

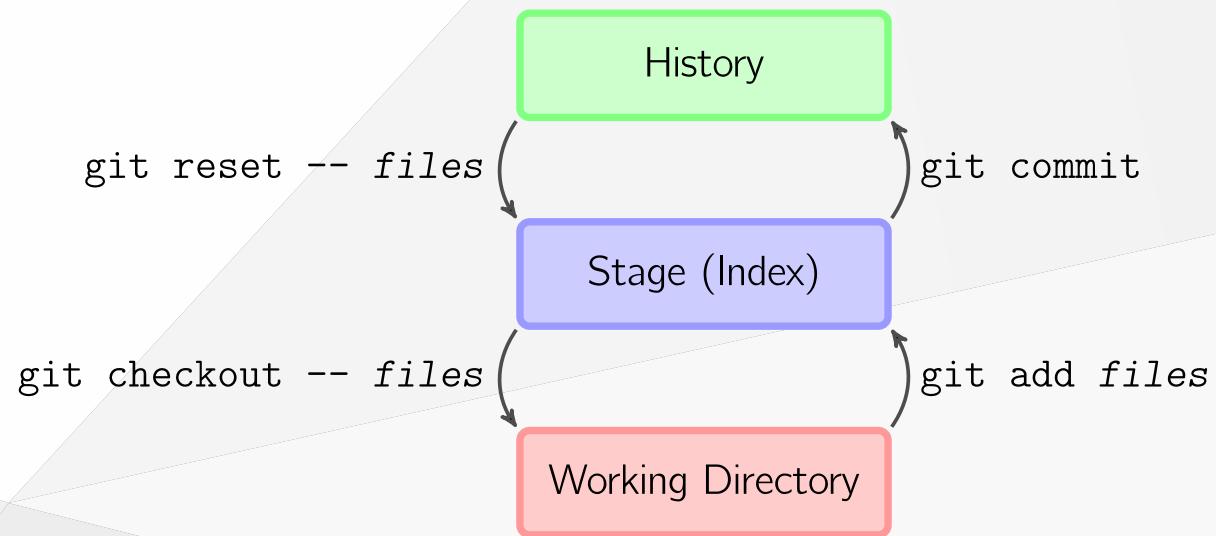


GIT, GET, GOT!

程式碼管控與多人協作
[江豪文](#) + Bing Chat

Git 的三態

- 英文的三態:
 - sing, sang, sung
- Git 的三態:
 - git 工作目錄
 - get 暫存目錄 (索引)
 - got 儲存庫 (歷史)

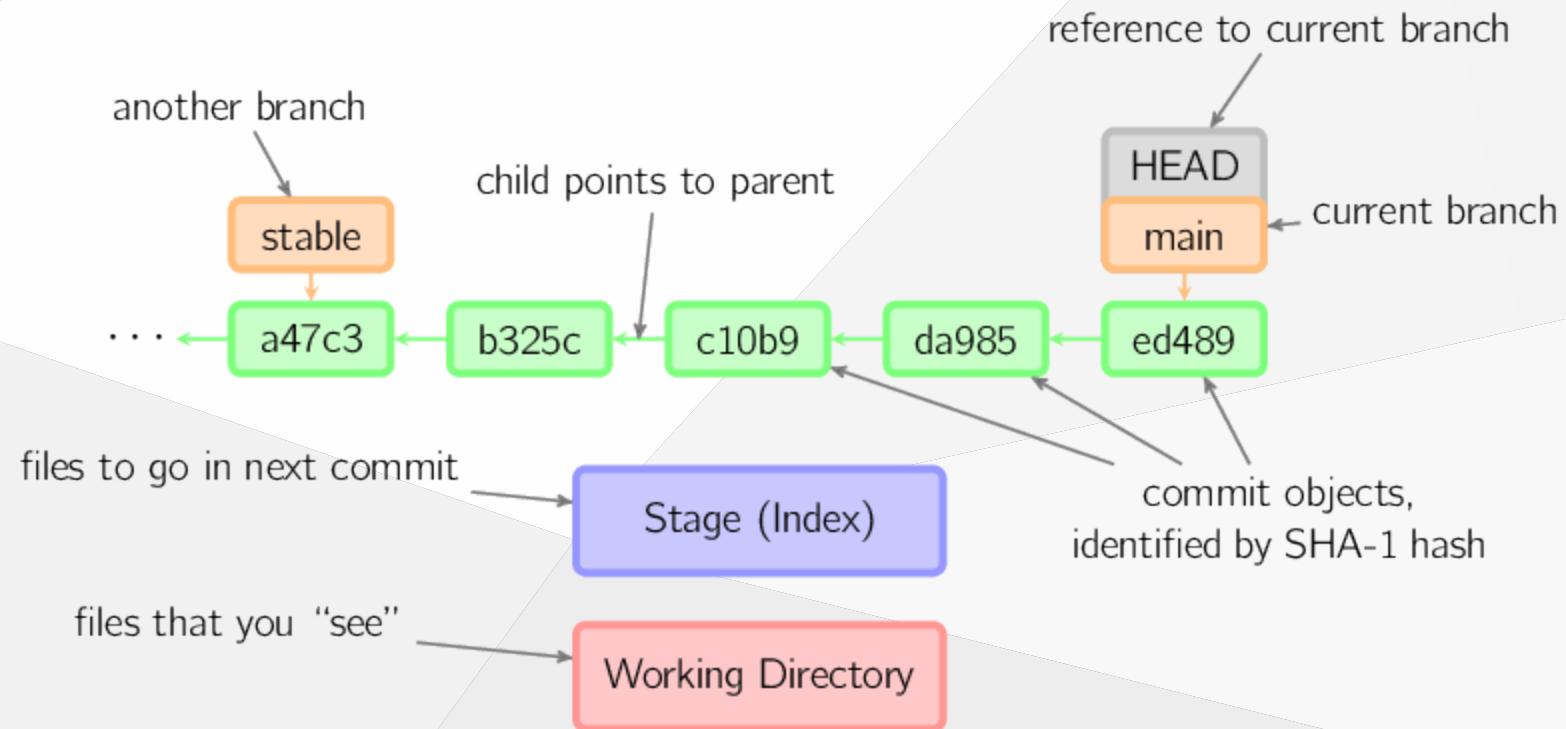


錯誤的版本控制

| Name |
|---|
| ►  文檔企劃案_2 |
| ►  文檔企劃案_20160503 |
| ►  文檔企劃案_20160607 |
| ►  文檔企劃案_20160607_最新版 |
| ►  文檔企劃案_20160721_James_edit |
| ▼  文檔企劃案_最新版 |
|  行程表_20160521 |
|  行程表_20160707 |
|  行程表_20160724 |
|  行程表_開會專用 |

