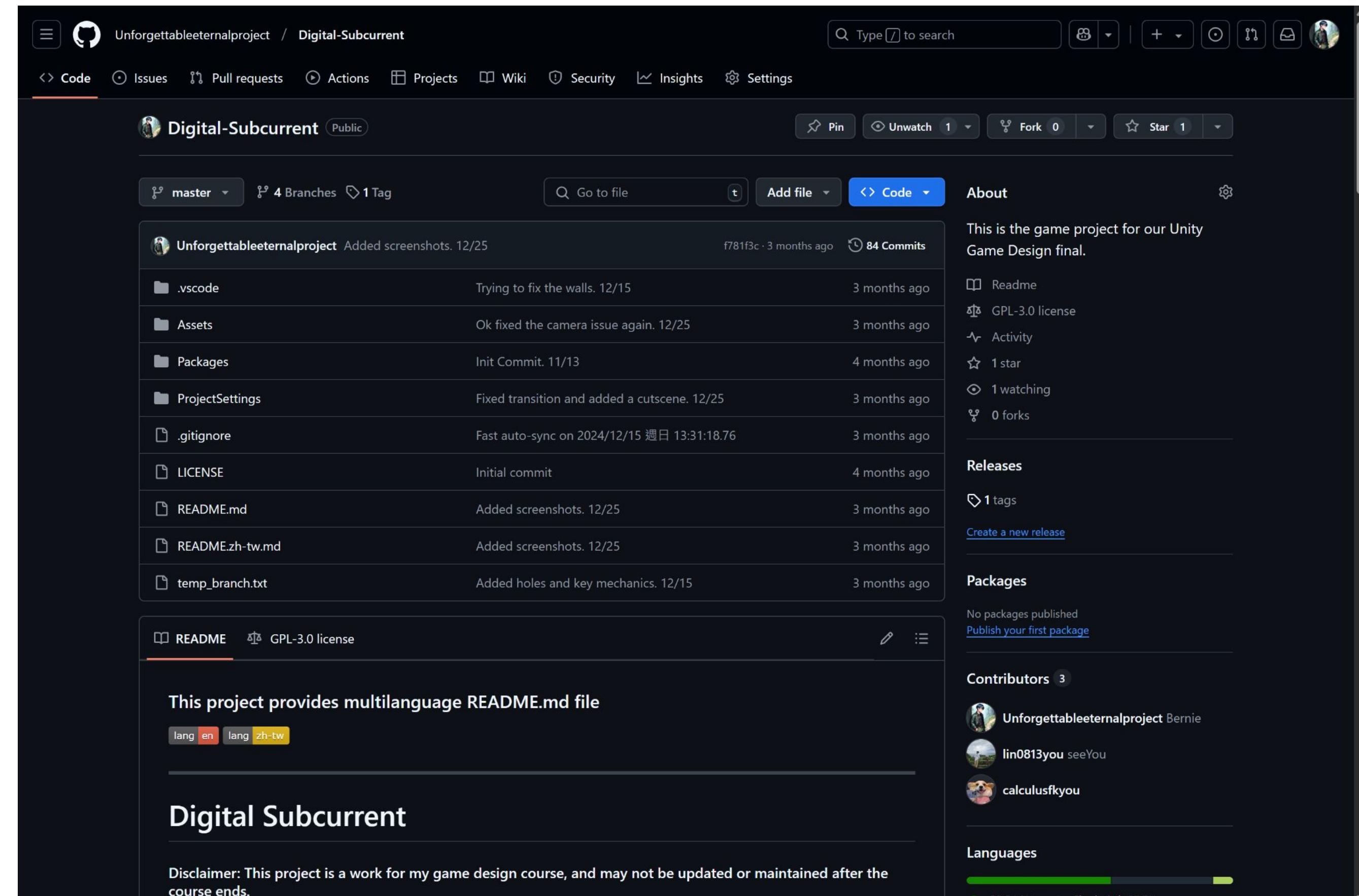
...存放庫 (Repository)

儲存你的程式碼以及各種有的沒的

Git中最重要的機制就是存放庫，用來進行各種Pull, Push, 以及Branch操作，它的用途很多，我們可以在GitHub中視覺化。

你可以把GitHub上的存放庫設為公開或者私人，這會影響外部人士對他的存取。

UI上很多東西對吧，我們一個一個來解釋。



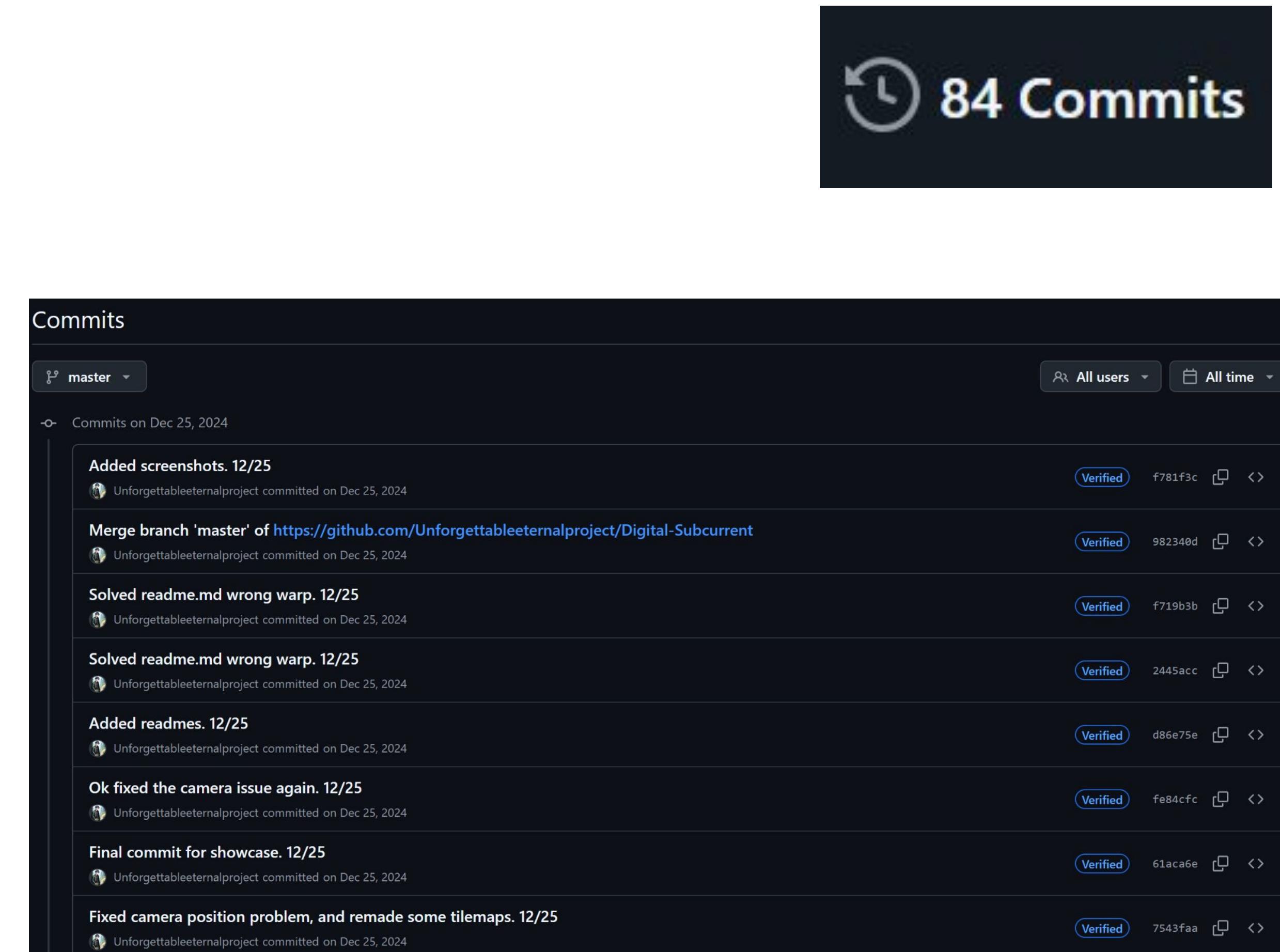
認可提交與版本紀錄

就像日誌一般每一次變動都記得很清楚

先前有提到認可的概念，沒錯這東西是真的很重
要，等等會用git commit來實際帶大家試試。

在存放庫的頁面中我們可以看到所謂的版本紀
錄，也就是每一個認可的紀錄。

當然在不同分支中認可的紀錄也是不同的，不過
所謂的分支又是什麼意思呢？



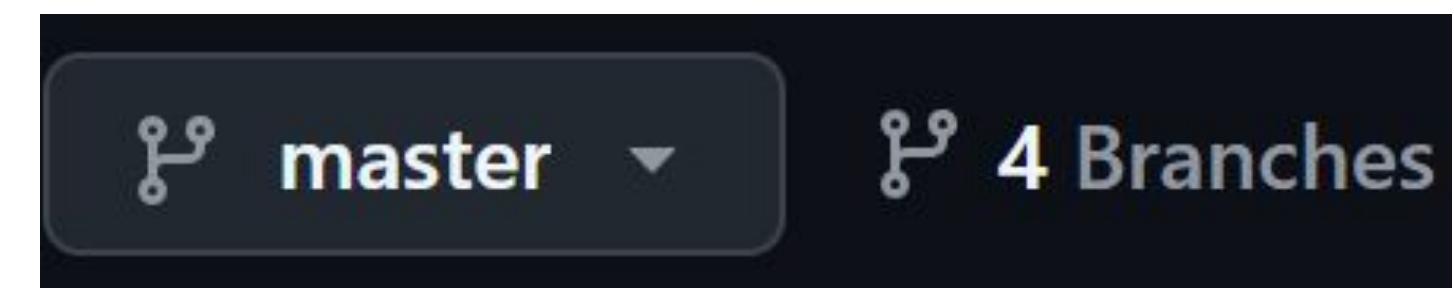
分支管理

從根部衍生出的各個枝條

我們創建存放庫時預設的分支叫做main (或者master, 現在都有在用), 意思就是你的認可存放位置。

今天想要添加實驗性內容時, 為了避免汙染到main的內容, 我們一般會創立新的分支。

在完成更新(Stable)後, 再把他們給合併起來。



Branch	Updated	Check status	Behind	Ahead	Pull request
master	3 months ago	Default			...
SL-3	3 months ago		17	2	...
menuUI_functions	3 months ago	✓ 1 / 1	9	0	#9
revert_feature	3 months ago		31	7	...

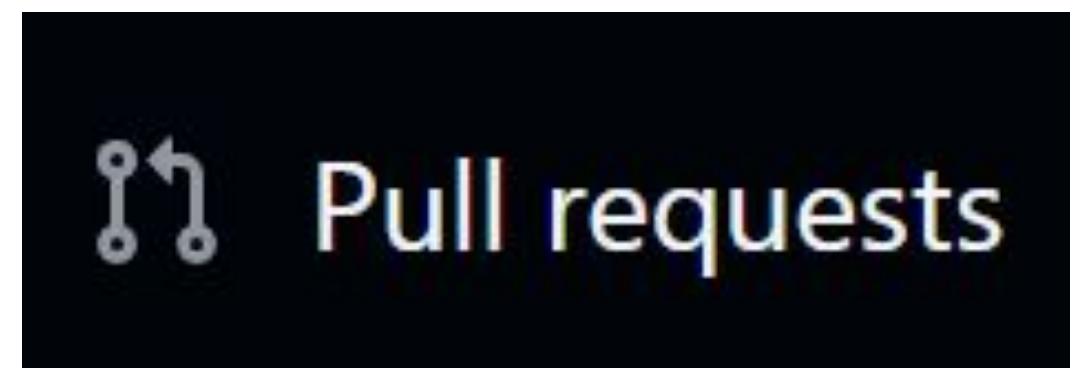
提取請求與合併

這麼做是為了進一步的保護資料

這個是之後會講到的部分，在分支與分支之間的合併時，我們會用到PR的機制，這也被用在Fork和Contribute之中。

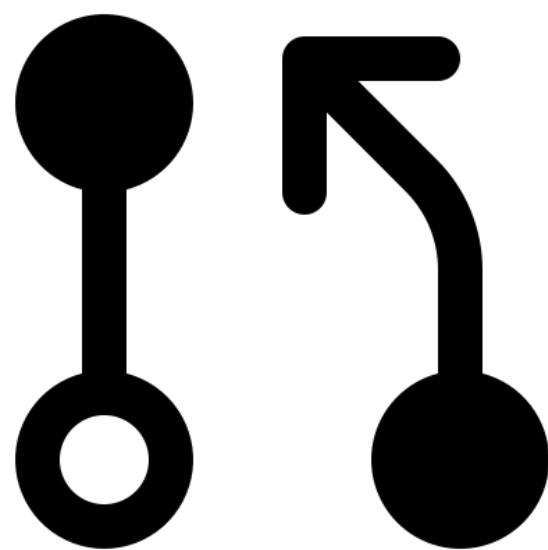
在這裡，來源分支和目標分支中的Conflict需要被解決，否則會無法去做合併。

解決Conflict也不是件容易的事，這裡先給個預告。



從一到一 +

工作區 -> 暫存區 -> 存放庫



我們先來介紹幾個名詞: HEAD, main, origin/main。

其中HEAD是一個指針，指向你當前的工作區分支，一般而言都是在main當中，你所在工作區中做的任何更改都會透過HEAD被放在變更中等待被暫存(Stage)，並且透過Commit被放在你的本地記錄裡。

