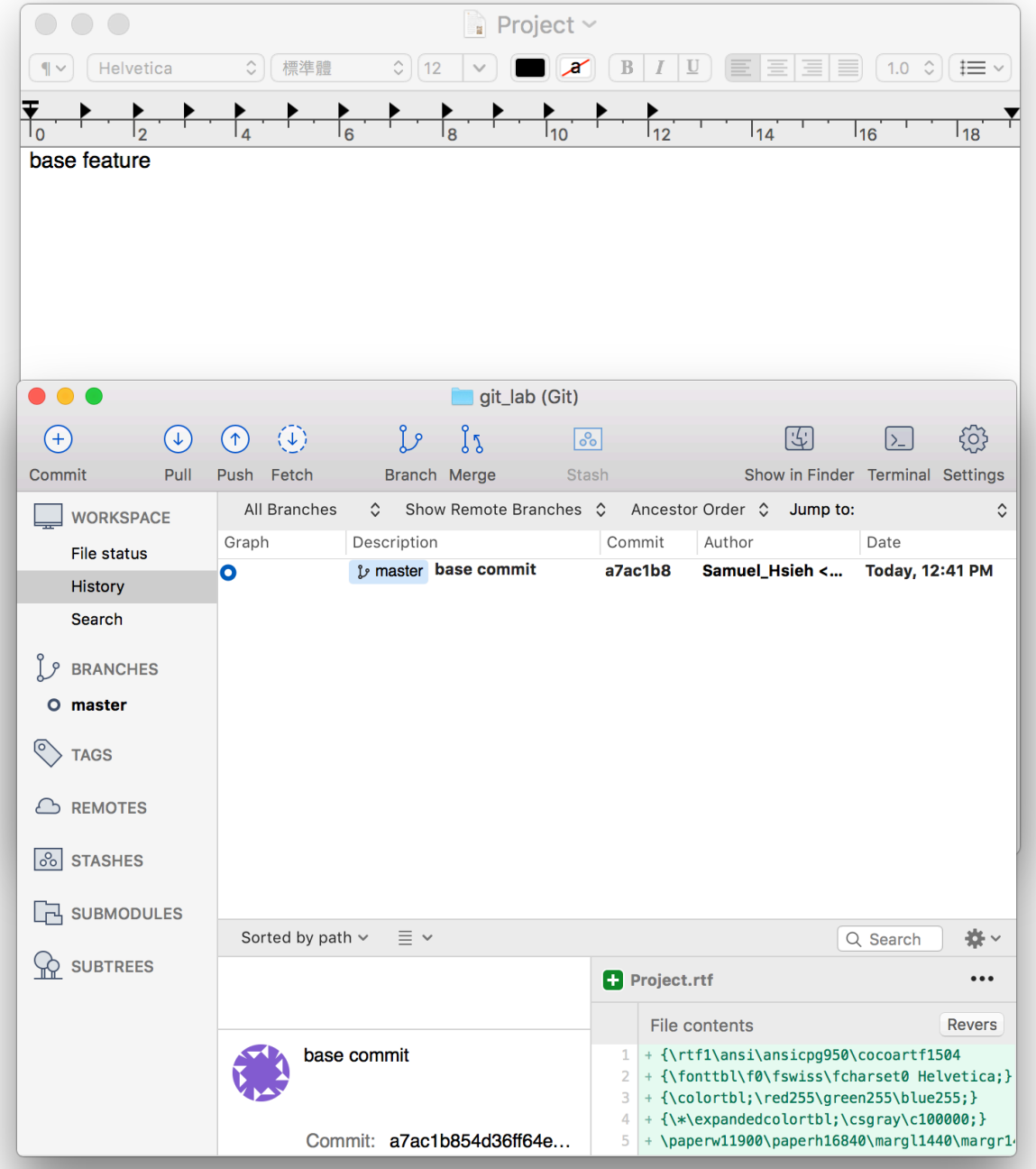
概述

通常我們在多人協同開發專案的時候，我們會有一個branch為master來作為開發的主幹，任何想要開發的新功能會先藉由這個主幹分支出去，然後在分支開發好之後才會merge回master，這樣會是一個比較標準的一個協同開發流程，以下會示範一些協同開發時可能會出現的狀況，利用文字檔案簡單做個演示。