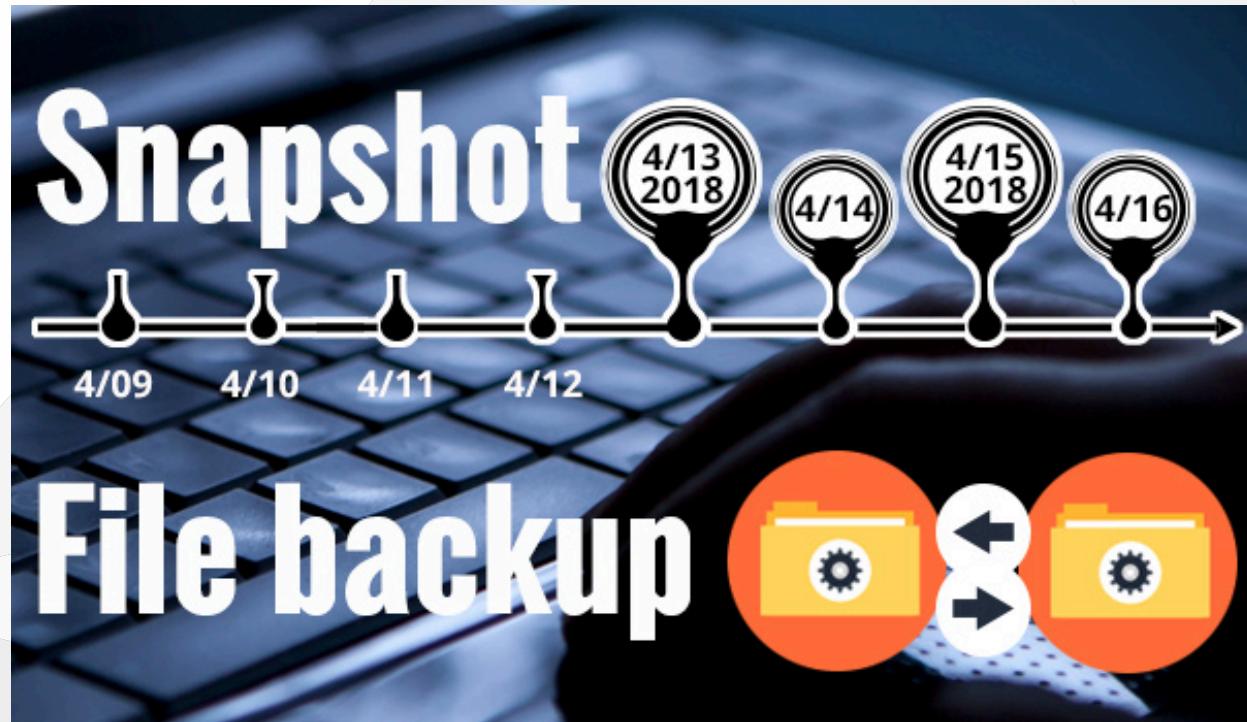
<http://yourgene.pixnet.net/album>

正確的版本控制: Git



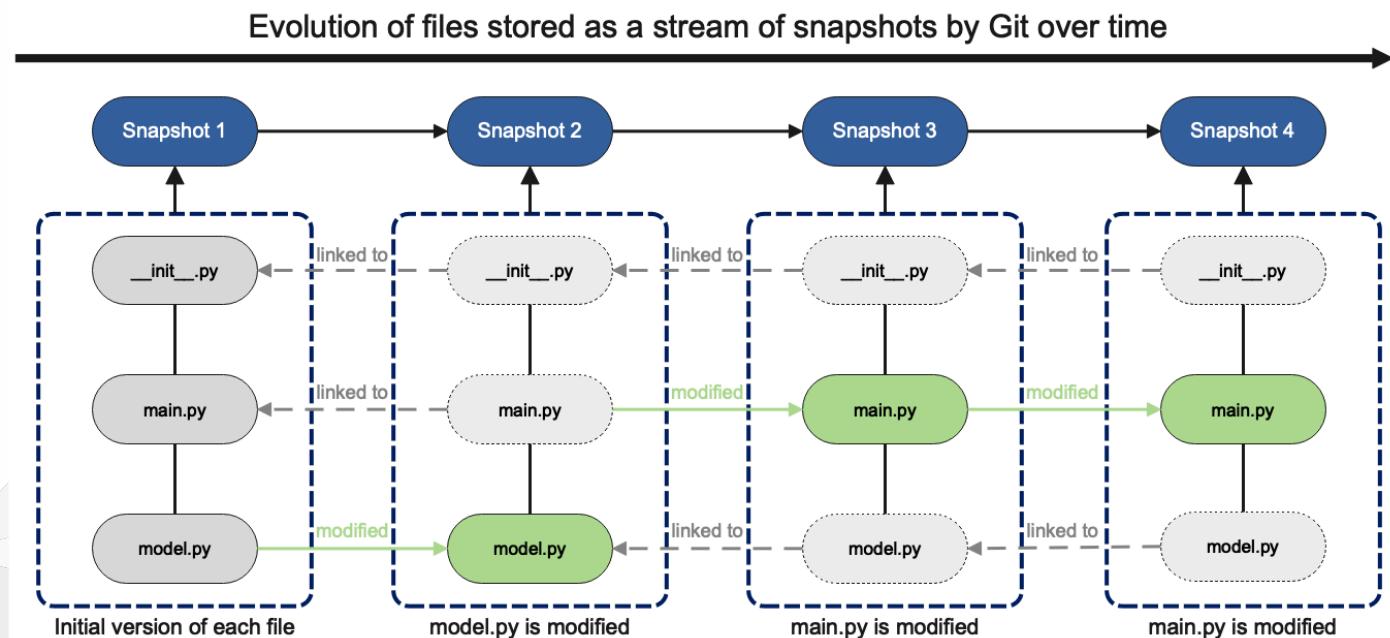
快照系統

- Git 的資料結構本質上是一個快照系統。
- 每次提交 (commit) 時，Git 都會記錄下所有檔案的當前狀態，如果檔案沒有變更，Git 就會重用上一次的快照。
- 這種方式使得 Git 非常高效，因為它只記錄變更，而不是每個檔案的完整複本。