透過同步可以將本地的認可給推送到雲端存放庫，也就是對應的origin/main位置，這是最簡單的Git運作三步驟。

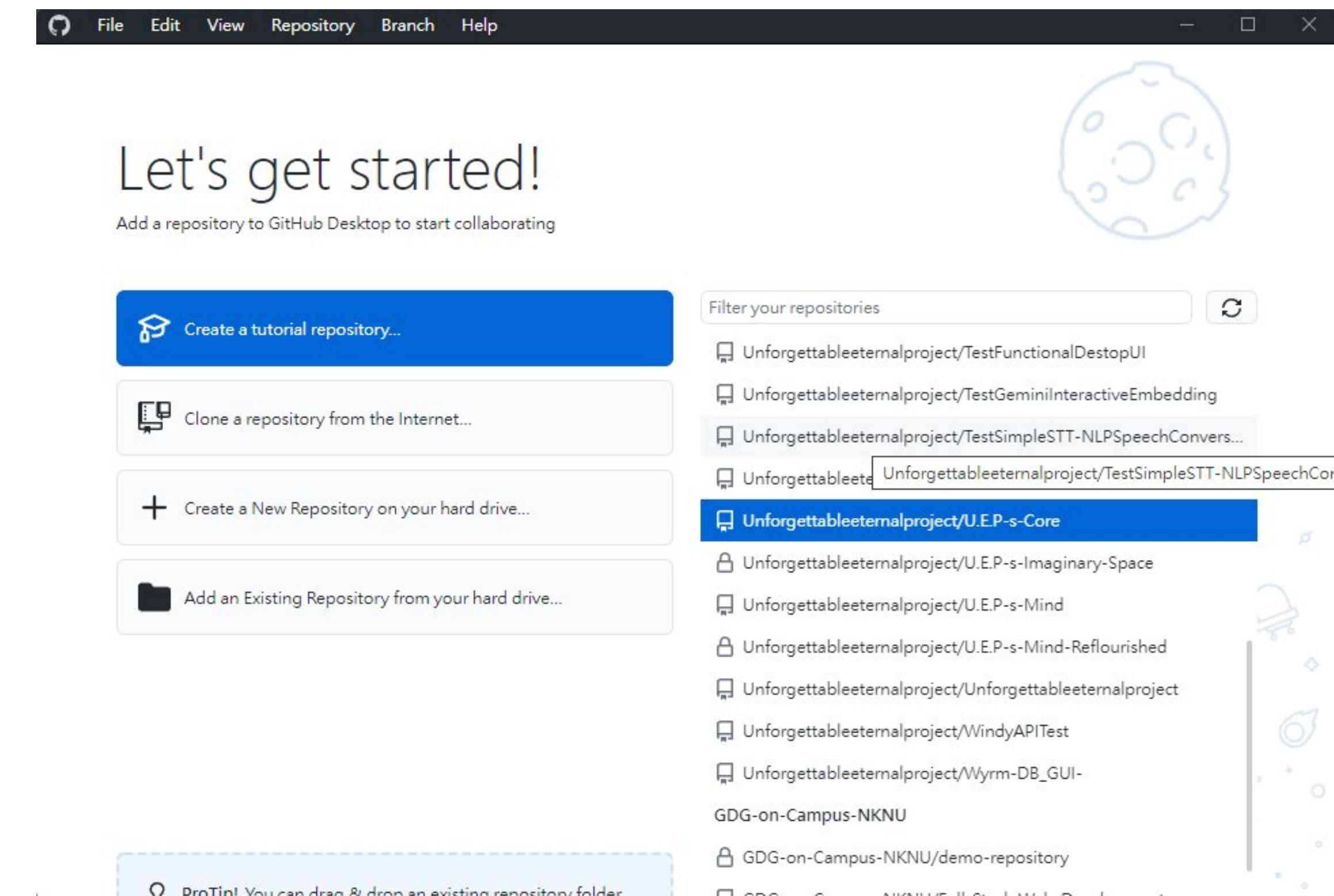
視覺化界面

誰不喜歡視覺化界面？可惜我們今天不會用這個來演示

GitHub其實有提供桌面版本可供下載，也有CLI去讓你進行一些指令的操作（跟等等要用到的Git Bash不同）。

大多數的編譯軟體也有內建Git的相關功能以及與GitHub的連結，之前用過的VS Code就是如此。

不過我們只是講一下，不會用到這東西。



3

基礎操作指令

全部都是GIT

```
choic = filterByOrg ? study.lead_organization === filterByOrg : true
status = filterByStatus ? study.status === filterByStatus : true
matchStatus) {
    Function filterStudies({ studies, filterByOrg =
        filterByOrg ? studies.filter(study => study.lead_organization === filterByOrg) : studies
        .filter(study => study.status === filterByStatus)
    }
}
```



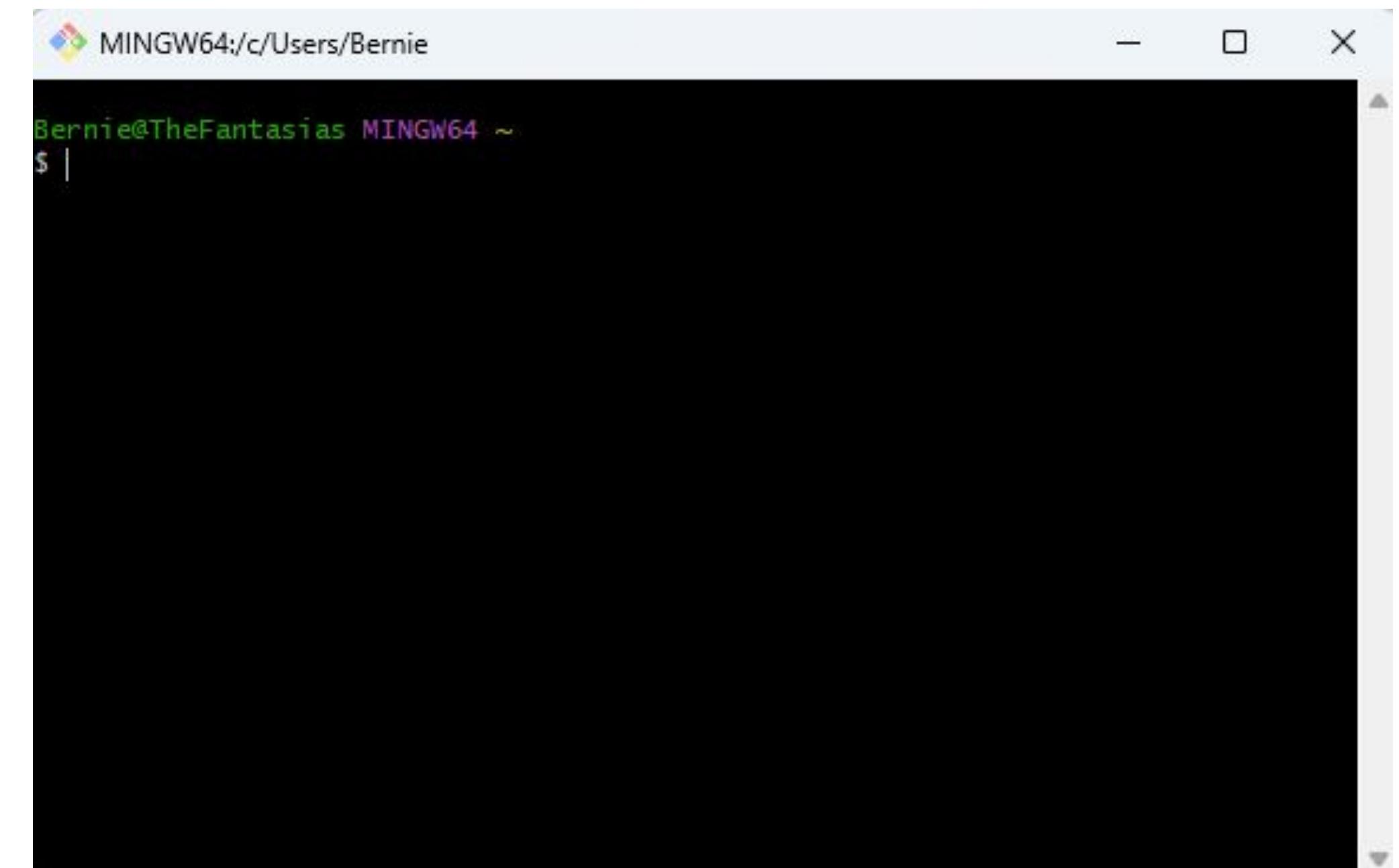
GIT BASH

使用Linux架構的終端機，為了 Git而生

安裝了Git的電腦裡，使用開始工具列去搜尋你會發現多了兩個東西出來：Git Bash和Git GUI，兩者都是協助你進行Git指令的工具。

我們等等會讓各位去熟悉Git相關的指令，並實際上傳一個資料夾到遠端存放庫上，這裡就先請各位去註冊GitHub帳號，如果有有了的話請幫我開著。

大概長右邊這個樣子。



專案架構，對 還行啦，這次才幾個檔案而已

我們只需要一個資料夾跟一個最基本的文字檔案就可以進行Git操作，在接下來的幾張簡報裡會一一帶到。

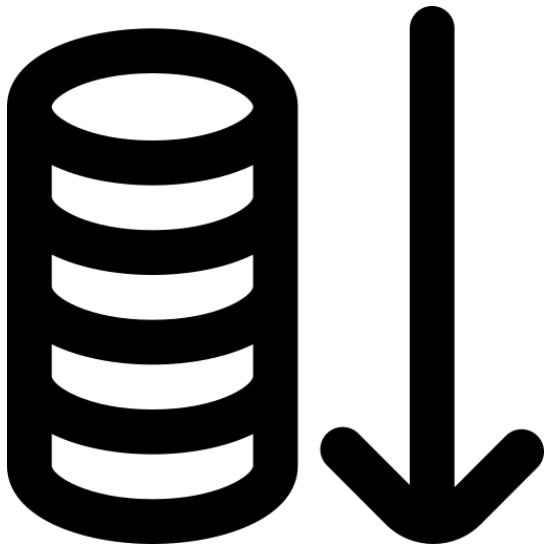
當然你也可以新增其他的東西，例如資料夾、程式檔案等等，甚至是圖片。

在Bash裡cd根位置到你的資料夾路徑，接著我們就可以開始了。

```
git-demo-project/  
|— notes.txt # 主要用來示範 Git 操作的檔案
```

INIT - 初始化

創建一個本地存放庫



第一件要做的事，就是讓整個資料夾能夠被Git所觀察，也就是被轉成存放庫，要這麼做就是單純的在Bash中使用git init這個指令。

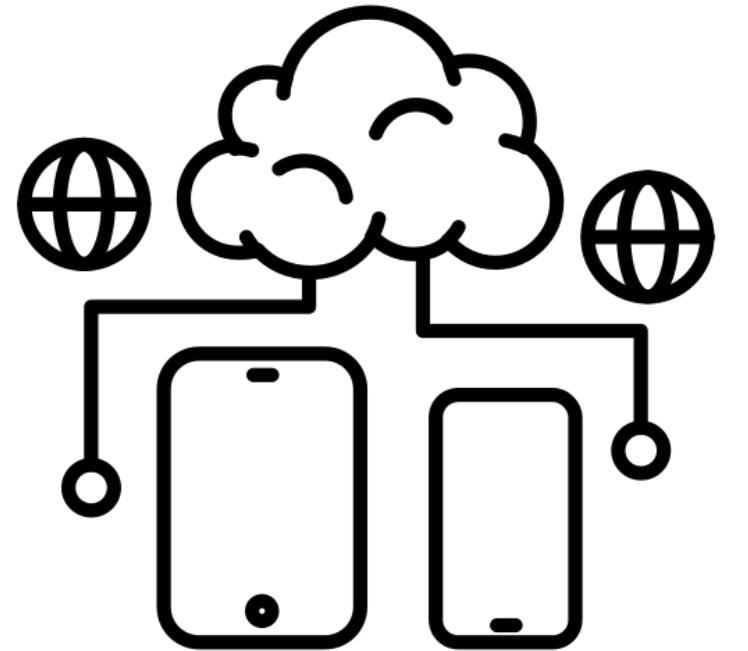
如果打開資料夾內隱藏的檔案檢視，你會發現他創建了一個.git資料夾，對，我們之後對這個存放庫所做的commit都會被存放在這裡，等待被發送到遠端。

你也可以用git status來看看當前目錄的追蹤狀態，不過應該是什麼都沒有。



REMOTE - 遠端連結

創建本地存放庫與遠端存放庫的連結



要跟GitHub進行連結，我們也要在網站上先創建一個存放庫，記得不要讓他創建README.md，不然會出現衝突而且我們不希望現在處理。

接下來，你可以複製遠端存放庫的網址，並在我們本地端的Bash裡輸入`git remote add origin "你的網址"`，他就會把兩者進行連結了。

你可以用`git remote -v`來確認當前遠端狀況。

COMMIT - 認可創建

先追蹤再認可，之後才是提取推送



我們要將資料夾內的檔案提供給Git進行追蹤，最直接了當的方式就是使用`git add <檔案名稱>`，或者是也可以直接`git add .`來把所有東西全部都追蹤起來。

每一次對資料夾內物品進行更改之後都要記得用`git add`來進行對資料的追蹤，隨後我們就可以用`git commit -m "認可訊息"`來建立版本，這個認可會被存在你的`.git`資料夾裡。