初始樣貌

首先我們會只有一個master主幹，姑且有一些完成的基礎功能我們叫base feature

```
git branch
> * master
```



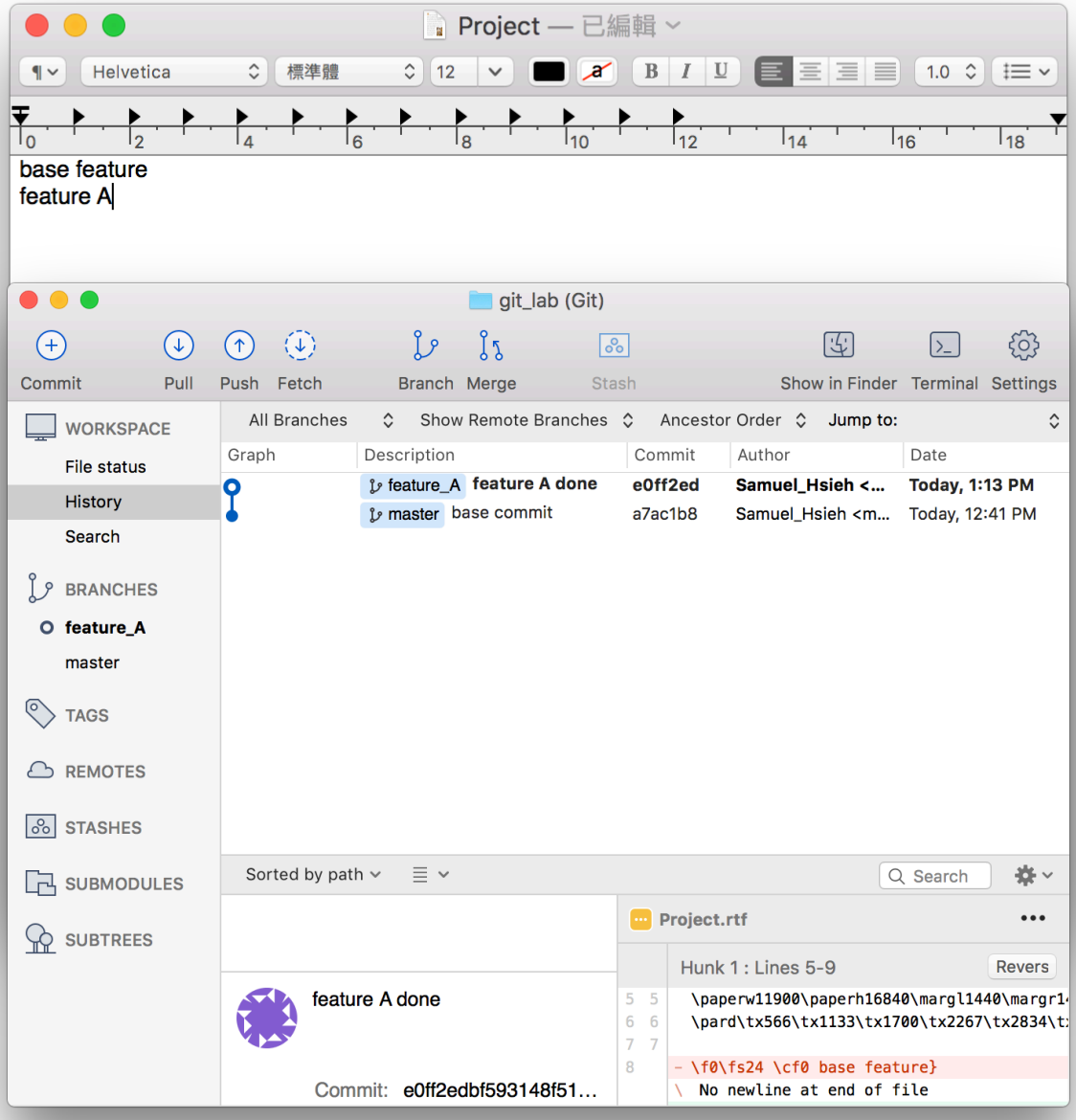
開發功能 A

目前有一個功能 A 的需求需要被開發，為了不直接在master上開發功能，我們把專案先分支出去。

```
git branch feature_A
git checkout feature_A
git branch
> * feature_A
    master
```

開發專案完成

```
git add .
git commit -m “feature A done”
```



合併到主幹之merge和rebase

開發完feature A之後，我們就可以把功能合併回主幹。
確認目前的branch為master之後，我們可以直接下git merge <branch name>來合併，也可以先使用git rebase <branch name>來跟master最新的commit做整合再merge到master，兩種結果比較如下方演示

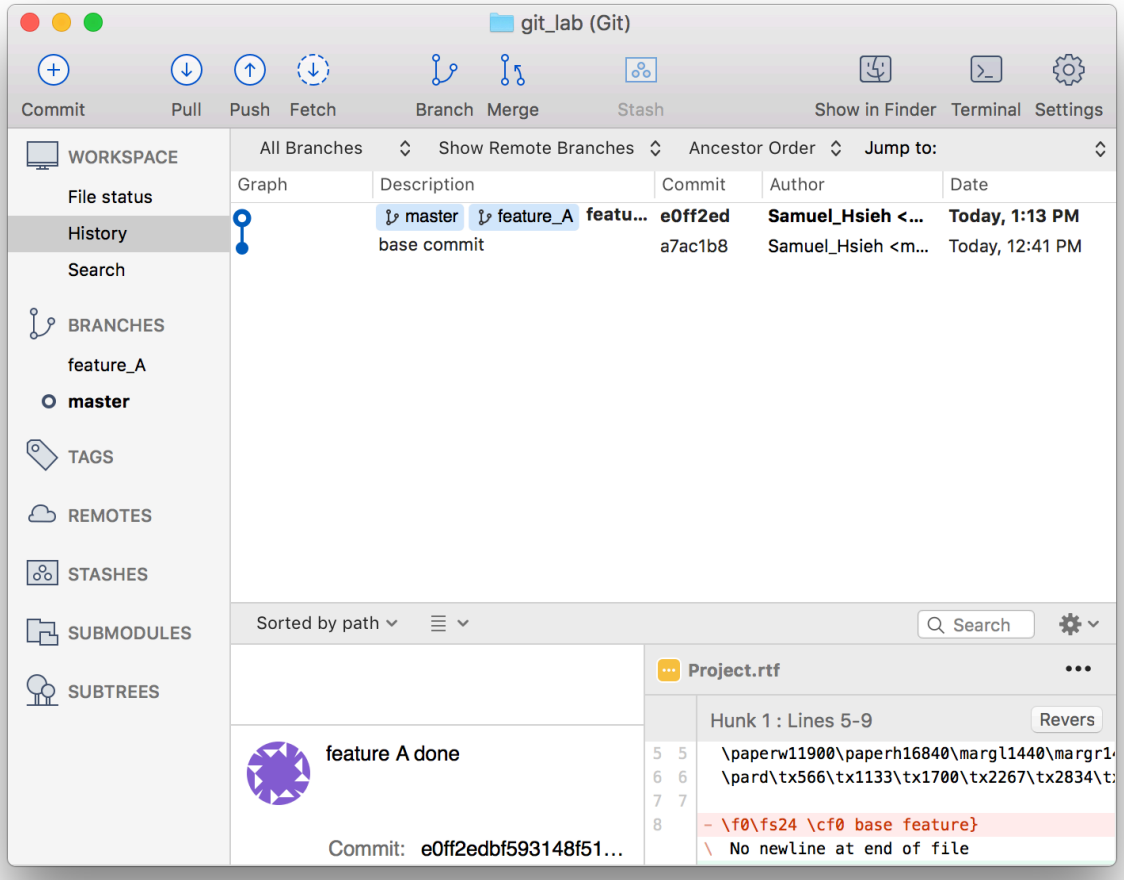
Merge

首先我們需要checkout回master

```
git checkout master
git branch
> feature_A
  * master
```