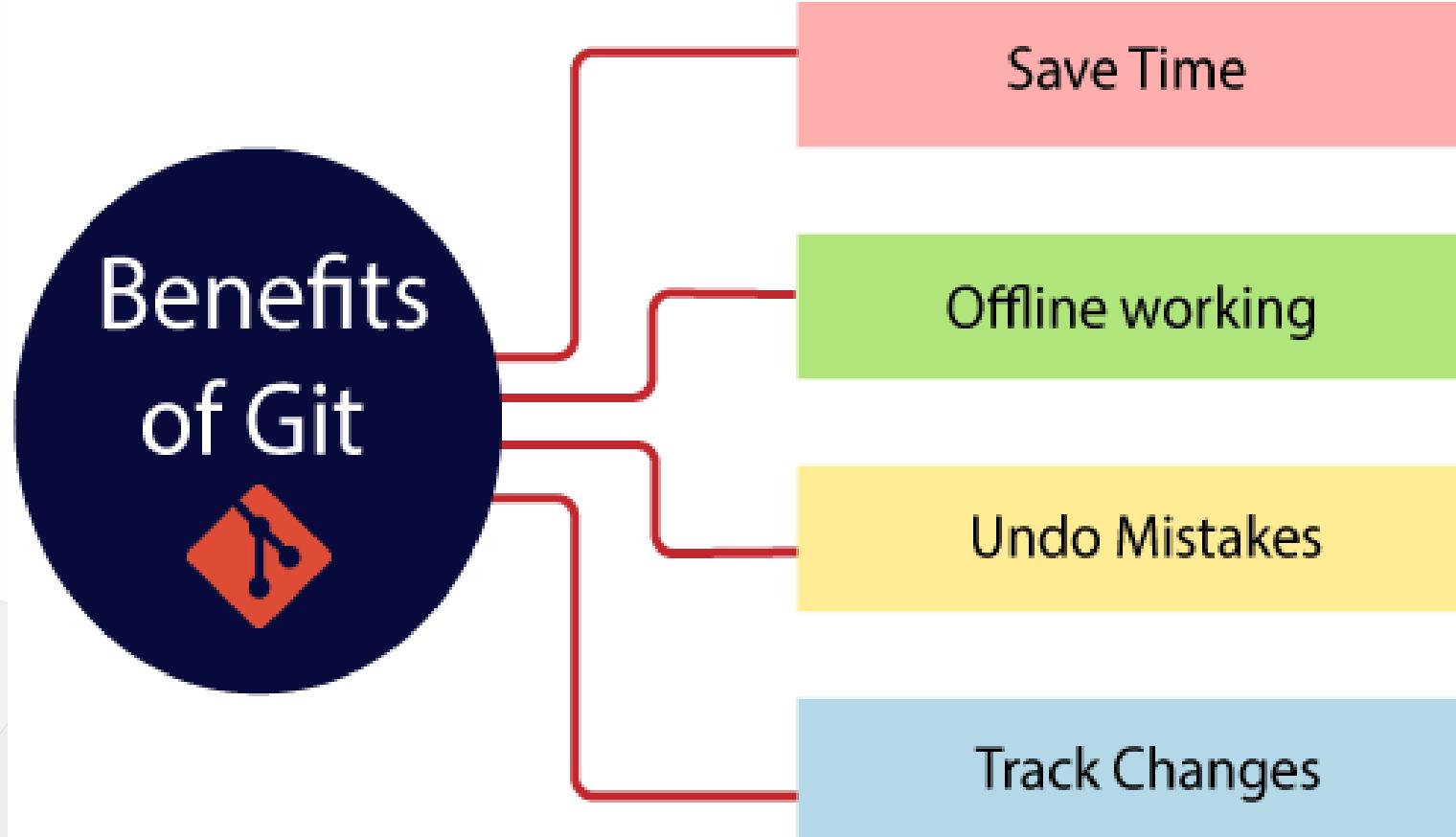
相簿的比喻

- **每頁相簿**：每次提交，就像是相簿中的一頁。
- **重用照片**：如果事件之間沒有變化，就像是重用上一次的照片。
- **記錄變化**：相簿只記錄每次事件的變化，而不是每次都有全新的照片集。



大綱

- 01 > 基本操作
- 02 > 本地修改
- 03 > 新增分支
- 04 > 遠端更新
- 05 > 處理檔案



安裝 Git Client

Windows 環境

- Git for Windows
- MobaXterm

The screenshot shows a terminal window titled '/c' running on MobaXterm Personal Edition v12.3. The terminal displays the following text:

```
? MobaXterm Personal Edition v12.3 ?
(X server, SSH client and network tools)

Computer drives are accessible through the /drives path
DISPLAY is set to 192.168.1.102:0
Using SSH, your remote DISPLAY is automatically forwarded
Command status is specified by a special symbol (✓ or ✘)

About:
MobaXterm Personal Edition. The Professional edition
allows you to customize MobaXterm for your company: you can add
your logo, your parameters, your welcome message and generate
an MSI installation package or a portable executable.
Also modify MobaXterm or develop the plugins you need.
Information: https://mobaxterm.mobatek.net/download.html
```

Terminal history:

- ⌚ 01:55.55 ➔ /home/mobaxterm ➔ ls
LauncherFolder MyDocuments
- ⌚ 01:55.59 ➔ /home/mobaxterm ➔ cd C:
- ⌚ 01:56.15 ➔ /drives/c ➔ ls
 - baafa696b059812b4
 - ettings
 - KRECYCLE
 - LeakHotfix
 - PerfLogs
 - Program Files
 - Program Files (x86)
 - ProgramData
 - Recovery
 - System Volume Information
 - Users
 - VC67
 - Windows
 - bootmgr
 - drivers
 - hiberfil.sys
 - kingsoft
 - pagefile.sys
 - swapfile.sys
 - usr
- ⌚ 01:56.16 ➔ /drives/c ➔