在這裡我建議各位設立一個commit訊息的寫法準則，像是在後方加入日期等等，會讓整體看起來更整潔也更好閱讀。

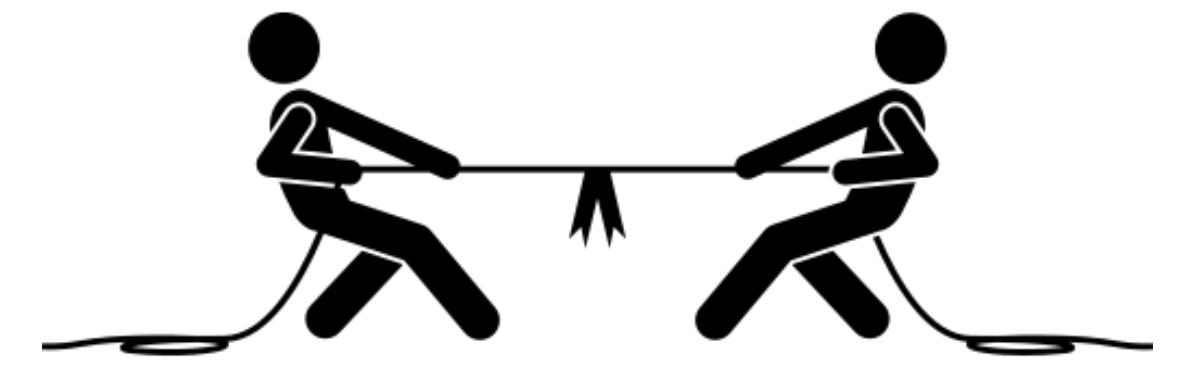


PULL/PUSH - 提取推送

與遠端資料庫的認可傳輸，同步版本

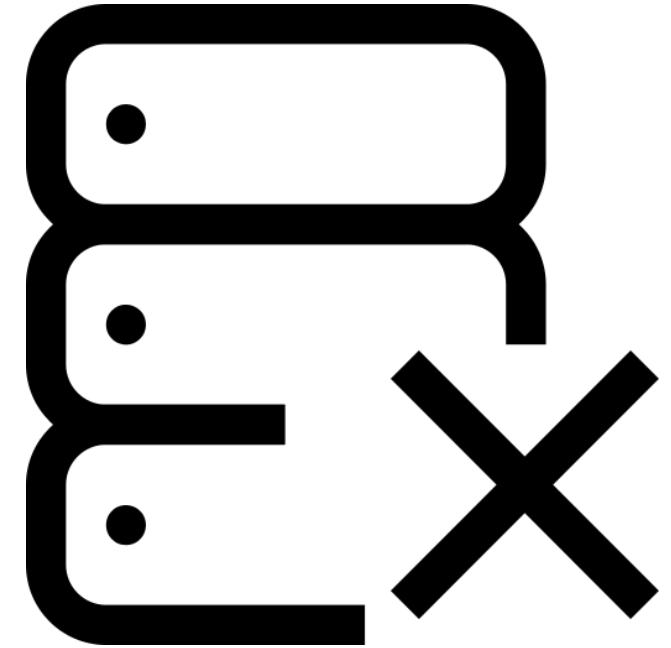
同步版本的順序是先拉再推，並且本地不應該與遠端有所衝突，當一方的版本有所更動時，我們進行以下的步驟：

- git push -u origin main (第一次推送專案時，-u是設定上游)
- git pull origin main (main也可以是你當前的分支)
- git commit -m “認可訊息” (進行修改之後)
- git push (由於已經設定上游了，因此只需要push就好)



.gitignore 和快取清除

一個特別的檔案，讓你忽略特定資料



在本地存放庫中添加一個叫做.gitignore的檔案會有特殊的效果，任何你寫在這個檔案裡的其他檔案/資料夾名稱會被git停止追蹤，並不會去嘗試列入變更。

這種作法能夠讓你很輕易地保護自己的祕密檔案，或者是各種安裝套件等等，但他有個容易被忽視的問題 – 已經被認可過的檔案並不會因此消失。

如果希望Git將當前新增的忽略檔案從存放庫中移除，可以用`git rm -r --cached "檔案名稱"`，如此一來在快取中也會清除剛該檔案。



基礎操作方式已經學會了

他其實就只是一個流程罷了



我們來總結一下到目前為止所提到的部分，Git的使用順序如下：

1. git init → 初始存放庫。
2. git remote add origin “你的網址” → 連結遠端存放庫。
3. git add . → 將資料夾變更進行追蹤。
4. git commit -m “認可訊息” → 在本機建立版本認可。
5. git push -u origin main → 初始上游關聯。
6. 之後定期的pull/push。

如此這般，一個簡單的Git流程就完成了。



Google Developer Groups
On Campus • National Kaohsiung Normal University

4

分支與共用

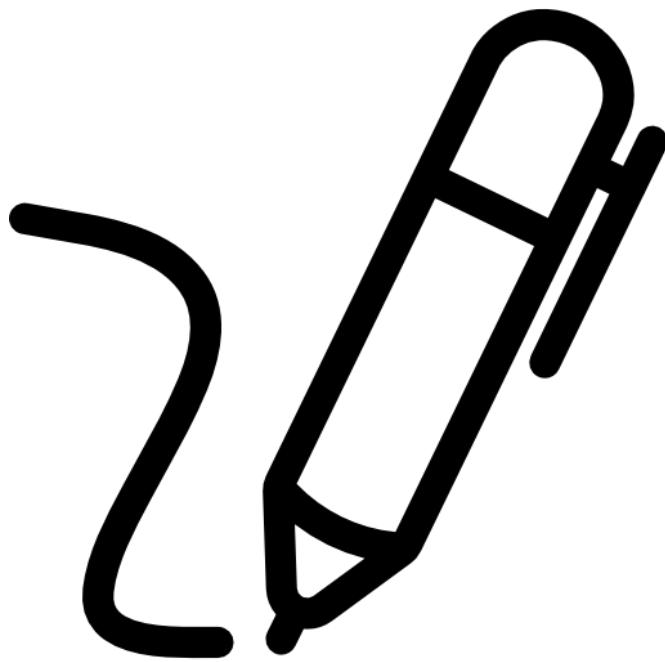
一棵樹所延伸的枝節

```
filterByOrg = study.lead_organization === filterByOrg .  
filterStatus = filterByStatus ? study.status === filterByStatus : true  
if (filterByOrg & filterStatus) {  
    return study  
}  
  
function filterStudies({ studies, filterByOrg ,  
    filterByStatus }) {  
    const filteredStudies = studies.filter(study =>  
        filterByOrg || filterByStatus  
    )  
    return filteredStudies  
}
```



實驗性內容

分散風險的原則



進行專案往往不是一條路通到底，尤其是在進行大型應用程式開發，我們會有很多獨立的模組需要去添加，而要是我們將全部的變更給放上main，有時可能會遇到不樂見的問題發生。

與其經常對main進行回溯，倒不如一開始就創立新的分支來協助管理，在不同分支上進行的變更都會是獨立的，也就因此可以避免資訊汙染發生。

在多人開發時，這樣子的做法幾乎變成了必要的，因為每個人對於一件式的作法不盡相同，而且這樣也可以協助同時開發新內容。



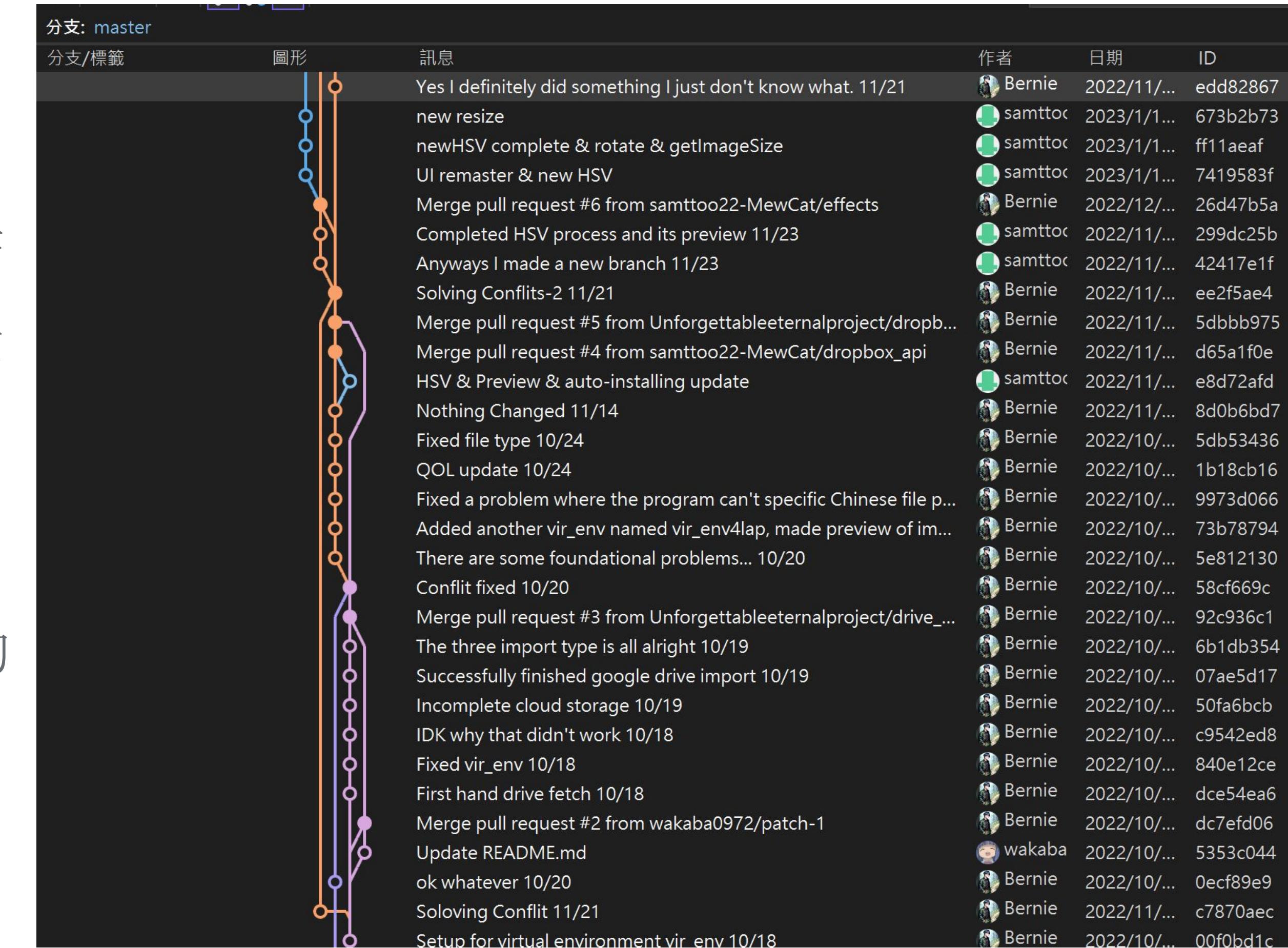
真正的樣子

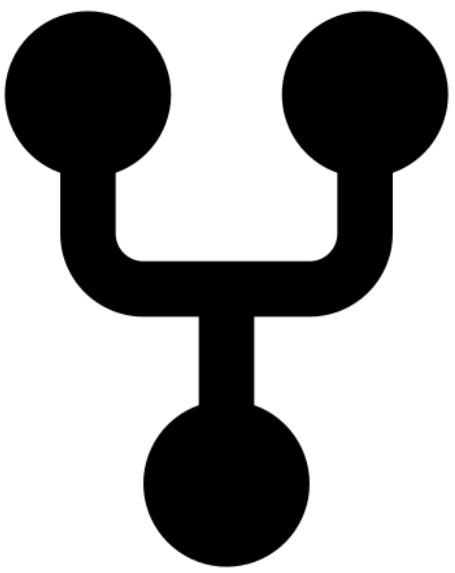
一顆永無止境的樹

分支樹多起來那可就複雜了，在右圖中的每個點都是一次認可，而根據不同分支的分分合合他最終看起來就類似這樣。

子分支基於一個根部 (Root) 而生，且如同真正的樹枝一般 – 他可以持續擴展。

接下來帶各位實作一下分支的使用。





BRANCH → CHECKOUT

分支的創立與跳轉

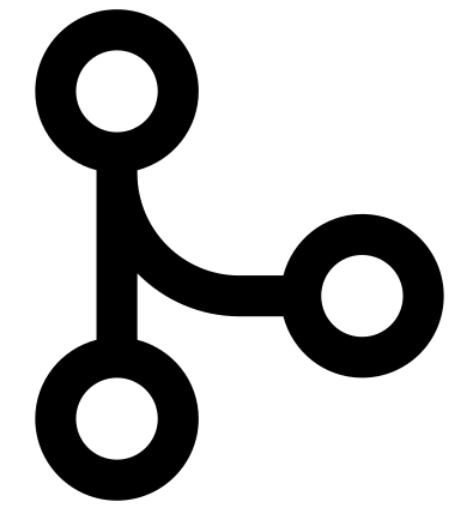
使用git branch “分支名稱” 可以建立一個由當前分支所延伸出去的新分支，再使用git checkout “分支名稱” 來跳到該分支上去（也可以直接用 git checkout -b “名稱”）

分支間是獨立運行的，直到merge之前都不會互相影響，因此你可以想像是一個平行時空的概念，在不同分支上進行的操作只會留存在該分支當中。

新的分支在有至少一次認可紀錄後就會被保護，防止被誤刪。

大珠小珠落玉盤

-d / -D 的差異



一般來說分支的結尾是合併入父分支內，但偶爾也會有被廢棄的情況，或者是在合併後失去作用，這時候可以用`git branch -d "分支名稱"`去將其進行刪除。

但要注意有兩點限制存在，第一：不能刪除當前所在的分支，第二：分支內不能有未經處理的認可，但如果無論如何都希望刪掉呢？

使用`git branch -D "分支名稱"`可以強制刪除，但是這是很危險的方法。

不需要代價的反悔

Git的好處就是非常彈性

如果不小心刪除了重要的分支，Git提供你方法來復原他，並且所有的認可紀錄也還會存在。

在刪除分支的時候他會連同分支的ID一起告知你，只要用該ID去創建一個新分支，他就等同於復原你原先刪除的分支。

但是最好還是不要誤刪啦...

```
wishingsoft@SMC MINGW64 ~/Desktop/Test (master)
$ git branch -d "Another"
Deleted branch Another (was 3d007f6).
```

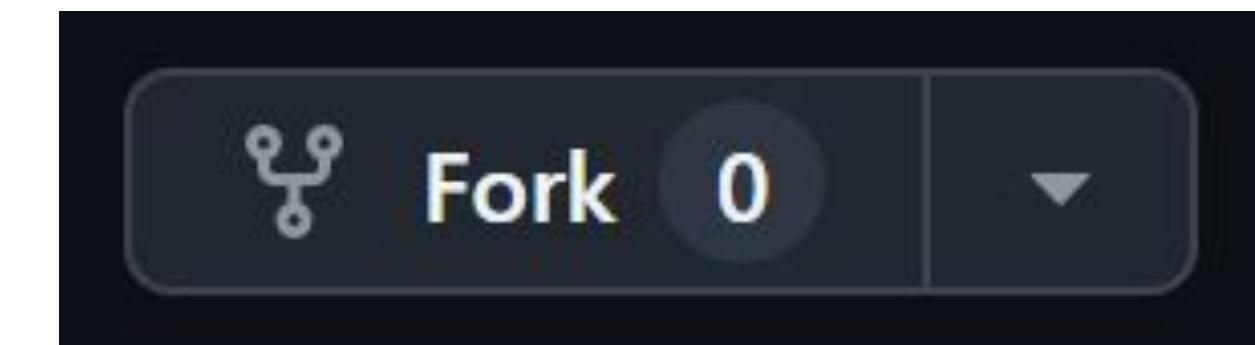
```
wishingsoft@SMC MINGW64 ~/Desktop/Test (master)
$ git checkout -b Another 3d007f6
Switched to a new branch 'Another'
```

別人的分支 對，這個 fork就是叉子

GitHub有Collaborator的機制，也就是你可以邀請其他人到你的專案中。

但是，還有另一種方式可以跟別人協作，也就是著名的fork功能，這東西會讓你複製一份別人的存放庫為自己所用。

你對其進行的變更，都可以透過PR來貢獻。



Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Required fields are marked with an asterisk ().*

Owner * Choose an owner / **Repository name *** Digital-Subcurrent

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

This is the game project for our Unity Game Design final.