然後做merge

```
git merge feature_A
```



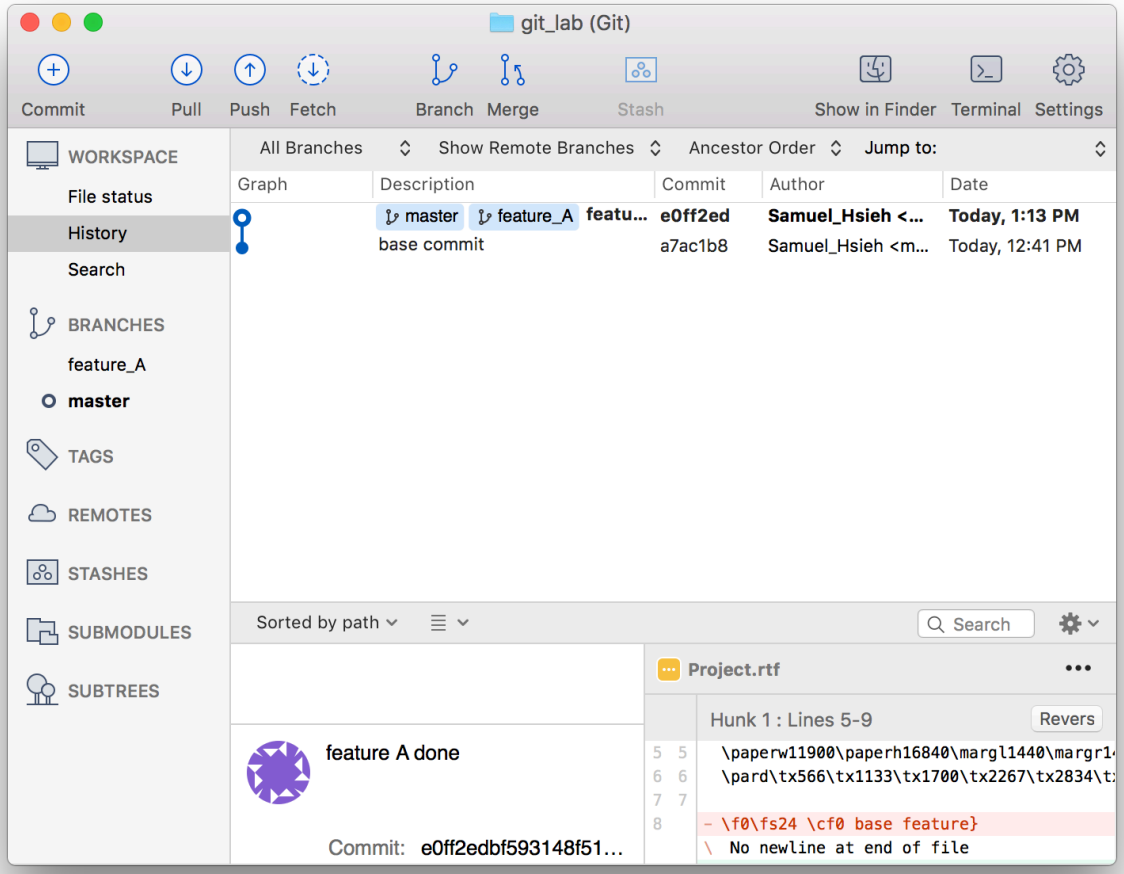
Rebase

先對feature_A個master做整合

```
git rebase master
```

然後在merge到master

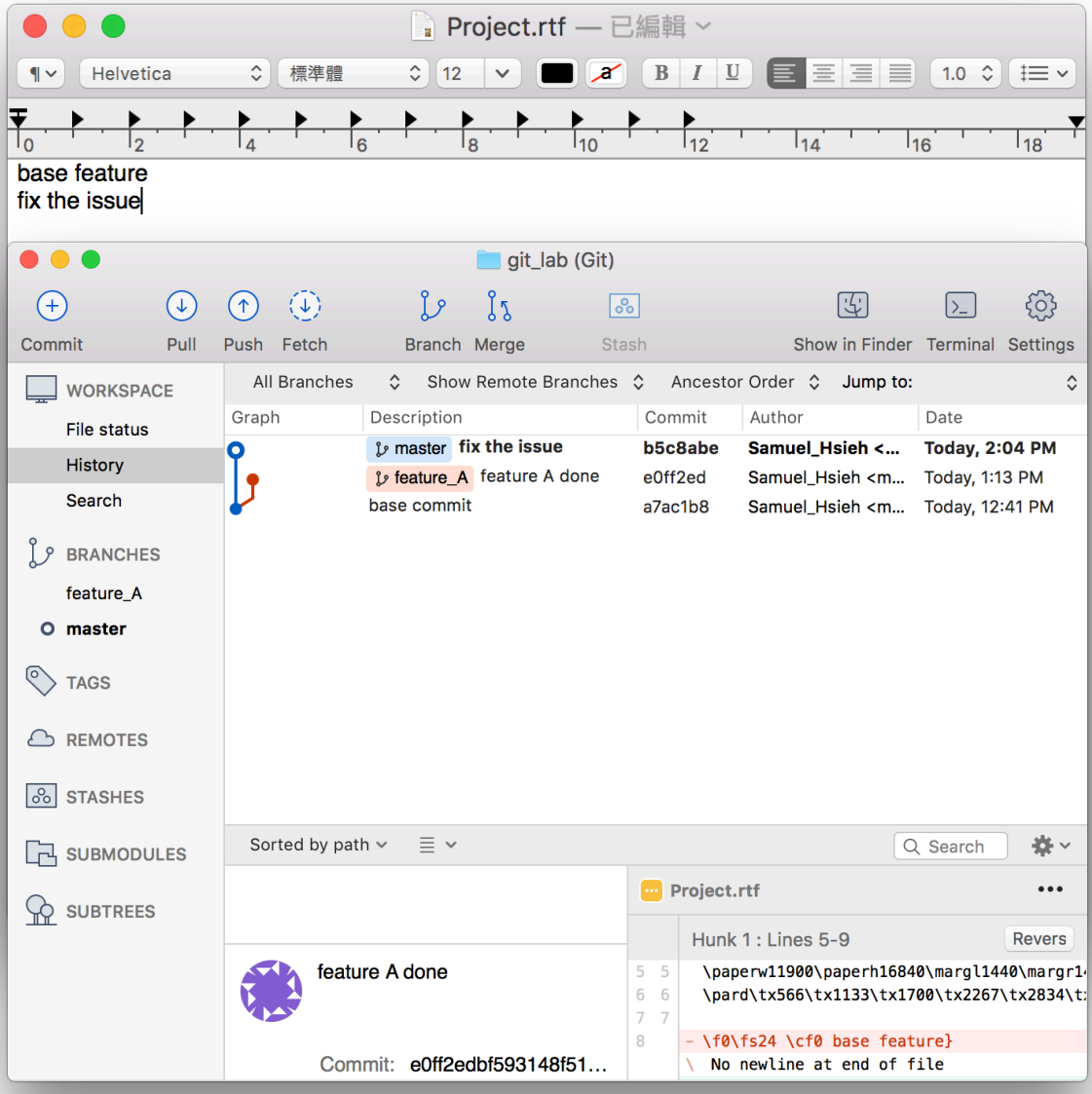
```
git checkout master
git branch
> feature_A
  * master
git merge feature_A
```



這樣Git分支圖不是都長得一樣？

我們剛剛試了merge和rebase，但是....分支圖不都長得一樣？