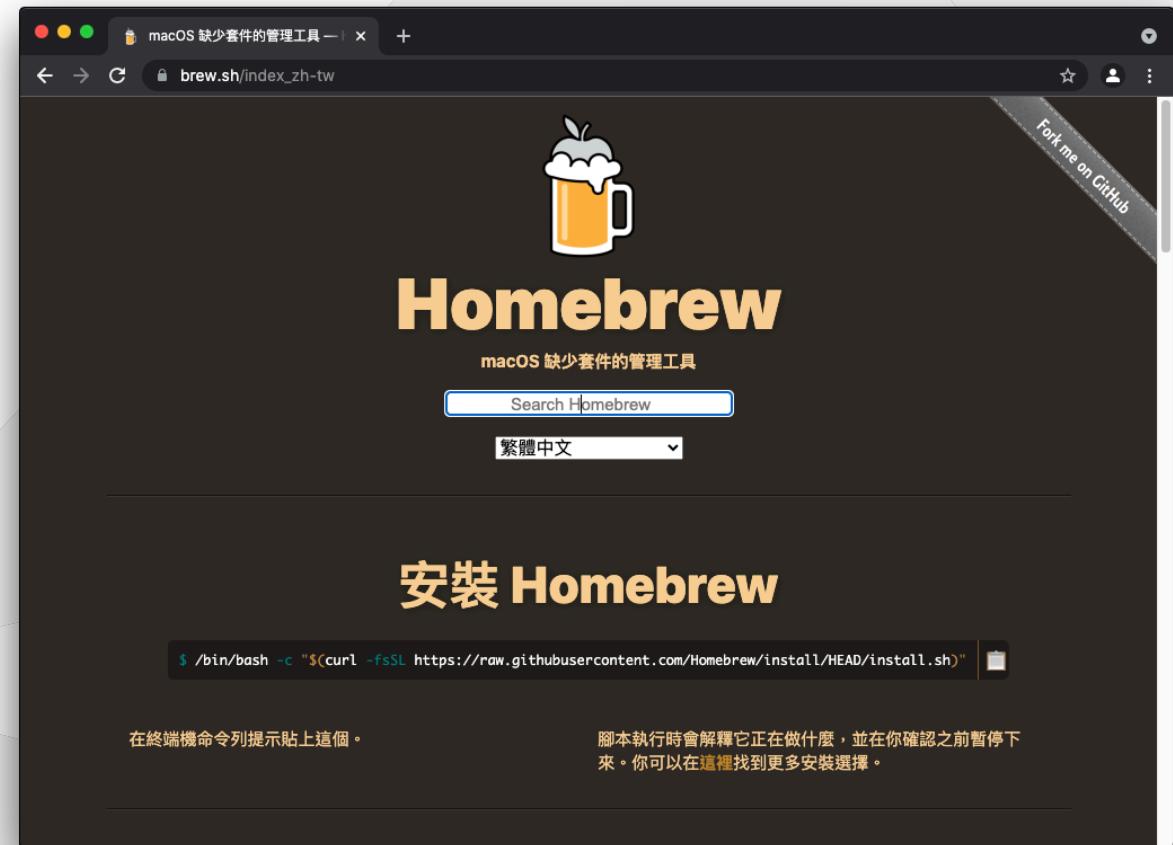
安裝 Git Client

macOS 環境

1. 如果已經安裝了 Homebrew，
可以直接在終端機輸入

```
brew install git
```

2. 如果沒有安裝 Homebrew，可
以從[Git 官方網站](#)下載並安裝



安裝 Git Client

Linux 環境

在終端機輸入以下指令：

1. Debian/Ubuntu 系統：

```
sudo apt-get install git-all
```

2. Fedora 系統：

```
sudo yum install git-all
```

Debian vs Fedora



Debian



Fedora



Git Config

設定識別資料

在你安裝 Git 後首先應該做的事是設定使用者名稱及電子郵件。這一點非常重要，因為每次 Git 的提交會使用這些資訊，而且提交後不能再被修改。

```
git config --global user.name "Your Name"  
git config --global user.email your_email@example.com
```

Git Config

指定編輯器

- 你可以設定預設的文書編輯器，當 Git 需要你輸入訊息時會使用它。
- 啟動 VS Code 的命令為 `code`。

```
git config --global core.editor code
```

Git Config

檢視所有設定值

- 若你想檢查設定值，可使用 `git config --list` 命令列出所有 Git 在目前位置能找到的設定值。
- `--global` 表示所有 Git 儲存庫，`--local` 表示當前 Git 儲存庫。

```
git config --global --list
```

Git Config

檢視特定設定值

你也可以輸入 `git config <key>` 來檢視某個設定目前的值。

```
git config --global user.name
```

Git Init

要在當前目錄創建一個儲存庫，可以使用 `git init` 新增。



- 若當前目錄有 `marp-sample.md` 與 `git-get-got.md` 兩個檔案，執行 `git init` 後的狀態為：

```
$ git status
On branch master

No commits yet

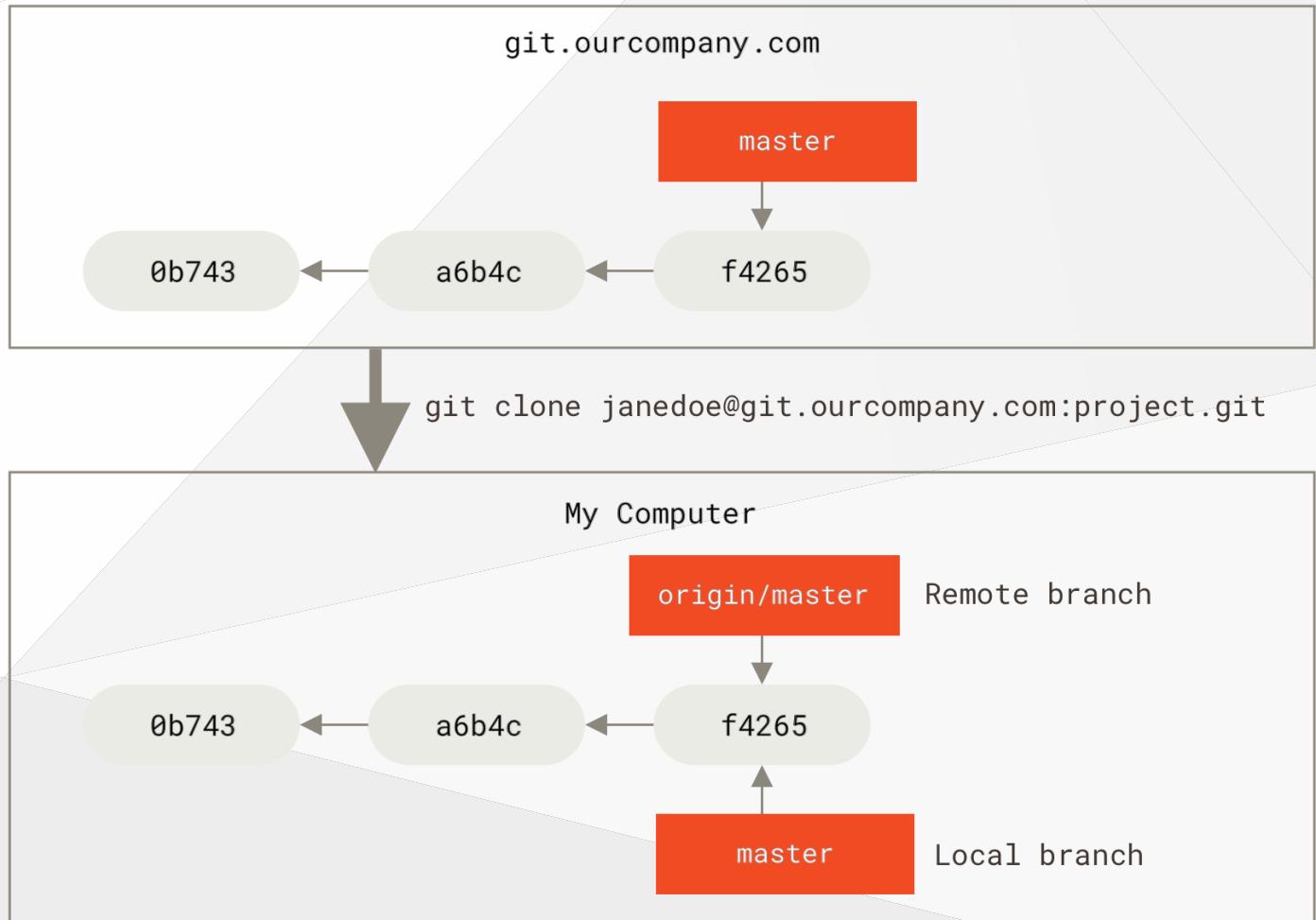
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git-get-got.md
    marp-sample.md

nothing added to commit but untracked files present (use "git add" to track)
```