Copy the master branch only

Contribute back to Unforgettableeternalproject/Digital-Subcurrent by adding your own branch. [Learn more.](#)

Create fork

貢獻者的一份子

你願意接受我的心意嗎？

所謂的PR就是提取請求 Pull Request 的縮寫，當你fork他人的存放庫，並且在自己的分支當中進行變更之後，你可以透過PR的方式去向原有使用者提出貢獻請求。

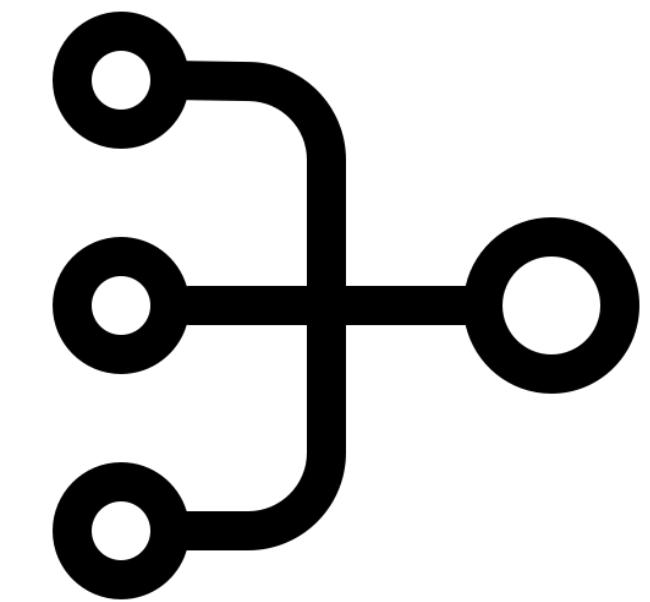
在所有審核通過之後，你的變更就可以被併到原本的存放庫內，並且你會被作為貢獻者的一部分被標註起來。

對社會產生貢獻的感覺如何？



MERGE - 分支合併

把失散的孩子給帶回來



學會了怎麼創建分支，現在是時候學學怎麼將創出去的分支給合併到主要的main當中了（當然你也可以把分支合到另一個分支之內），但這其實是門學問...

預計沒有任何衝突的情況下，我們可以直接使用git merge “分支名稱”（使用指令的位置要在合併的目的地上），幸運的話所有分支中的變更都會被帶到目的地裡面。

但是世事難料，誰知道在分支的運行當中會不會出現有衝突的情況呢？



Developer Student Clubs
National Kaohsiung Normal University

5

衝突解決

Diffmerge

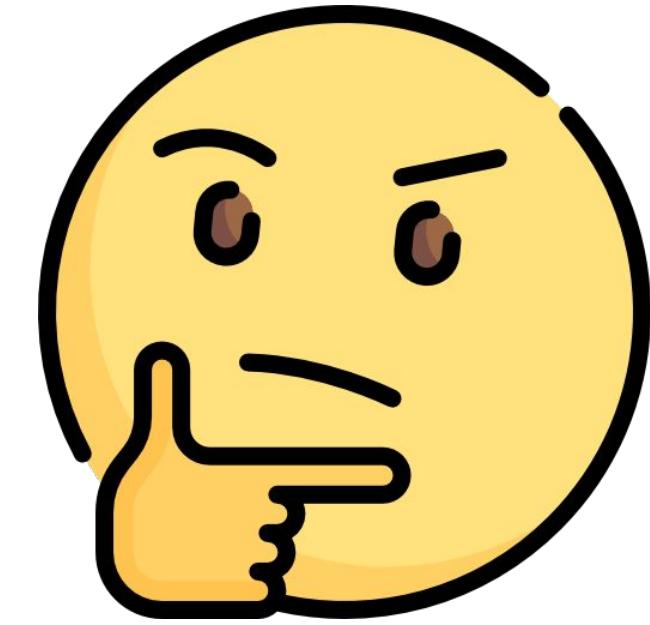
```
    study = filterByStatus ? study : study.filterByStatus(status)
    status = filterByStatus ? study.status -- filterByStatus : true
    if (status &gt;= matchStatus) {
```

```
        return filterStudiesIf(studies, filterByOrg)
    } else {
        return studies.filter(study =>
```



設定一個情境

我們先來設想一個可能的情況



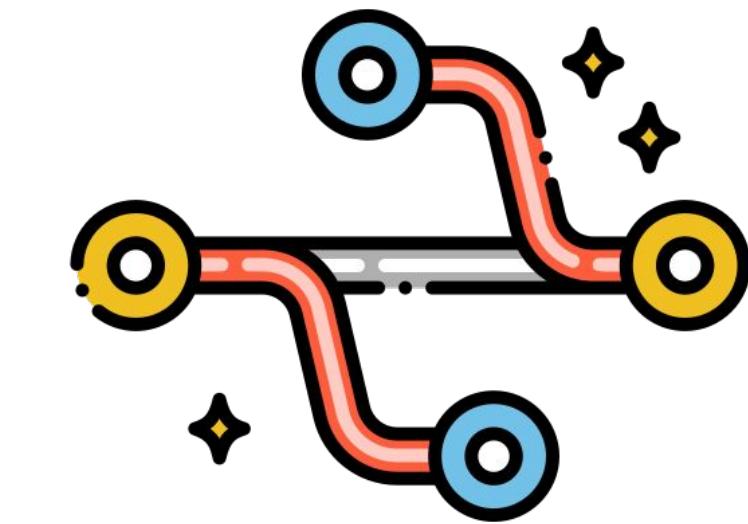
通常自己在開發專案時，比較不常發生分支認可衝突的情況，因為一般都是一个模組一個模組的開發（我不知道你們的模式啦，但我自己是還好）。

但在多人同時開發的時候，一些主要檔案可能就會因為環境不同而被迫進行更改，導致在不同分支中修改了同一個檔案，這下就沒辦法自動完成合併了。

那麼該怎麼辦呢？要動到我們的衝突解決程序了。

分支動盪

本末倒置，但這是為了演示



我們要在main跟剛剛創建的新分支中更改同一個檔案，最簡單的方式就是直接使用那個文字檔，在一個分支中寫一些東西，認可，到另一個分支寫一些東西，認可。

接著，回到原本的分支裡，在同樣的位置上寫些不同的東西之後，認可。

這樣在兩個分支當中就會出現分歧的檔案，此時想要嘗試merge就不是這麼容易了，他會把出現衝突的部份給列舉出來，等待我們進行處理。

DIFF - 檢查衝突文件

好在socket是Python內建的函式庫

在發現沒有辦法直接合併時，可以使用git diff
這個指令來尋找問題的位置。

他顯示的方式如右圖，會跟你说衝突的檔案，以及
分別在HEAD與來源分支的更動。

```
$ git merge Another
Auto-merging Thing.txt
CONFLICT (content): Merge conflict in Thing.txt
Automatic merge failed; fix conflicts and then commit the result.

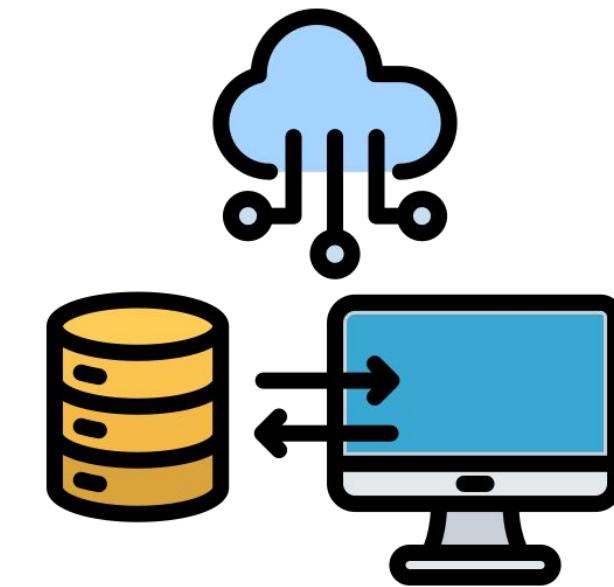
wishingsoft@SMC MINGW64 ~/Desktop/Test (master|Merging)
$ git diff
diff --cc Thing.txt
index 9397aa8,1530719..0000000
--- a/Thing.txt
+++ b/Thing.txt
@@@ -1,3 -1,3 +1,7 @@@
    But why do we do this?

- This is NOT for a conflict.
- This is a conflict.
++<<<<< HEAD
++This is NOT for a conflict.
+=====
++This is a conflict.
++>>>>> Another
```

而衝突的解決有直接更改相關檔案，或者是使用一
些額外的解決工具。

本機 <-> 來源

只能保留一個，要不然就是兩個都改



使用git diff之後，你會發現檔案被更改了，跟剛才一樣有HEAD跟來源分支，這裡就是讓你選一個留下，要不然就是再改成新的樣子。

```
wishingsoft@SMC MINGW64 ~/Desktop/Test (master|MERGING)
```

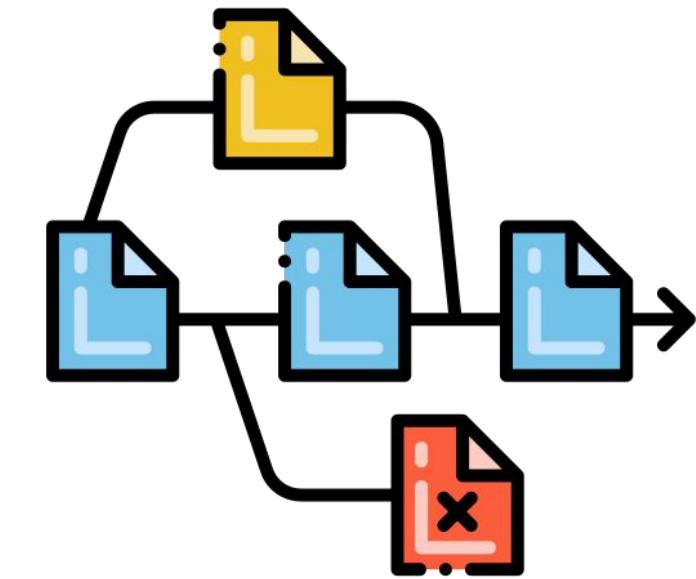
現在的Bash後方會顯示目前是合併的過程，等待你給出一個回覆。

在完成想要的衝突解決之後，將更改過的結果提交認可，合併過程應該就會自然完成了。

簡易的處理就是這樣，但這也是只有一個檔案的情況，多個檔案可能更加複雜。

多合一

一次合併多個不同的分支



假設你從master分出了四五個不同的分支用來開發不同的功能，並且你希望能夠一次全部合併的話，Git本身的架構是支援的。

實際上，Git提供很多不同的合併策略，多合一最直覺的方式就是`git merge <branch1> <branch2> <branch3>`這樣的寫法。（多合一千萬要避免衝突，不然會很死人）

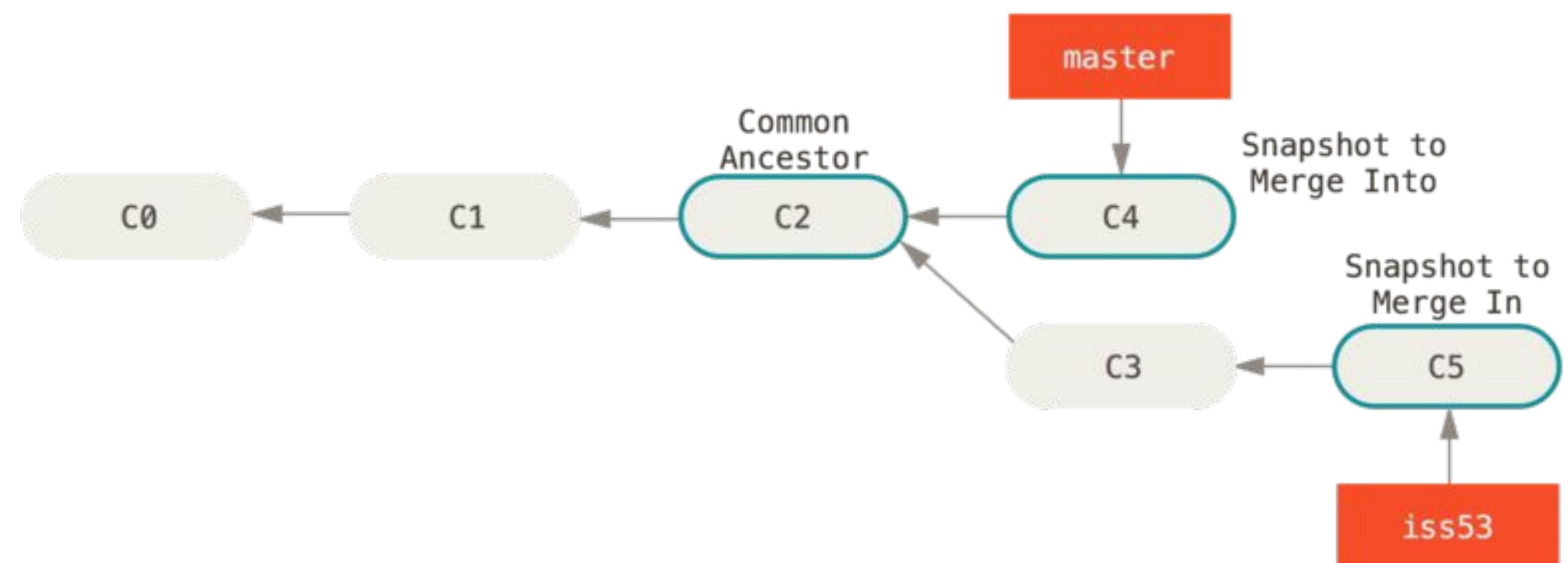
這樣子的策略被稱作octopus，除此之外還有ours，subtree等等，但我們就先不提了。