嗯... 我們目前的master很幸運的是，還沒有人在我們開發feature A的期間去更動它，但是別忘了我們作為協同開發專案，這是一定會有可能發生的情形，所以為了讓情況更真實一點，我做了一些改變。

我們checkout回master，然後假設發現master的base feature裡有一些issue存在，另一位開發者順手修復了他，並且提交commit上去，此時會發現git分支圖長這樣，像是一個叉子了。

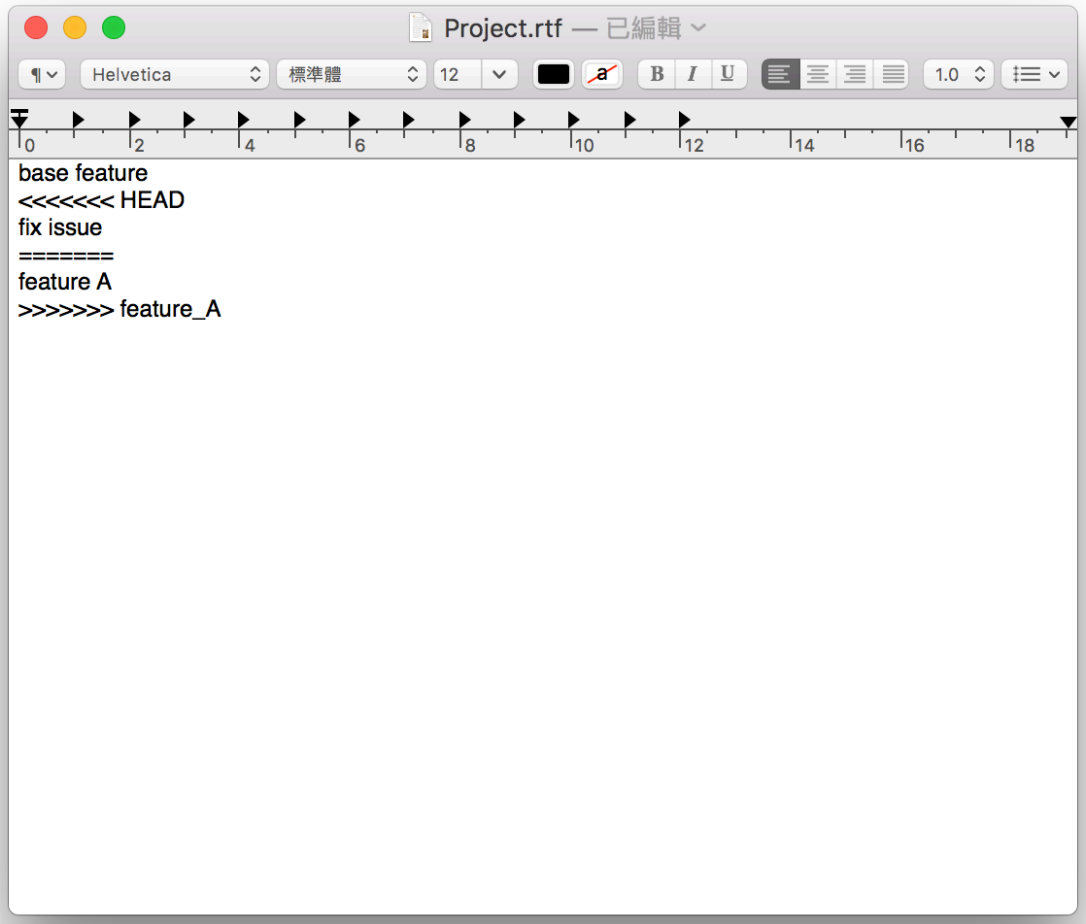


然而我現在作為feature A的開發者，我想要把feature A合併回master時，發現原先我是基於base commit這個支點去延伸出feature A這個功能的，我應該要合併回master的話，我可以先rebase master再merge回master也可以直接做merge，現在就來繼續演示兩種作法的不同。