Git Clone

如果你想獲取一個現有的 Git 儲存庫的副本，可以使用 `git clone` 命令來克隆儲存庫。

```
git clone <repository-url>
```



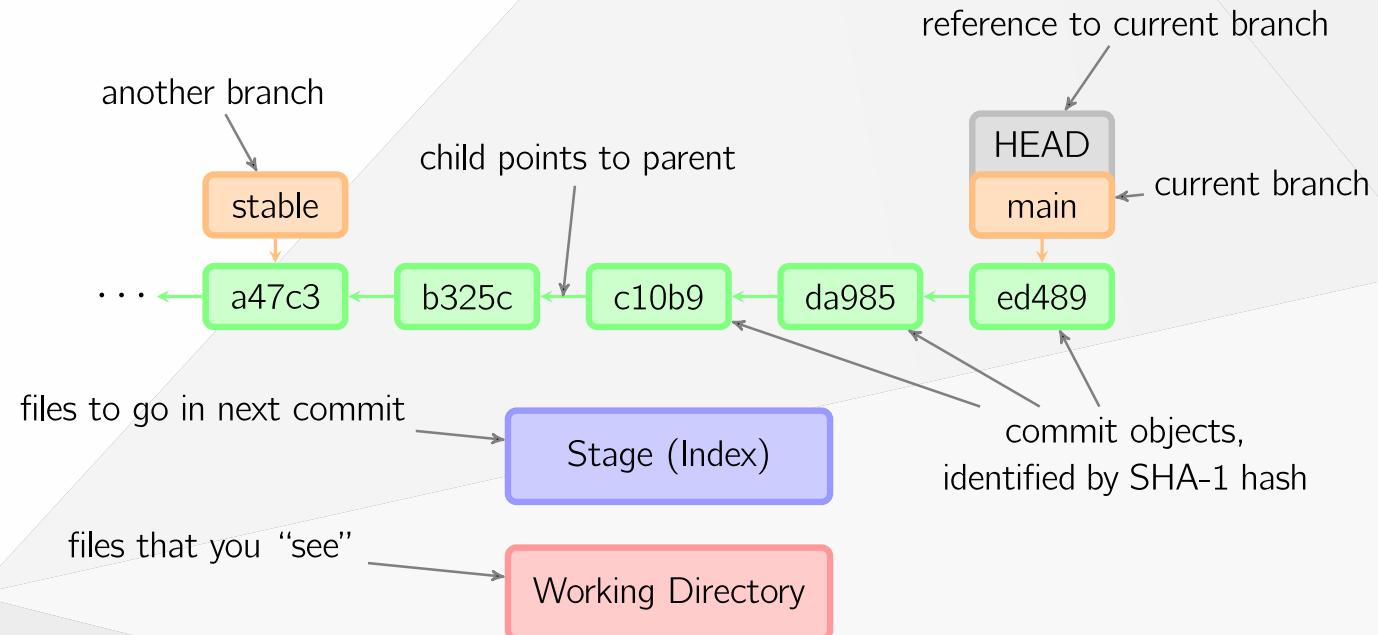
Git Status

- 使用 `git status` 確認當前目錄為 Git 儲存庫。
- 若當前目錄非 Git 儲存庫。

```
$ git status  
fatal: not a git repository (or any of the parent directories): .git
```