合併的意義

就是節點與節點之間的互動罷了

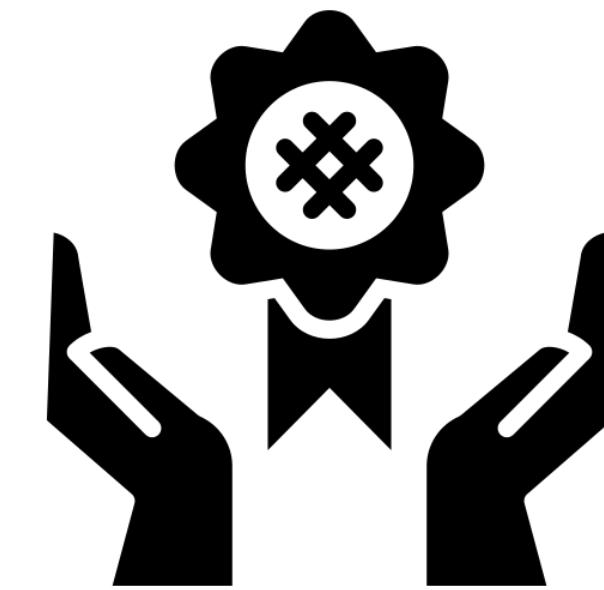
合併的依賴是所謂的共同祖先，也就是最完整的版本，而合併的資料則是每個分支上最新的快照。

這些最新的快照包含了當前分支的所有認可，這些認可進度可能在其他分支之前或之後這樣，因此才需要進行合併。



進階課程預告

如果有興趣的話再來試試吧



既然今天的課程是Part One, 那麼在不遠的將來我們也會有一個Part Two存在, 只是不同於今天的簡單指令, 在下一次的課程中我們會以更專業的角度來使用Git。

可以先透漏的是, 像是git reset, git rebase, git cherry-pick等等指令都會用到, 以及真正的Git工作流的介紹這些都會留到下一次。

因此, 如果希望學到更多更有用的Git, 下一次的課盡量不要錯過喔。

6

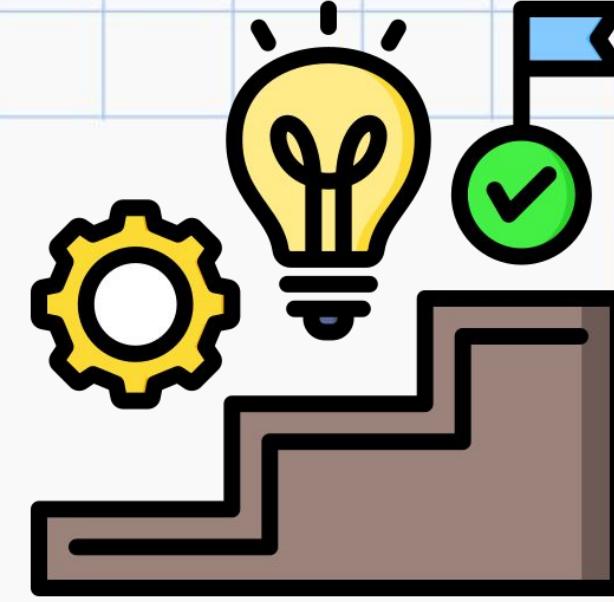
Git與工作流

一套流程，每個人都應該遵守

```
const filterByOrg = study => study.organization === filterByOrg;
const filterStatus = filterByStatus ? study.status === filterByStatus : true;
const filterPatchStatus = filterPatchStatus ? study.patchStatus === filterPatchStatus : true;

function filterStudies({ studies, filterByOrg = null, filterByStatus = null, filterPatchStatus = null }) {
  return studies.filter(study => {
    if (filterByOrg) {
      return filterByOrg(study);
    }
    if (filterStatus) {
      return filterStatus(study);
    }
    if (filterPatchStatus) {
      return filterPatchStatus(study);
    }
    return true;
  });
}
```

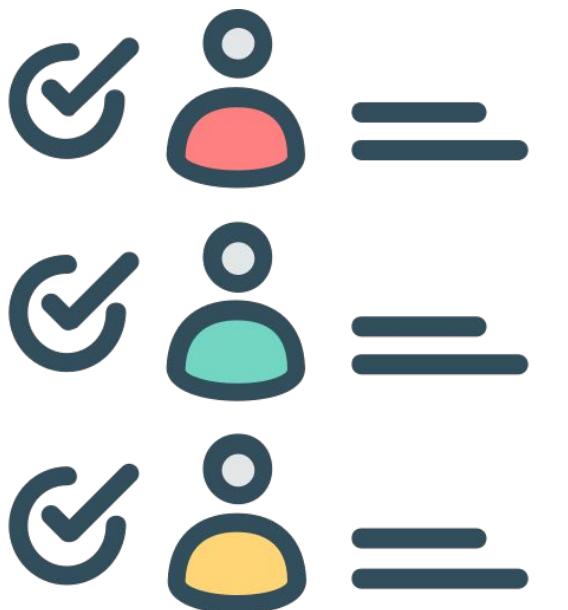
一個完整的開發程序



大部分狀況下分支這種東西我們只會在需要的時候創建，原因很簡單 – 因為全都放main上面就好啦，反正版本有紀錄。

但在實質應用上，這樣反而會讓管理上變得困難許多，尤其是在你沒有認真地撰寫每一次Commit的訊息時。

與其這樣，我們不如制定一套方式來幫我們做管理。



點題時間

之所以叫做工作流，是因為他是工作的流程

從上次的課就有提及到所謂的工作流 (也是我們這堂課的名稱)，實際上就是一套SOP，讓你在進行專案的開發時可以更有效率地去處理版本的流動。

它並沒有規範你要怎麼去寫程式，頂多是給你一些模板讓你去嘗試。

在工作流的部分其實還有分為Git / GitHub / GitLab等等不同體系的做法，在這裡我們以Git的工作流來做介紹，他其實比你們想得都還要簡單理解。



代步 (develop)

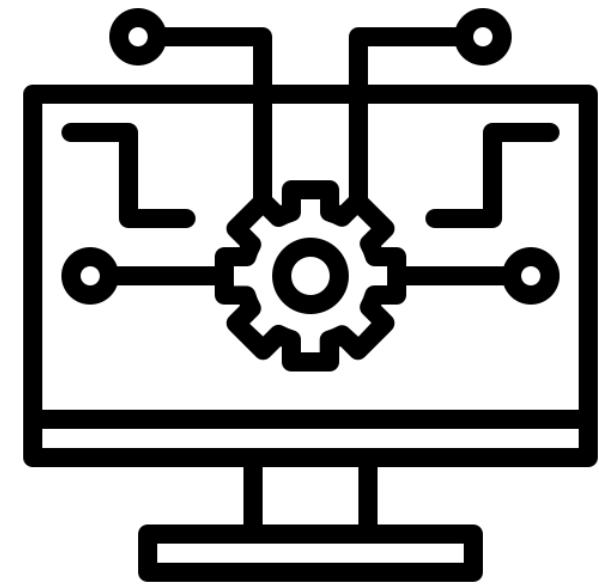
第一個也是最重要的主要分支

很多人會以為main就是用來做主要開發用的分支，但這麼想其實不太對，main的定位可能更像是一個穩定版本，一個展示給他人看的樣子。

在工作流的第一部分，我們通常會使用一個develop分支來協助我們進行開發 – 也就是幾乎每個更新都放在這裡，只有最完善的版本會被推回到main。

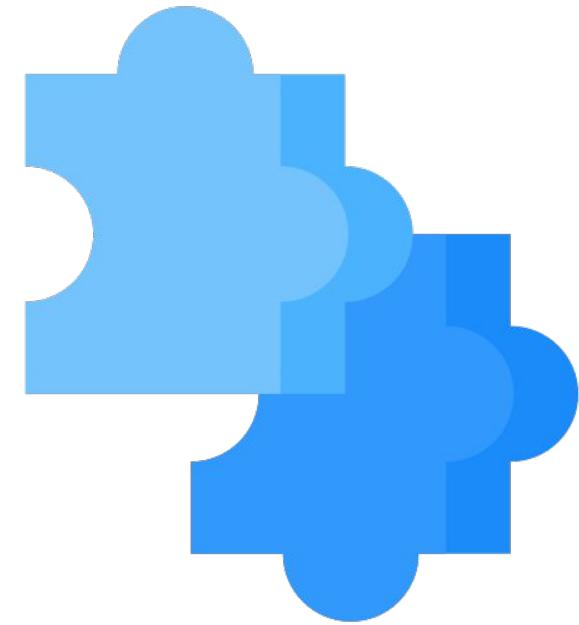
而由於在merge之後該分支並不會消失，因此我們可以繼續在上面進行修改。

直至專案完成之前，這條分支都會一直跟著。



新功能就要外擴 (feature/xxx)

功能分支，也是最容易被遺忘的一環



這樣看起來，所有東西都新增在develop上就好了？並不是，因為他只是用來做整體開發跟維護的，處理的是整個系統，如果要進行實驗性功能的常識，我們就要開feature。

feature/xxx是很常見的作法，每當你要為系統添加/修改/移除功能、並且該功能還沒有辦法被完善併入develop時，我們就應該為它獨立開出一個分支。

但基於很多人都直接在develop上進行開發了，因此很容易被忽略，請牢記在心。

熱修 (hotfix/xxx)

快速處理重大問題根源



hotfix的用途是在已發布版本出現某些不被期望的問題時對其進行處理，因此大多情況下是直接從main中分出來的，用來做緊急修正。

從hotfix併回去main的合併過程我們一般會叫做Patch，也就是補丁，因為概念上就是幫現行版本加上快速的修正，以維持檔案完整性以及使用者體驗。

有時候我們會在現行的熱修上疊另一層的熱修，可以的話盡量避免這種狀況。

預發布 (release/xxx)

在進到大眾的眼中前做最全面的管控



實際上，在從develop進到main之前我們有時還會做一個release分支，來協助進行正式版本發布前的最後修正，在這個階段通常也是修最多的。

預發布的存活時間一般來說不會太長，因為只是為了要確認版本的完整，很多開發者甚至會去省略這一個部分，但有這個步驟也算是增添一條防線啦。

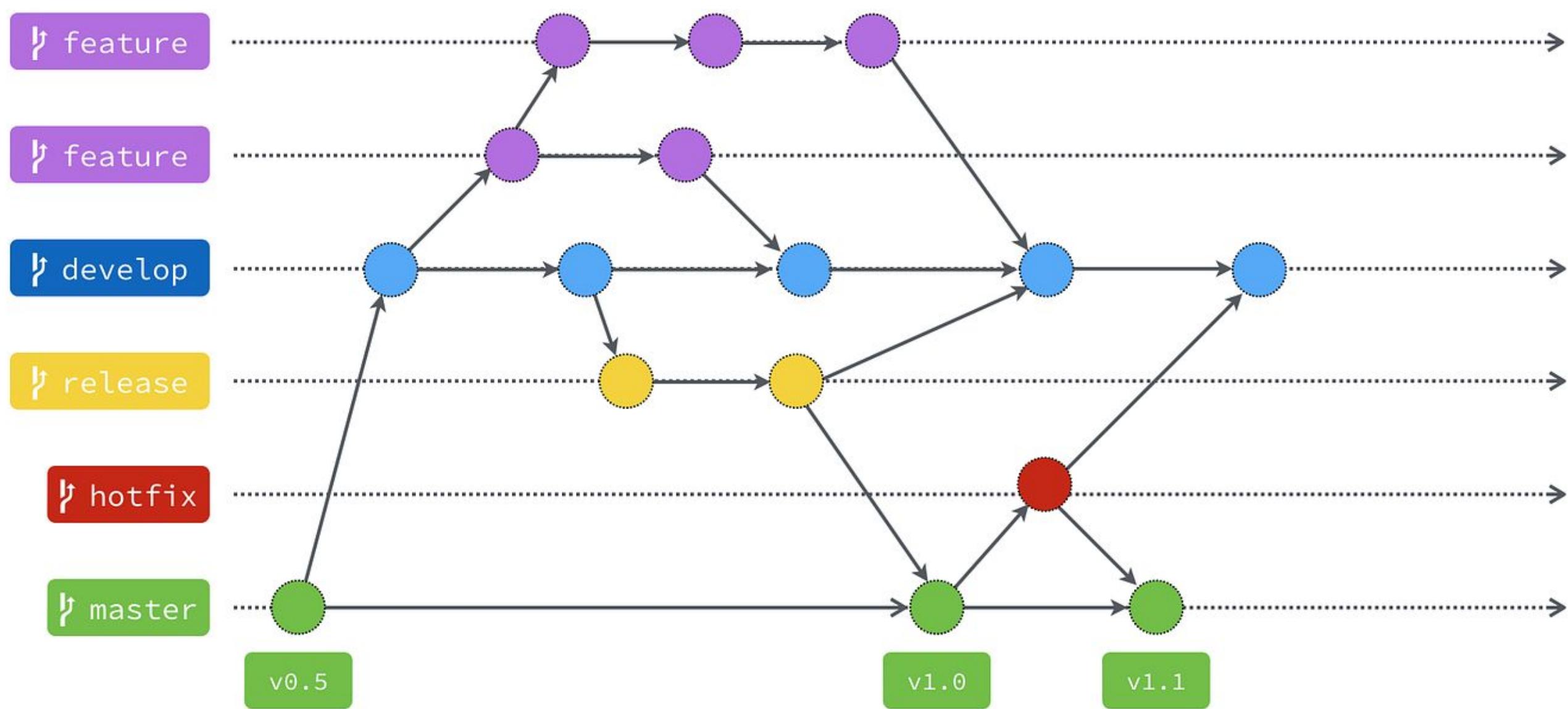
要注意的是，一個完成的release除了併到main裡面之外還要順道併回develop喔。