Merge

```
git checkout master
git branch
> feature_A
  * master
git merge feature_A
> Auto-merging Project.rtf
  CONFLICT (content): Merge conflict in Project.rtf
  Automatic merge failed; fix conflicts and then commit the result.
```

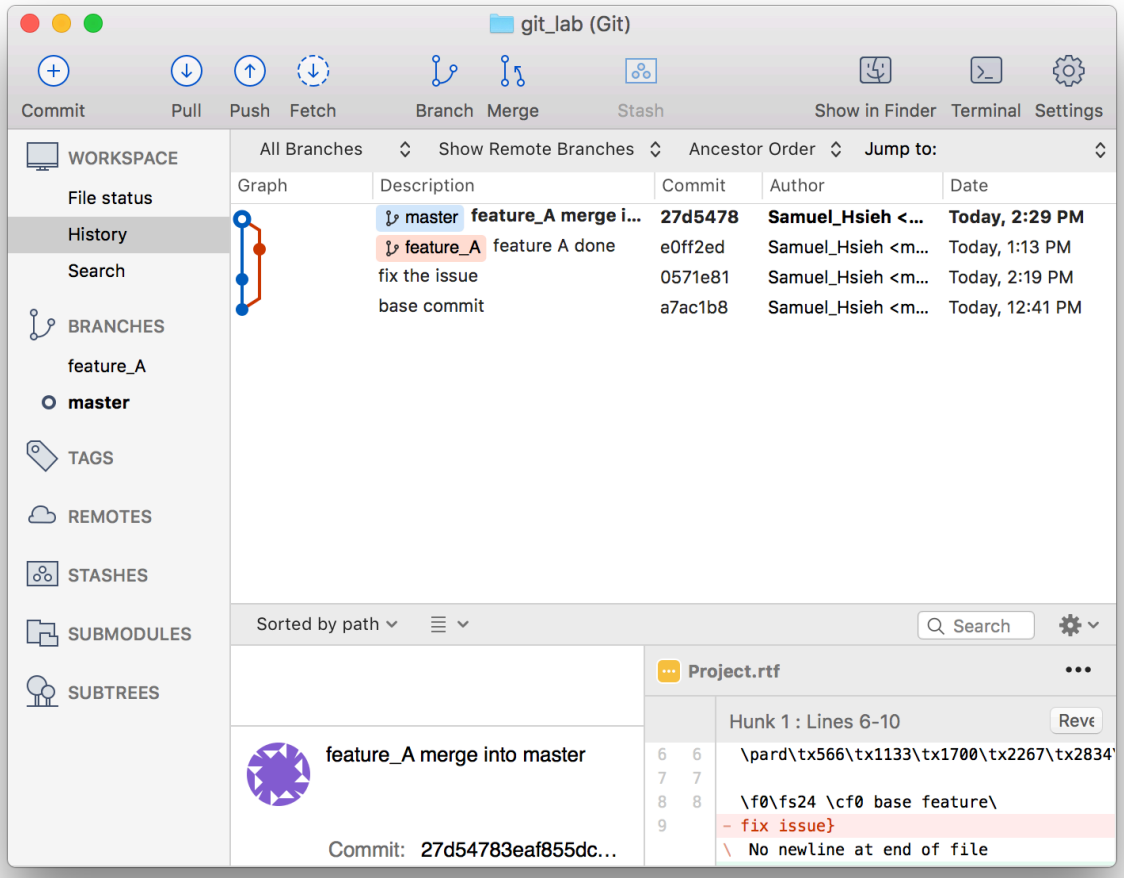
此時出現衝突，必須手動處理（必須也把<<<<<<HEAD ======= >>>>>>>feature_A 等等這些 error message 刪除）