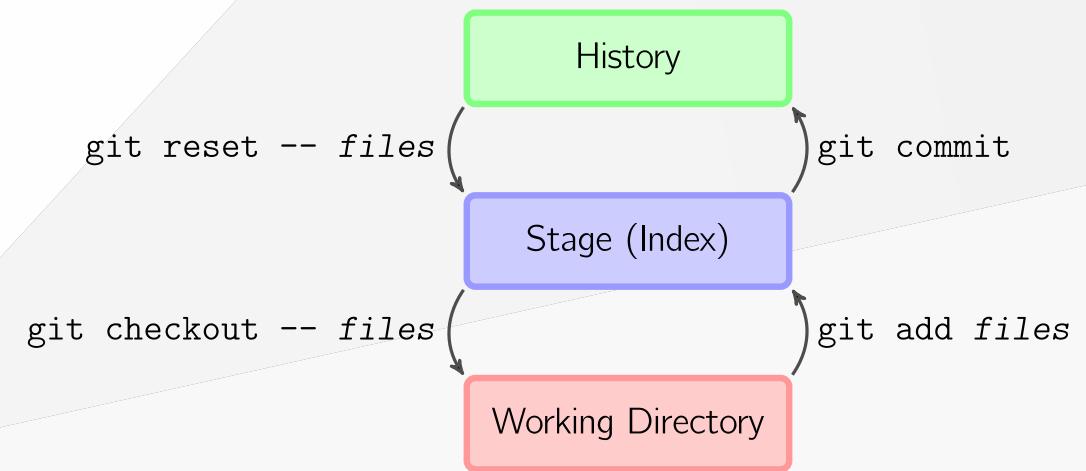
Git 的三態與歷程

- Git: 工作目錄
- Get: 暫存目錄
- Got: 儲存庫



4 個基本本地端操作命令

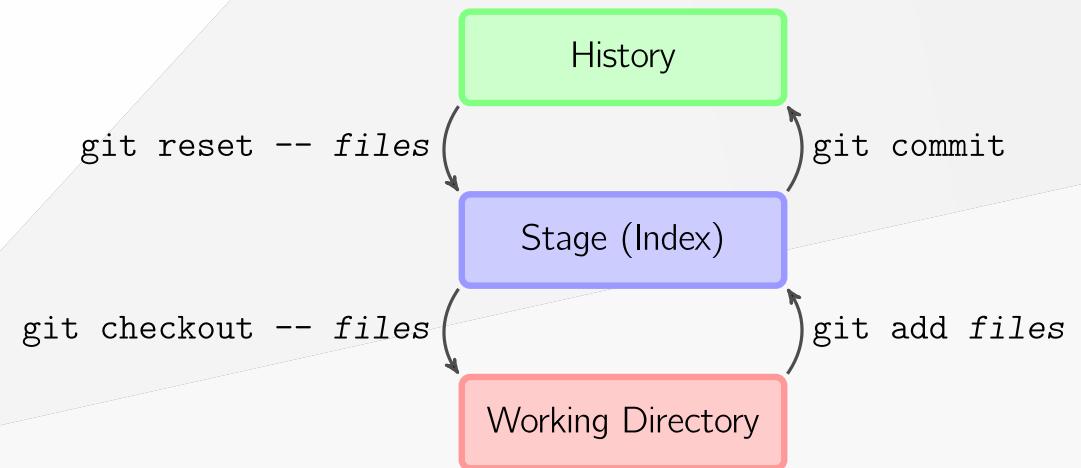
- `git add files` 把指定檔案放入暫存區域。
- `git commit` 給暫存區域產生快照並提交。
- `git reset -- files` 用來撤銷最後一次 `git add files`。
- `git checkout -- files` 把檔從暫存區域複製到工作目錄，用來放棄本地修改。



Git Add

`git add` 是一個用來將修改的文件添加到暫存區的指令。

1. `git add <file>` : 添加指定文件到暫存區
2. `git add .` : 添加所有修改的文件到暫存區
3. `git add -p` : 互動式添加修改的文件到暫存區



Git Add

git add -p 使用方式

- 當你執行 `git add -p` 時，Git 會顯示工作區中的每個修改，並讓你選擇是否要將其添加到暫存區。
- 你可以選擇 `y` (yes) 將修改添加到暫存區，選擇 `n` (no) 忽略該修改，或選擇 `s` (split) 將修改分成更小的部分。
- 互動式地選擇在工作區中修改部分內容，並將其添加到暫存區，這對於將多個邏輯上相關的修改組合到一個提交中非常有用。

- 執行 `git add marp-sample.md` 之後：

```
$ git status
On branch master

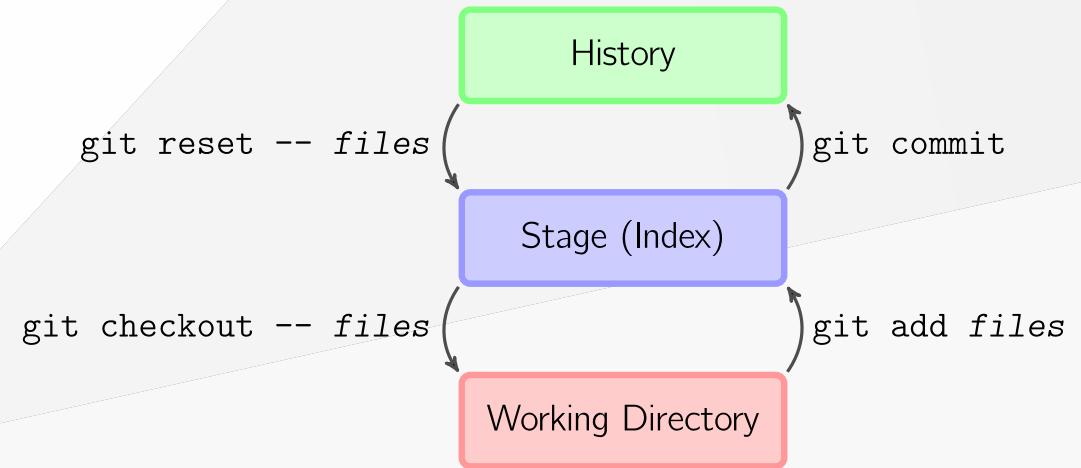
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   marp-sample.md

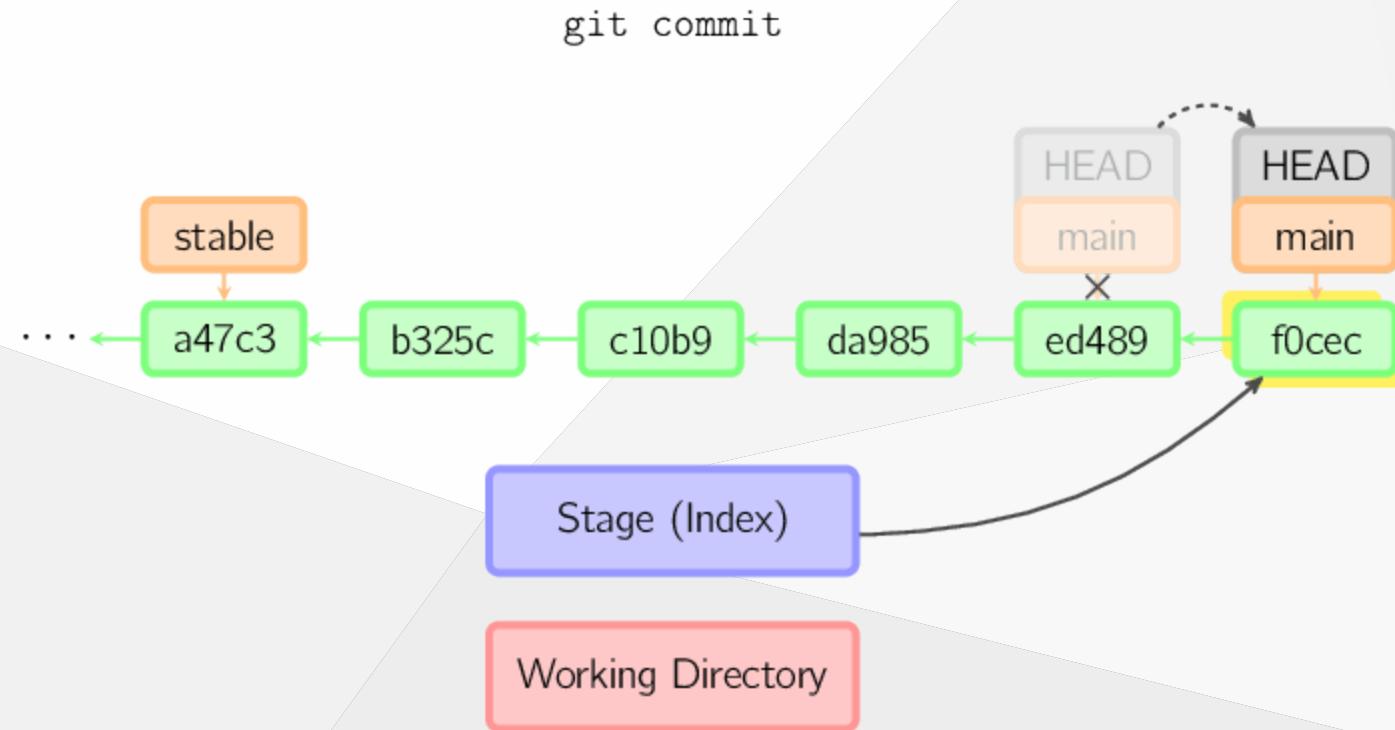
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git-get-got.md
```