完整工作流

實際在版本控管中的樣子

右圖就是整個Git工作流的樣貌，你也可以看得出來每個分支之間的協調與工作是長得甚麼樣子。

最活躍的非develop莫屬，與feature相互成長，並且透過release和hotfix來完善main本身，最後才有完整的成果。



7

進階常用指令

允許更多對於認可與版本上的操作

```
const filterByOrg = study => study.lead_organization === filterByOrg;
const filterStatus = filterByStatus ? study.status === filterByStatus : true;
const matchStatus = filterStatus || !filterStatus;

function filterStudies({ studies, filterByOrg, filterByStatus }) {
  return studies.filter(study => filterByOrg(study) && matchStatus(study));
}
```

更高操作性

不要小看Git所能夠提供的效益

現在看起來Git和GitHub所能提供的無疑是同步、版本整理、分支、多人協作等等的功能，已經很好用了，而且也是絕大多數人會用到的功能，但這樣就夠了嗎？

這還遠遠不是Git的能耐，馬打馬打。

精通Git指令的話，你會知道這東西到底可以做到什麼離譜的效果。

而這就是今天這堂課要教的，從簡單的修補，到直接整體回撤。



COMMIT -AMEND - 認可修改

給你一個亡羊補牢的機會

認可是認可了，但在實際PUSH之前他單純就只是一連串的資料游離在你的電腦裡面，如果今天不小心送出了認可，但還有東西沒加進去的話呢？

與其為了幾個檔案創建新的認可，我們大可以直接用git commit -amend去修改之前的那個，並且把現在暫存區內的內容給一併加進去，這樣就變得簡單多了。

不過要注意的是，如果上一個認可已經被PUSH到遠端，這麼做可能會跟你想的不太一樣，因為他會需要幫你創立一個新的commit，並跟你的舊commit給合併在一起。





STASH - 暫存起來

一個暫用的儲藏空間

一個很好用，但卻很常被忽略的指令是git stash, 這個指令能夠將當前工作去尚未認可的修改進行暫存，用處可多著呢。

被暫存的內容會儲存在當前的分支，因此在進行分支間的切換時，該暫存狀態不會被清除，之後再回來的時候就可以透過git stash pop就可以回復檔案。

這東西容易跟暫存區(Stage)做混淆，之後會更詳細的講解。。

TAG - 創造版本標籤

給予標誌版本一個好認的標籤

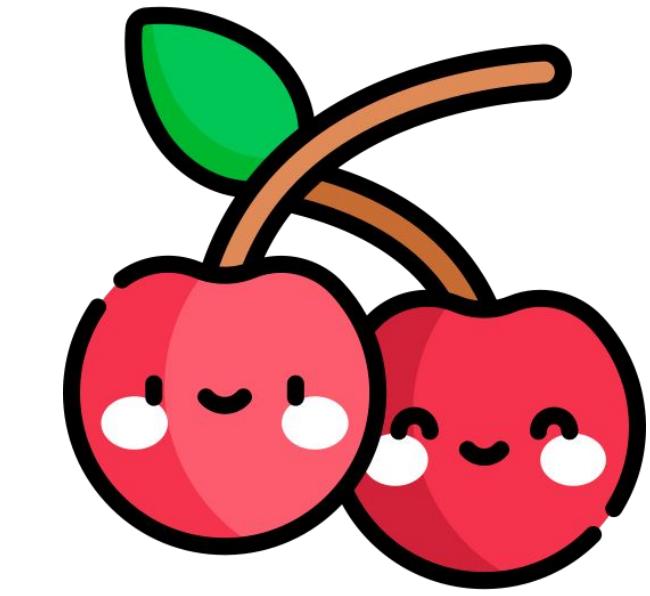
在追蹤或者需要編輯認可的時候，我們會需要知道該認可的編號，但那有點太難記而且每次都要額外查詢，倒不如直接為一個認可上一個版本標籤。

git tag {你想宣告的版本} 的使用在進行Git工作流時格外有用，因為你可以將main上的穩定版本加上一個標籤，讓其他人也可以比較好讀。

如果有想要用GitHub做release的，這樣子的標籤化是相當重要的，不過這個以後有機會再提吧。

CHERRY-PICK - 萬中選一

從別的分支拉一個能用的認可



Cherry pick是一個很強大的功能，他能夠讓你從某一個分支中挑選特定的認可並加到當前的分支當中，因此該認可裡面的檔案也會一併拉過來。

用法是git cherry-pick {認可編號}，他會去瀏覽所有分支中的認可，找到對應的並拉到工作區裡，要注意的是這可能會造成衝突，不過通常會比分支合併時還要好解決。

他拉過來的認可會幫你自動commit，因此也不需要額外去做add和commit。

小型總結

來快速整理一些新增的這些指令

到目前為止我們介紹了四個新的，特別用途的指令，分別如下。

- -amend: 加在commit後面，可以修改最後一次認可 (最好是還沒PUSH的狀況)。
- stash: 把修改放在暫存區，以便之後調動，適合用在跨分支的情況。
- tag: 將認可加上標籤，方便辨識，也可以做為版本號使用。
- cherry-pick: 將別的分支上的認可挑到目前的分支上，並應用變更。



接下來呢，我們要來點更加進階的認可操作。

8

重新整理

從Reset到Rebase

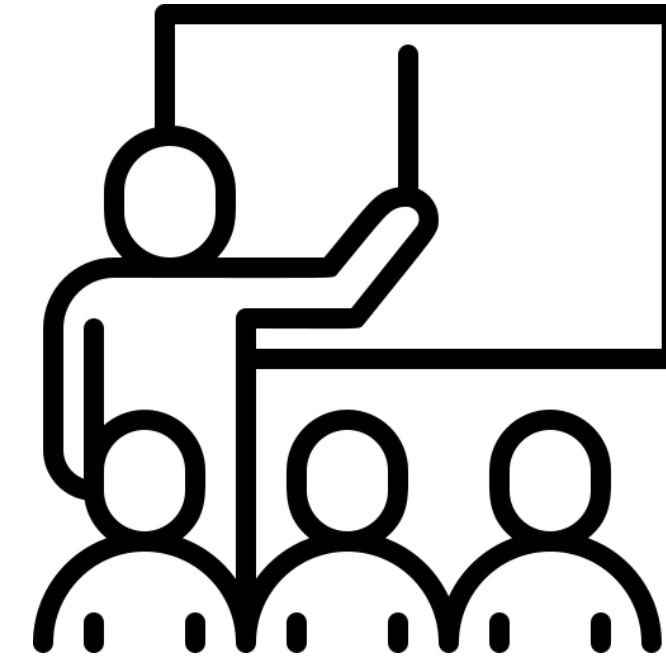
```
filterByOrg = filterByOrg ? study.lead_organization === filterByOrg : true
filterStatus = filterByStatus ? study.status === filterByStatus : true
const matchStatus = filterStatus || !filterStatus

function filterStudies({ studies, filterByOrg, filterByStatus }) {
  return studies.filter(study => {
    if (!study.lead_organization) return false
    if (!study.status) return false
    if (filterByOrg && study.lead_organization !== filterByOrg) return false
    if (filterStatus && study.status !== filterStatus) return false
    if (matchStatus) return true
    return false
  })
}
```



HEAD、Stage和Workspace

在進行接下來的講解前，這個觀念很重要



HEAD的部分之前有提過了，他是一個對於當前位置的指標，在git中，可以使用^符號來表示前調，或者用~n來直接表示跳轉。

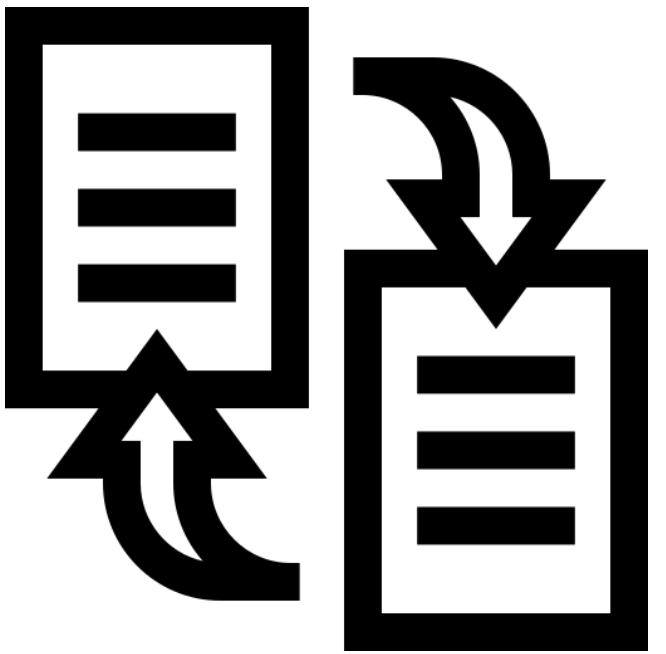
Stage的概念就是你使用git add之後將檔案給丟進去的「暫存區」，與Stash的暫存不同，而且很容易搞混，請注意。

最後就是Workspace，也就是你當前工作的範圍，我們就叫他工作區。



歷史可以被改寫

很多人在意的是要如何復原版本



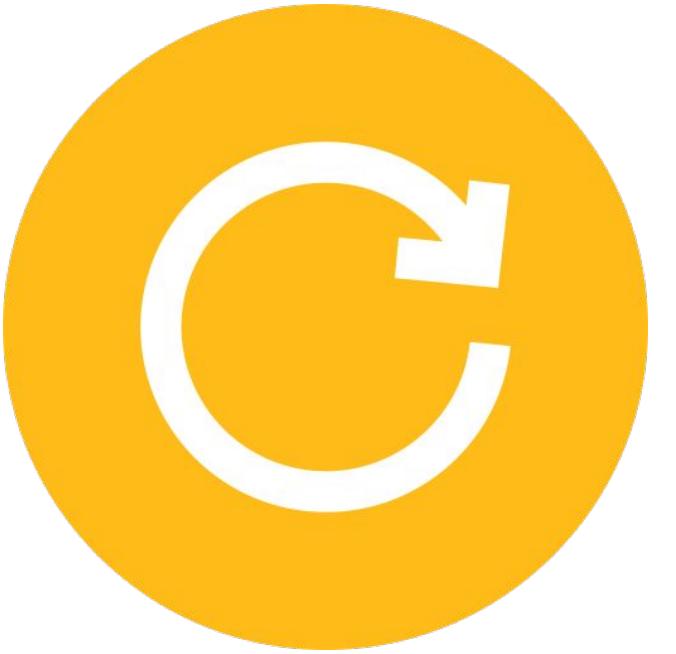
Git裡面另一個，也算是最核心的功能，非屬回溯本身不可。

當然，你有追蹤版本，也就要能夠有辦法定能夠回溯到某一個時間點才行。

Git提供了幾個方法，最常用也是最好用的就是git reset本身，而這個指令又有三種強度之分：
--soft、--mixed和--hard，這三種用途不同。

一般來說預設的方式是用--mixed。





重置(軟、混合、硬)

丟回之前所進行的變更區，或者把它們殘忍的刪除

軟重置，也就是git reset --soft {認可編號}的效果是重新回到那個認可，並將該認可之後做的所有變更給回撤到暫存區裡，好好地待著。

混合重置，git reset --mixed {認可編號}的效果介在軟跟硬之間，他會把變更給保留起來，但是直接丟回工作區內，也就是不會幫你做git add。

最後，也是最可怕的硬重置，是在你回溯認可的同時把所有的變更全部都刪除，也就是東西都會直接不見，因此是非常殺手鐗。

REFLOG - 調度所有歷史

Git記得比你還清楚



用硬重置把認可給抹除之後，東西就真的找不回來了嗎？概念上來說是的。

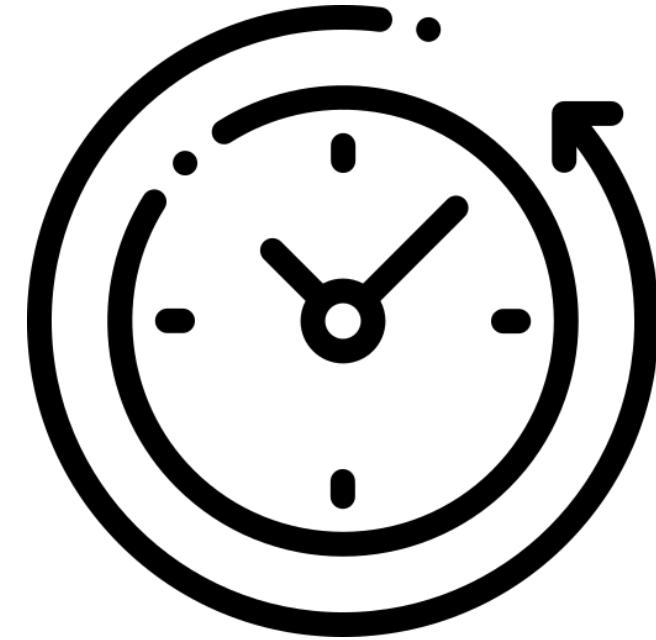
但是嘿，我們有一個神奇最終手段來讓你還原，也就是大名鼎鼎的git reflog，這裡用SHA-1值紀錄了你歷史中的每一次認可，包含被刪除的那些。