改好之後就可以做commit

```
git add .
git commit -m ”feature_A merge into master”
```

之後分支圖會長這樣，可以看到分支圖的完整歷史流程，而且會在合併的節點自動產生一個commit節點。



小Tip：

1.如果想取消合併可以利用reset指令，的方式回到最初叉子的樣子

```
git reset —hard HEAD~
```

2.如果想取消衝突

```
git reset —hard
```

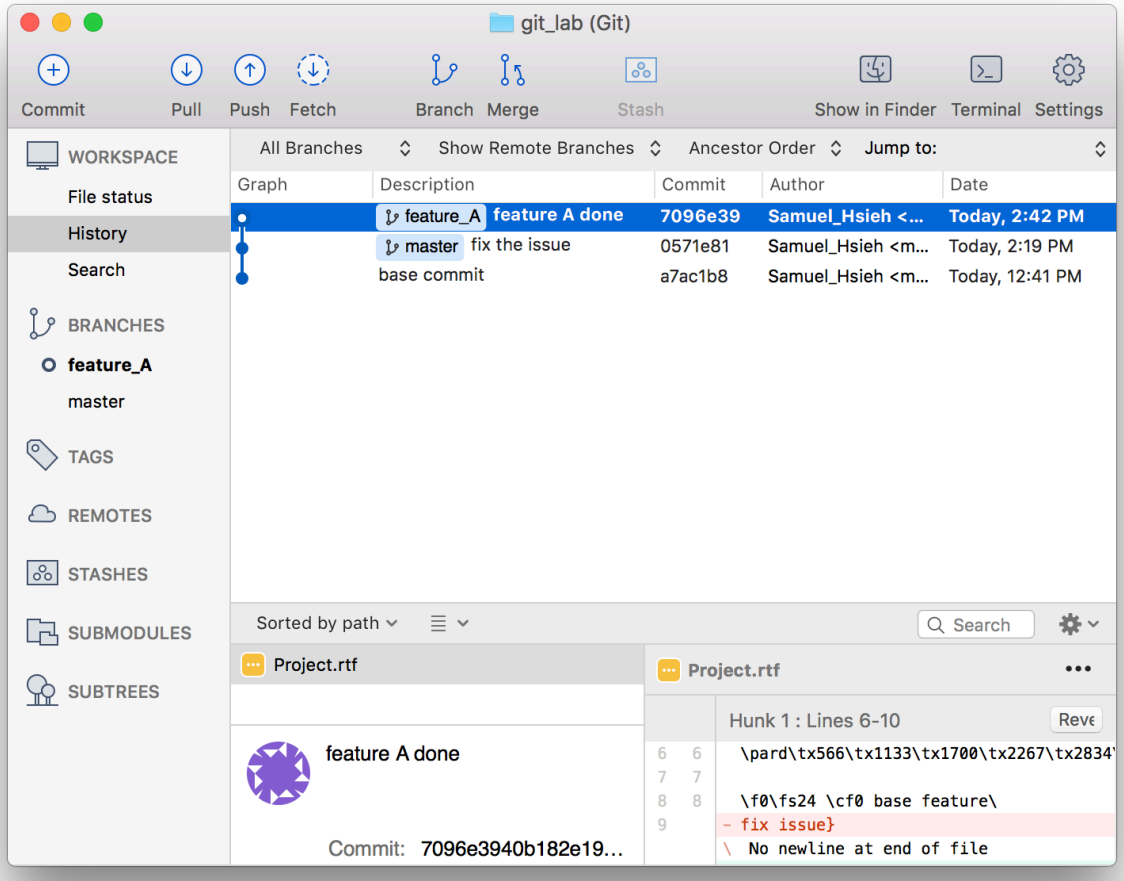
Rebase

```
git checkout feature_A
git branch
> * feature_A
  master
git rebase master
git checkout master
git branch
> feature_A
  * master
git merge feature_A
```

(可以不必像我一樣checkout之後都檢查當前branch，此舉只是為了讓各位清楚而已)
rebase的時候也會出現衝突，改好之後利用—continue來繼續rebase

```
git add .
git rebase —continue
> Applying: feature A done
```

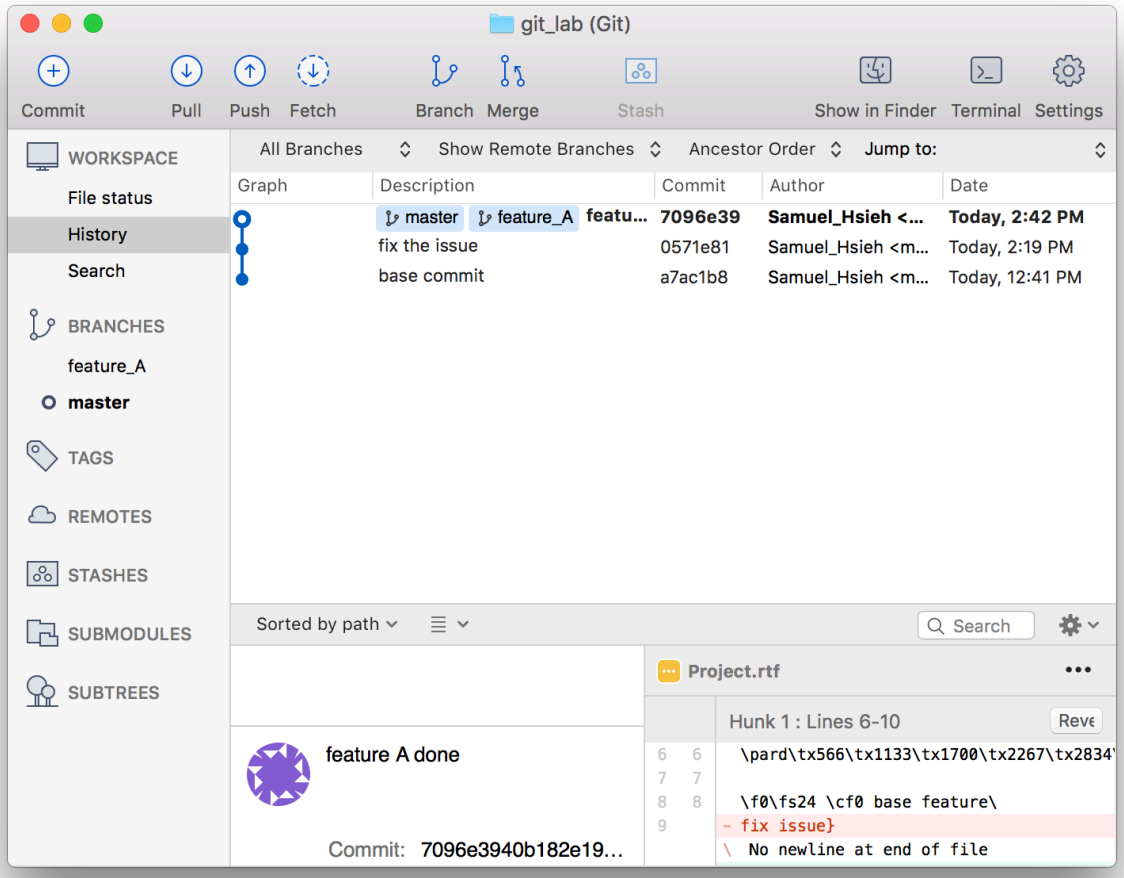
之後分支圖會長這樣，feature_A會重新基於master最新的commit內容



最後在merge到master，master會很自動的做Fast-Forward，將HEAD提到新的commit上

```
git checkout master
git branch
> feature_A
  * master
git merge feature_A
```

完成整個rebase合併，rebase的分支圖看起來乾淨許多，但是看不出來其實feature_A原先是基於base commit做開發的，之後master才fix the issue，所以歷史資料上就比較難回去調查了