Git Commit

- `git commit` 會將暫存區的所有修改提交到儲存庫。每次提交都會創建一個新的提交對象，並更新分支指針。



Git Commit



Git Commit

- 當你執行 `git commit` 時，Git 會打開一個文本編輯器，讓你輸入提交訊息。提交訊息應該清楚地說明這次提交做了什麼修改。
- 你也可以使用 `git commit -m "your message"` 直接在命令行中輸入提交訊息。
- 在[Git 遊樂園](#)裡盡情嘗試吧！

- 執行 `git commit -m "Added marp-sample"` 之後：

```
[master (root-commit) 851ed94] Added marp-sample
 1 file changed, 35 insertions(+)
 create mode 100644 marp-sample.md
```

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git-get-got.md

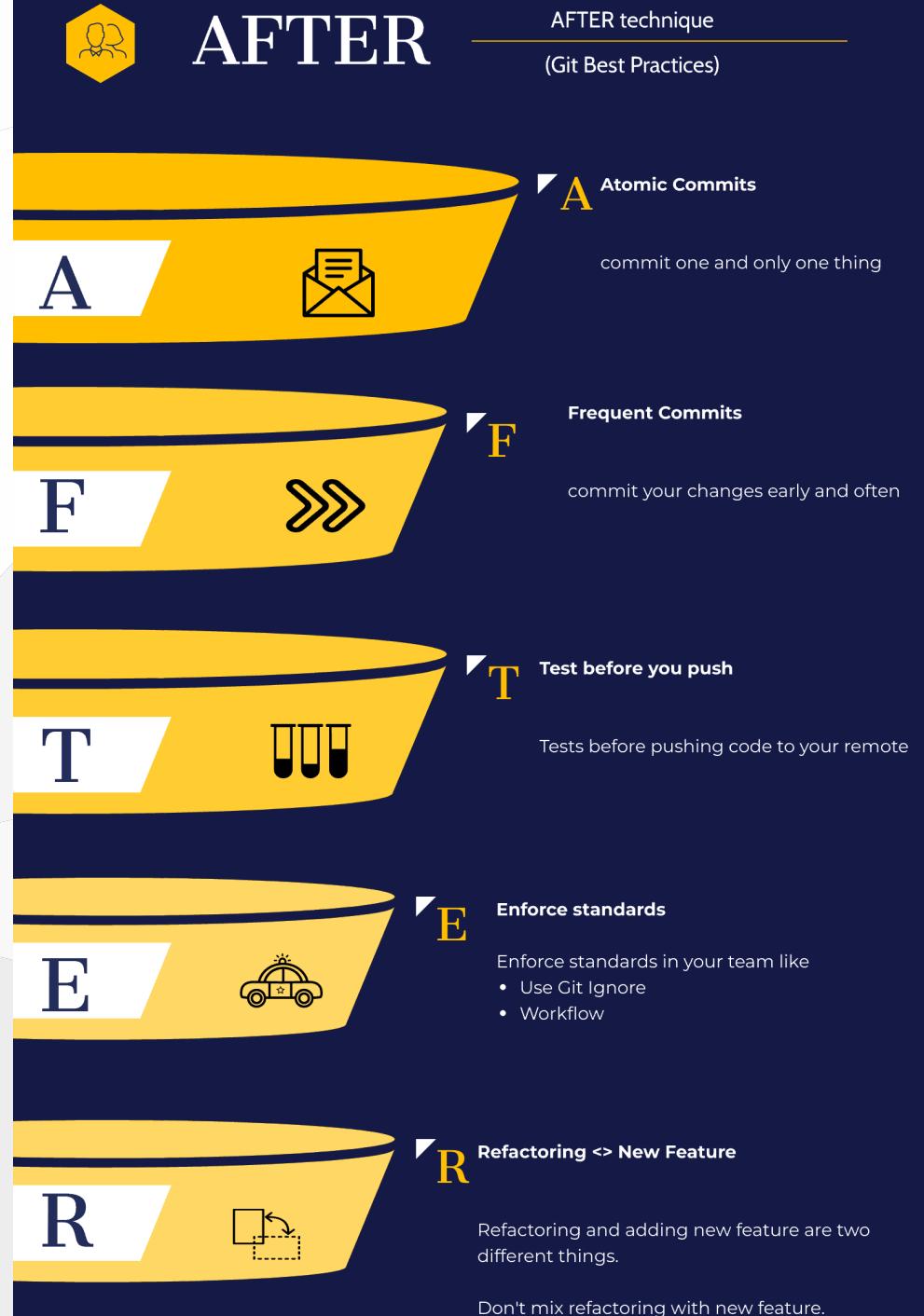
nothing added to commit but untracked files present (use "git add" to track)
```

Git 的提交策略

1. **原子提交**: 每次提交只包含一個獨立的變更，這樣可以使得每個提交都容易理解，並且在需要時可以很容易地回滾。

2. **提交早且頻繁**: 這樣可以使得其他開發者更容易跟上你的工作進度，並且可以降低合併衝突的風險。

3. **不要提交未經測試的工作**: 確保每次提交的代碼都是經過測試且可運行的，這樣可以使追蹤錯誤更加容易。



提交訊息的最佳實踐(1/2)

1. 明確的語言:

- ✗ 不好的範例: "修正錯誤"
- ✓ 好的範例: "修正購物車數量無法更新的錯誤"

2. 使用現在時態:

- ✗ 不好的範例: " Added feature to sort items"
- ✓ 好的範例: " Add feature to sort items"

提交訊息的最佳實踐(2/2)

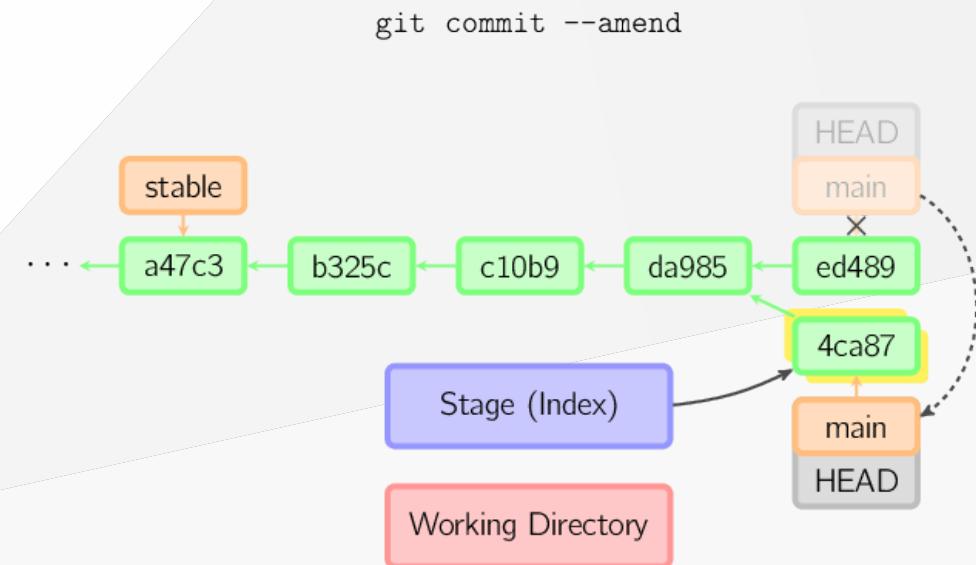
3. 首行為標題，後面為詳細說明:

-  不好的範例: "Fix bug. The bug was causing the app to crash when the user clicked on the 'Save' button."
-  好的範例:

“ Fix crash on 'Save' button click
The bug was causing the app to crash when the user clicked on
the 'Save' button. ”

修改提交

- `git commit --amend` 用於修改最後一次提交。
- 當你執行這個命令時，它會將暫存區中的變更與最後一次提交合併，並創建一個新的提交來替代原先的提交。
- 請注意，使用 `--amend` 修改已經推送的提交可能會對其他協作者造成困擾，因為它會改變提交歷史。



Git Log

- 使用 `git log` 命令查看提交歷史。
- 加上 `-n` 限制顯示的數量，其中 `n` 為整數。

```
$ git log -1  