因此透過git checkout {認可編號}的方式，我們能夠去找回這些遺失的認可，並順帶把變更給找回來，不過早知如此，何必當初呢？



REVERT - 完全相反的提交

一種另類復原的方式



與其回推commit本身，我們也可以試著用另一個方法去做版本的復原，也就是使用git revert {認可編號} 去做一個完全相反的認可。

沒錯，你新增東西他就刪除，你刪除東西他就加回來，就是名副其實的一個完全相反的認可，他不會動到你的commit順序，也不會出現什麼衝突。

如果今天只是要做一些小小的修正的話，用這種方式也無妨。

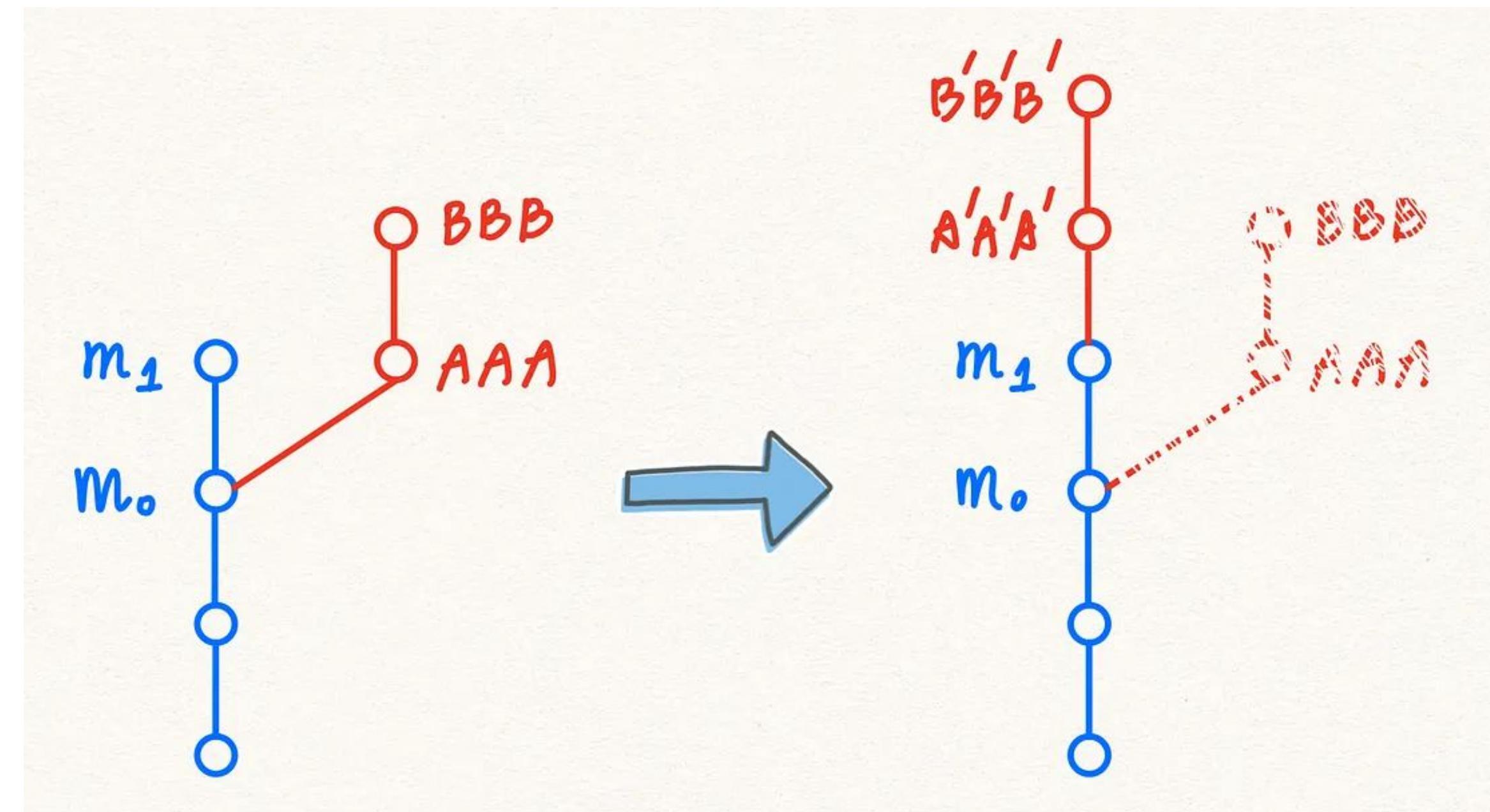


定義基準

一種另類的合併方式

接下來的部分比較麻煩，因為我們要來介紹所謂的Rebase。

Rebase的意義就是重整，小至認可的排序方式，大至分支的接頭都可以做到。



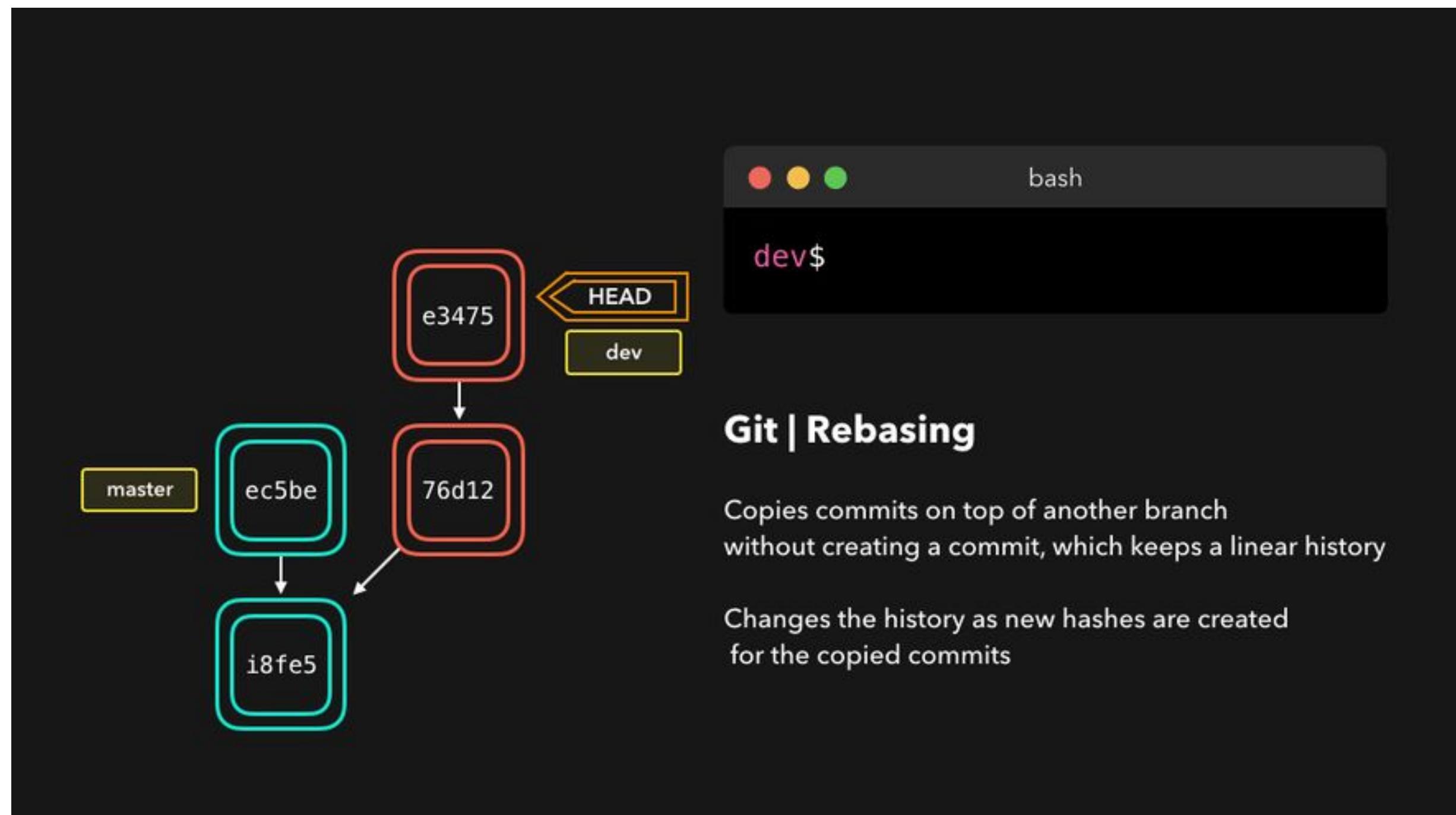
接下來我們根據他的功能來一一做介紹。

調整根節點

把分支的原點給移走

跟使用git merge {分支名稱}有些相似
, git rebase {分支名稱}的使用會把這條
分支的原點給接到指定的分支上。

記得跟merge反過來，這裡是把自己給接到目
標分支上，然後當然也會有可能出現衝突的情
況，這點與merge的處理相同。

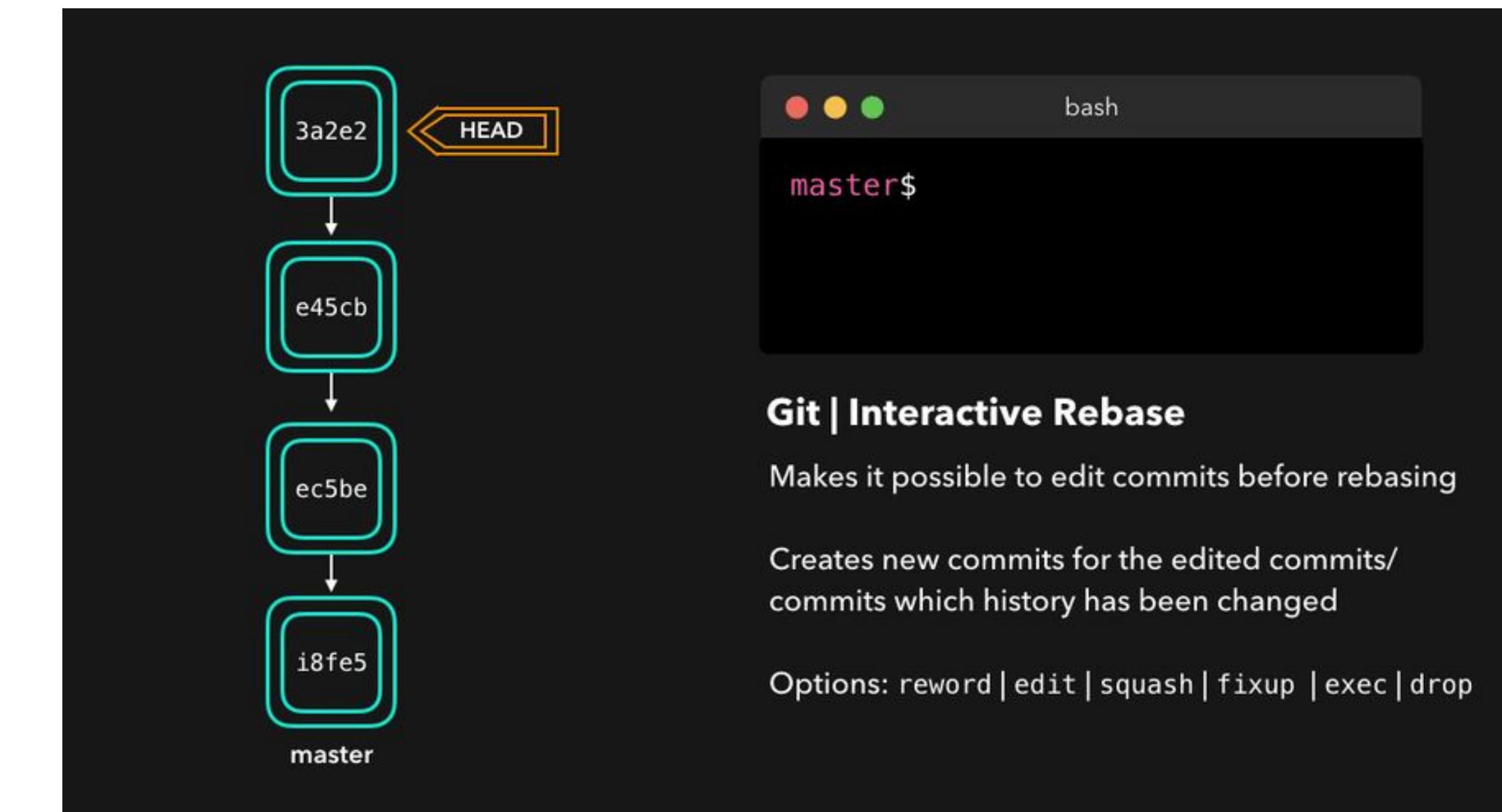


手動改認可

這個會用到VIM編輯器，為方便起見就不帶了

另一個rebase的重要功能就是他的VIM介面，
透過使用git rebase -i {認可名稱}，
他會去列出該認可之後的所有認可。

你可以透過VIM去對這些認可進行處理，包含合併、刪除、調換位置等等。



Rebase會整理成一個認可並提交。

放棄

他其實會提醒你啦



跟merge一樣，做rebase的時候很容易遇到衝突，這個時候無疑就是要繼續進行解決衝突的程序，並將完成後的結果透過`git rebase --continue`去提交。

然而，有時候自己都不知道自己的在做什麼了，而且VIM編輯器如果不熟，也很容易造成你不知道怎麼修改的後果，這個時候就果斷放棄程序吧。

`git rebase --abort`可以直接中止當前的程序，並且把一切都回復原貌。





Google Developer Groups
On Campus • National Kaohsiung Normal University

9

次要存放庫

很好用的功能，但很多人不知道

```
const filterByOrg = study => study.lead_organization === filterByOrg
const filterByStatus = filterByStatus ? study.status === filterByStatus : true
const filterByMatchStatus = filterByMatchStatus ? study.match_status === filterByMatchStatus : true

function filterStudies({ studies, filterByOrg, filterByStatus, filterByMatchStatus }) {
  return studies.filter(study => filterByOrg(study) && filterByStatus(study) && filterByMatchStatus(study))
}
```