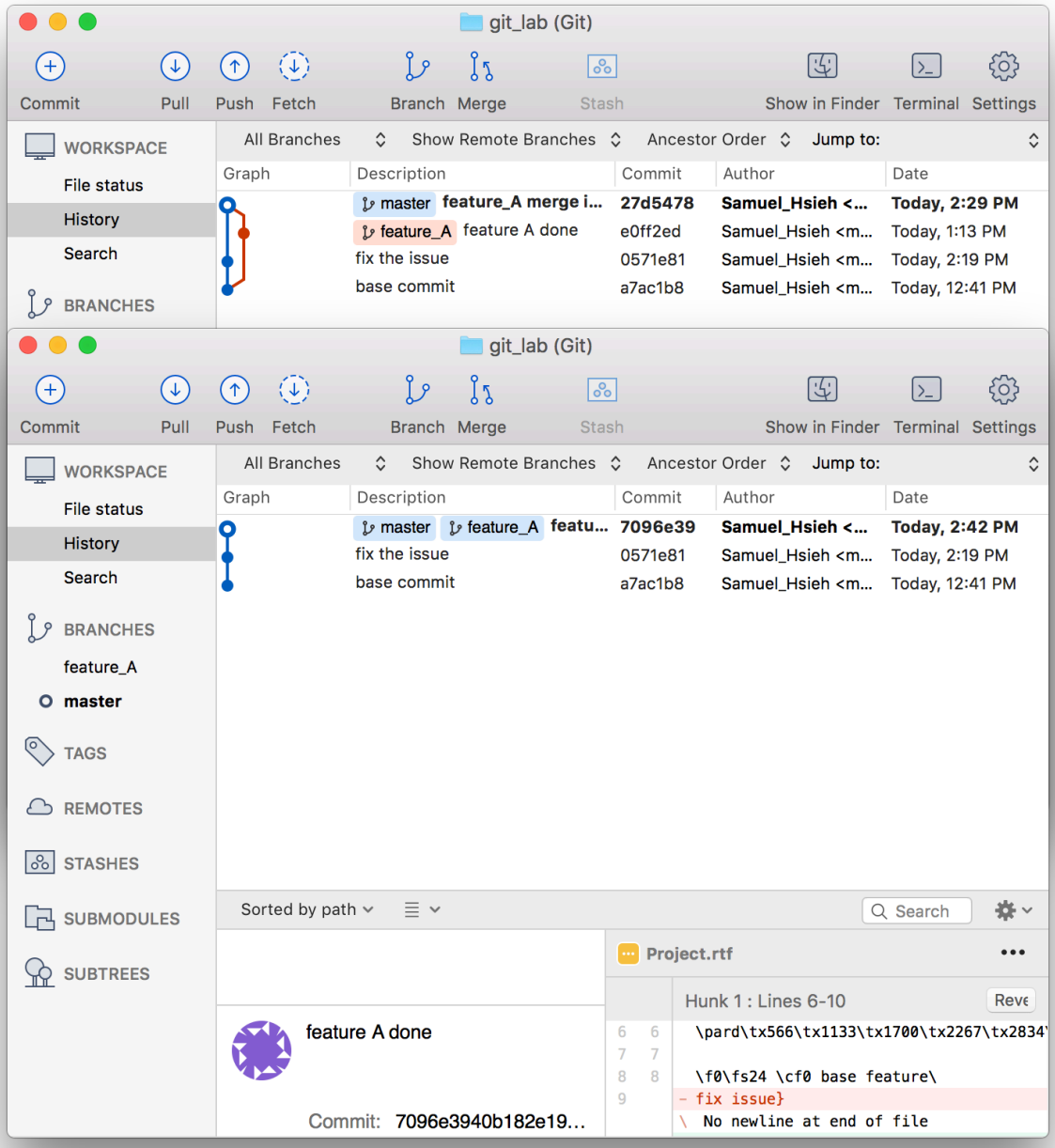


小Tip：

1.如果想取消衝突

```
git rebase —abort
```

放在一起比較一次



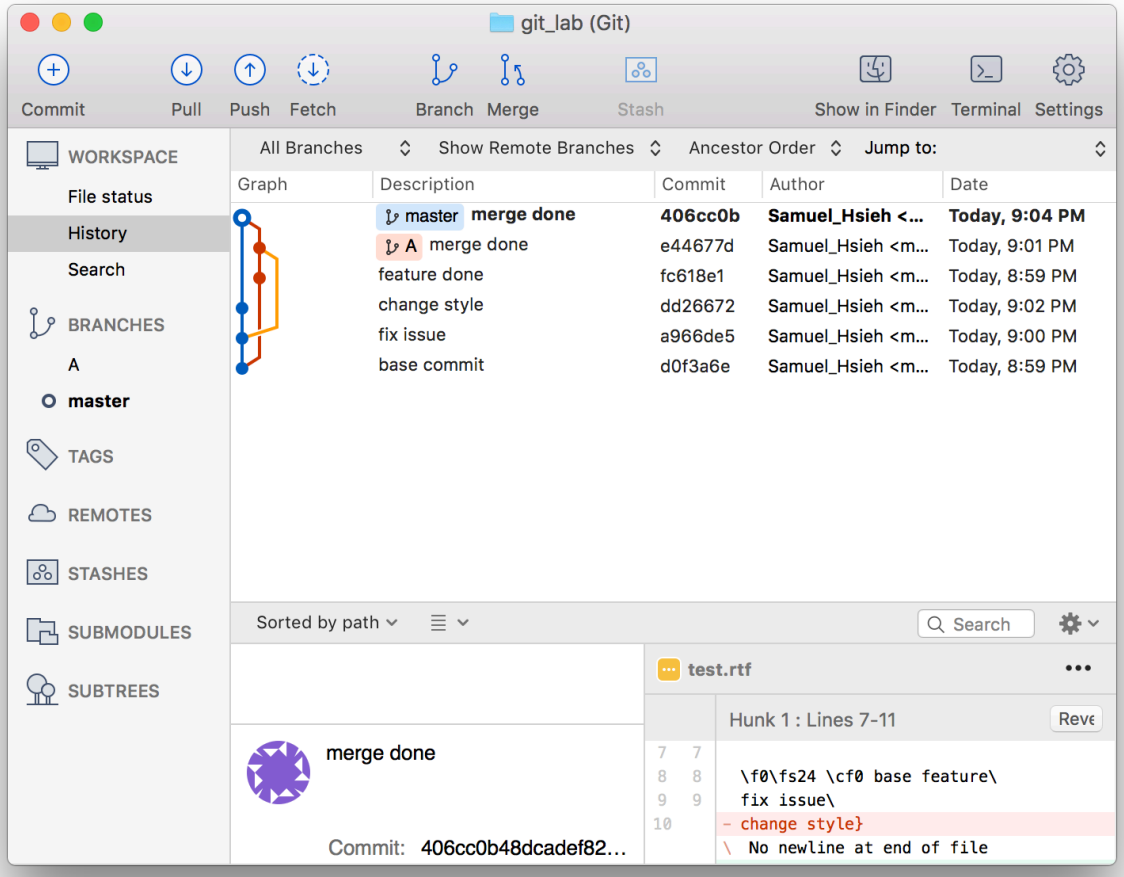
最後我們發現，有rebase的合併可以用來整合master的最新commit到feature_A，而且直接merge的結果是feature_A保有基於base commit的commit，順便比較出rebase和merge兩種合併的差異，rebase的合併就是用來重新定義參考基準，從最新的commit點去蓋上去，而merge是用來真正做合併兩個分支，通常多人協同開發時，rebase用來將master最新的commit合併到自己的分支，個人改完之後才用merge合併回master。