一個小朋友

專案和子專案的概念



有時候我們在做一個專案的時候，會需要一併的去進行子專案依賴，例如一個應用程式服務本身可能就是一個專案，使用者介面自己也是一個。

這種時候就要加入子專案的思維，也就是依賴在一個核心專案下的另一個專案。

一整個我們會稱呼他為專案系統，時常在大型應用程式開發中出現。

幸運的是，Git本身也有提供相關的操作可以讓你更容易地去管理這樣子的架構。

接下來就讓我們進入Submodule的世界。



子模組，或者說子存放庫

Git Submodule的基礎概念

在一個Git專案裡面放另一個Git專案，他的概念並不是直接把另一個存放庫的內容給貼起來，更像是用一個超連結的方式來進行處理。

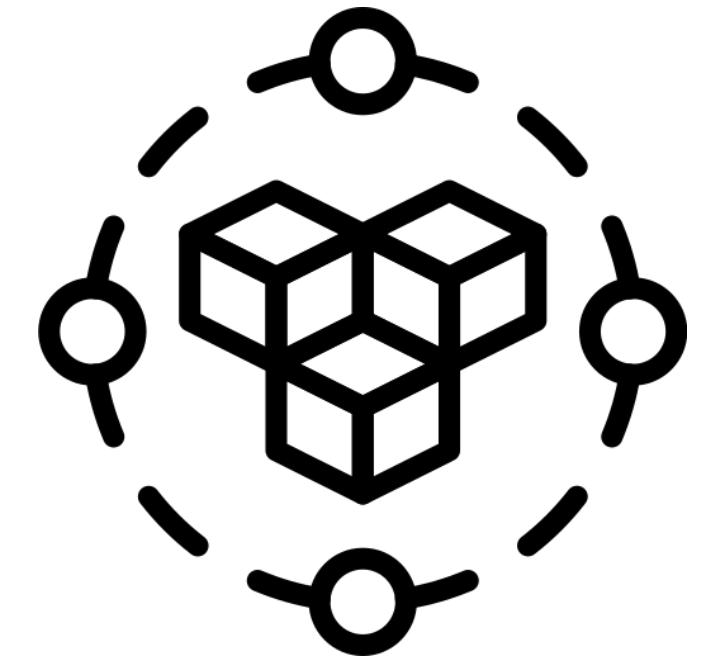
兩者之間還是獨立的存放庫，只是父專案 (Superproject) 會去定時更新子存放庫的認可，並維持兩者之間的聯繫，而且可以直接強制子模組同步成指定的狀態。

一個父專案可以有多個子存放庫，可以方便做管理。



Submodule的正確打開方式

他會直接創建一個 `.gitmodules` 檔案

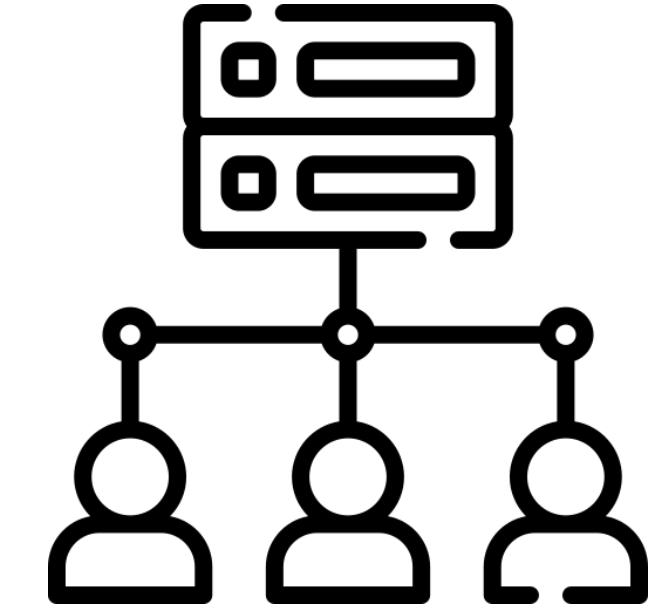


在父專案當中，我們可以使用`git submodule add <存放庫位址> <資料夾名稱>`來進行最基本的子存放庫添加，也就是產生兩者的連結。

當然，只是添加連結並沒有實質效益，我們還要對他進行`init`和`update`來初始化他。

要注意的是，如果從外部clone父專案，也必須要進行一次初始化，這個時候可以用這一行來進行：`git submodule update --init --recursive`，會一併加入子存放庫。

子存放庫本身 他的特性與限制，主要是獨立性的部分



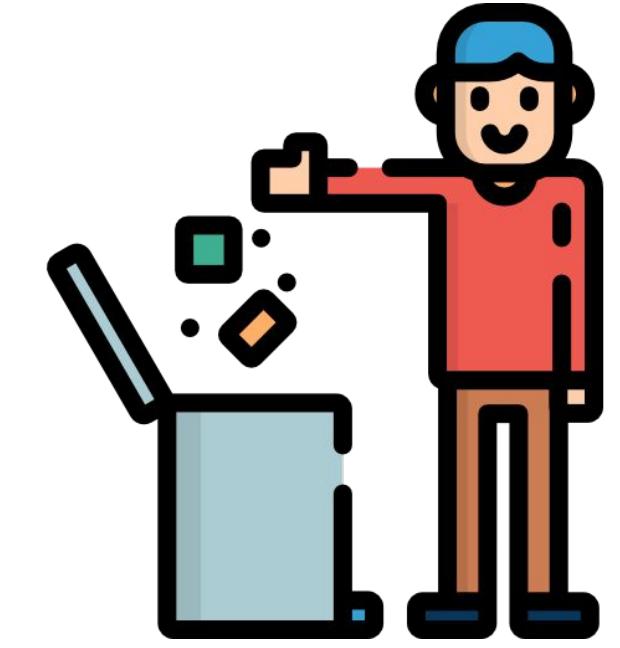
子存放庫與父專案的連結其實並不如大多數人想像的強，你沒辦法直接從父專案中新增認可到子存放庫之內，他做的更多像是個追蹤的角色。

每當子存放庫內做了一些改動，並且認可之後，要記得回到父專案去做git add submodule來追蹤到最新的認可，否則可能會逾期。

子存放庫本身就是一個工作樹，只是跟父專案有上下聯繫而已。

全部丟掉

剛好是個時機來補充 rm的使用



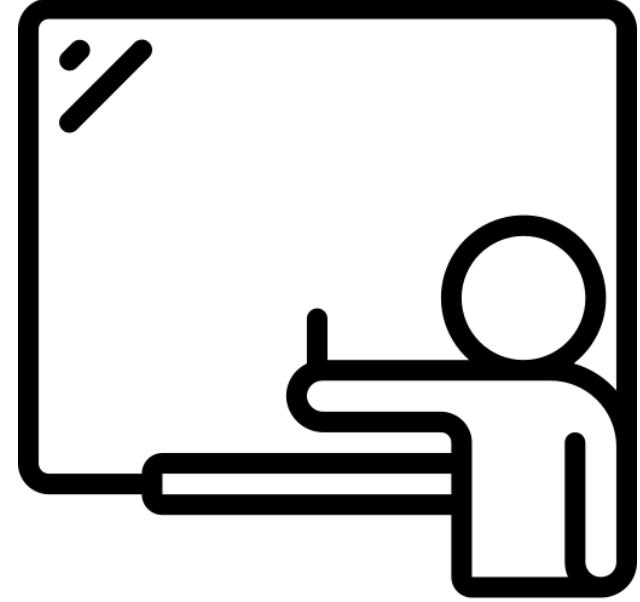
如果要棄置一個子存放庫 (當然除非你真的確定否則建議不要這麼做), 我們的做法就跟移除任何其他檔案一樣, 使用git rm <子存放庫名稱>去做。

git rm這個指令就像Linux系統上的rm一樣, 就是負責刪除檔案的指令, 只是這裡是透過git, 之前有提過git rm -r --cached, 這裡面的--cached其實就是指已經被追蹤的檔案。

順帶一提, -r的意思是所有相關檔案, 而-rf就是強制性的所有相關檔案。

簡單的QA

把容易出現的問題都寫出來了



在做Submodule, 因為他複雜的關係, 有時候會出現一些問題 (大部分剛才都提過了):

- 為什麼子存放庫資料夾沒有東西? → 因為你沒有init加上update。
- 為甚麼父專案沒有追蹤到? → 因為你可能沒有在子專案裡認可, 或者父專案還沒暫存。
- 怎麼加入多個子存放庫? → 單純地用多個git submodule add就好。

記得你雖然移除了子存放庫, 檔案可能還是會留著, 要記得一併刪掉 (除非你還有用到)。



10

本質

就是Git在基層結構上是怎麼運作的。

然後我會放這麼後面就是因為他很無聊。

```
    study = filterByStatus ? study : study.filterByStatus(status)
    status = filterByStatus ? study.status -- filterByStatus : true
    if (status) {
      return filterStudiesIf(studies, filterByOrg)
    } else {
      return studies.filter(study =>
```



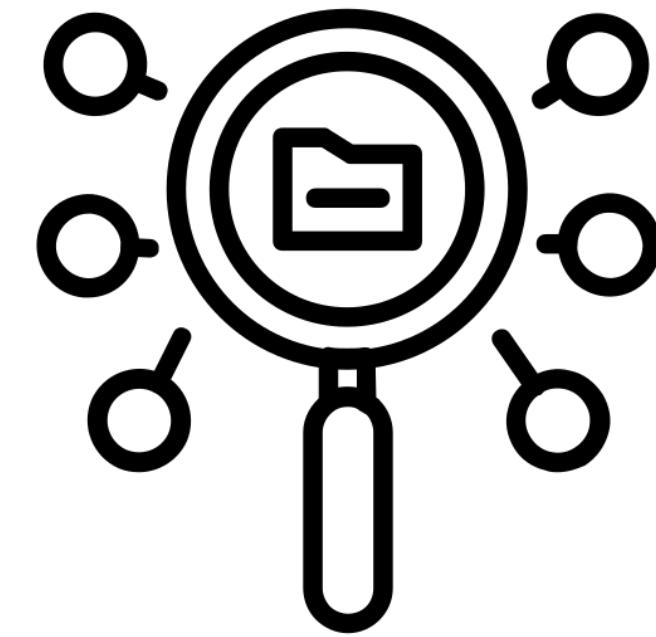
深入探索

其實 Git 是個內容可尋的物件資料庫（沒開玩笑

我們來顛覆各位迄今為止的所有認知，也就是Git比起版控，更像是資料庫系統，更詳細的說就是key-value儲存系統。

基本上你每一個提交，裡面包含的每一個檔案、訊息甚麼的全部在Git的眼中都被視為是物件(blob)，是被SHA-1編碼特別認可的特殊形態。

而類型又被分成：blob (檔案內容)、tree (資料夾)、commit (認可)和tag (標籤)。



SHA 日大作戰

對我已經沒有梗了，簡單來說就是演算法

一直提到的SHA-1，其名為Secure Hash Algorithm 1，在Git的架構當中是類似指紋的概念，是所有物件的唯一識別碼。

他有幾個特點：

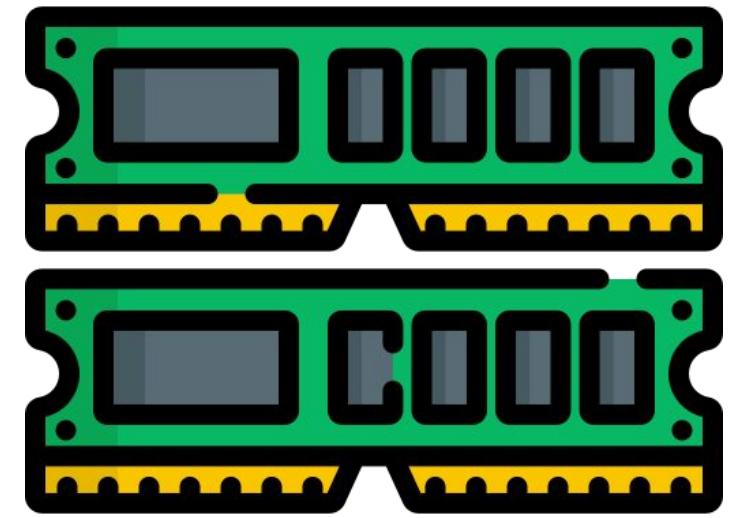
- 是根據內容本身 + 類型資訊算出來的
- 同樣的內容 → 一樣的 SHA
- 改一個字 → 整串 SHA 完全不同
- 無法「反推內容」，但能保證不重複(幾乎)

你可以透過git hash-object <檔案名稱>來看到一個檔案的SHA-1編碼。



記憶的迴廊

到底認可本身是怎麼被追蹤的？



剛剛有說到認可本身就是一個物件，至少對Git來說是這樣，一個認可其實就是一組資料指標加上說明而已，每一次你進行認可，Git都會建立一個Commit物件。

Commit物件裡面會包含：

- 哪個 tree(檔案狀態)
- 上一個 commit(parent)
- 提交者、時間
- commit 訊息

用git cat-file -t/-p <認可的SHA-1>可以看到右圖這樣的展示。

```
tree 20f93...
parent 8937fa2...
author Bernie <bernie@nk.com>
committer Bernie <bernie@nk.com>
```

新增登入按鈕

你可以試試

直接創一個檔案，然後用剛才提到的那些指令試試



先提一下cat-file所使用的兩個參數，-t是查看類型，-p是查看內容。

我們有了這些內部測試指令，你可以自己去實際試試Git的流動。

不過你可能會被嚇到就是了，右圖是我試著列出今天用的簡報的內容時遇到的狀況。

果然說非二進制檔案還是不要亂玩得好。

找到自己的位置

在哪裡， 在哪裡， 不要隱藏你自己



最後，我們來講一下git rev-parse HEAD這個指令，它的用處是找出你現在HEAD所指向的commit的SHA值（有夠繞口），也就是類似導航這樣。

你可以另外加入--abbrev-ref來讓它順便告訴你現在所在的分支名稱，跟我們沒提到的git branch --show-current類似，但又不盡相同（前者會順便提及斷頭）。

總而言之，這些大概就是Git你所應該要知道的相關資訊。

就醬了~



Discord



GitHub

感謝你們的參與!

有任何其他問題可以到我們的DC群詢問，也可以直接寄件給講師們！
今天的內容都放在我們的GitHub頁面上了，有興趣的人可以去看看～

Bernie: ptyc4076@gmail.com