更複雜的狀況

我這邊做了兩個專案，一個專案只用merge去做，另一個有使用rebase合併

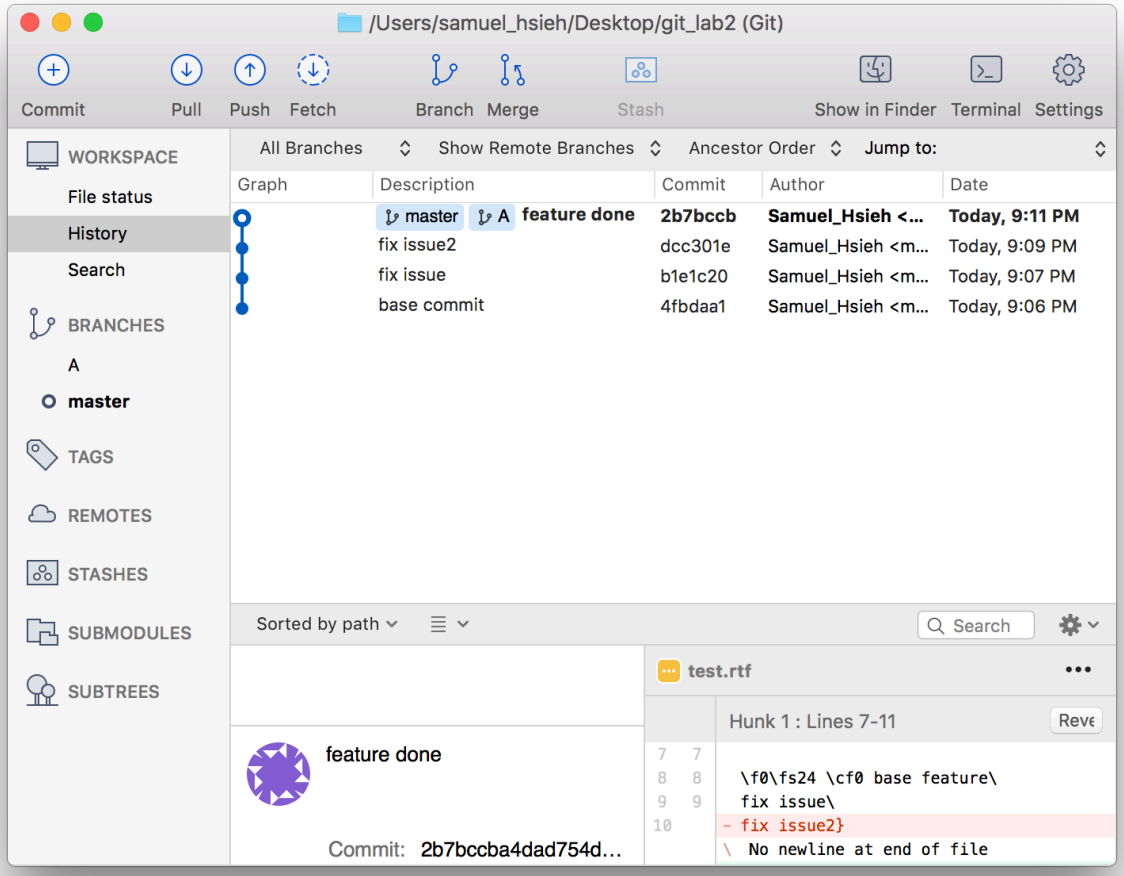
第一個專案過程是這樣：

1.base commit (master)
2.branch A
3.開發新功能並提交(A)
4.fix issue並提交(master)
5.merge master(A)
6.開發新功能並提交(A)
7.fix issue並提交(master)
8.merge A(master)



看起來超級複雜！！！

第二個專案過程如下：
1.base commit (master)
2.branch A
3.開發新功能並提交(A)
4.fix issue並提交(master)
5.rebase merge(A)
6.開發新功能並提交(A)
7.fix issue並提交(master)
8. rebase merge(A)
9.merge A(master)



是不是簡潔多了！！ 由此可知rebase的重要性！