commit 1552525ddafb48d8bc570dc794c1908ae22c4266 (HEAD -> master)  
Author: FF9999 <FF9999@example.com.tw>  
Date:   Tue May 28 20:28:25 2024 +0800
```

```
Added .gitignore
```

Git Log

- 使用 `--oneline` 選項精簡顯示。
- 使用 `--stat` 選項顯示增刪紀錄。

```
$ git log --oneline --stat
1552525 (HEAD -> master) Added .gitignore
  .gitignore | 4 +++
  1 file changed, 4 insertions(+)
851ed94 Added marp-sample
  marp-sample.md | 35 ++++++=====
  1 file changed, 35 insertions(+)
```

Git Log

- 使用 `--pretty` 選項訂製顯示格式。
- 使用 `--date` 選項修改日期格式。

```
$ git log -2 --pretty=format:"%h %ad | %s %d [%an]" --date=short  
1552525 2024-05-28 | Added .gitignore (HEAD -> master) [FF9999]  
851ed94 2024-05-02 | Added marp-sample [FF9999]
```

Git Tag

- 用於創建、列出、刪除標籤對象。
- 使用時機
 - **版本發佈**：當你準備好發佈一個新版本時，可以使用標籤來標記該版本。
 - **里程碑**：在達到開發過程中的重要階段時，可以創建標籤以記錄這一事件。

Git Tag

輕量標籤

“ 輕量標籤是一個簡單的指向特定提交的引用，不會儲存額外的資訊。 ”

- **創建輕量標籤**：`git tag <tag-name>`
 - 如果沒有指定提交 (commit)，Git 會自動將標籤指向最新的提交。

Git Tag

附註標籤

“ 附註標籤會儲存成一個完整的物件，在 Git 的資料庫中會被計算效驗碼。它包含建立標籤的人的名字、電子郵件和建立日期，以及標籤訊息。

- **創建附註標籤** : `git tag <tag-name> -a -m "Message content"`

- `-a` 用於創建一個附註標籤。
- `-m` 用於添加一個訊息。
- 如果沒有使用 `-m` , Git 會啟動編輯器以輸入標籤訊息。

Git Tag

查詢與刪除標籤

- **查詢標籤** : `git tag`
 - 列出所有標籤。
- **查詢特定標籤尋錫** : `git show <tag-name>`
 - 顯示特定標籤訊息。
- **刪除標籤** : `git tag -d <tag-name>`
 - 移除指定的標籤。

Git Tag

查詢標籤

```
$ git show git-slides-v01  
tag git-slides-v01  
Tagger: FF9999 <FF9999@example.com.tw>  
Date: Mon Jun 3 19:48:56 2024 +0800
```

Added git tag command

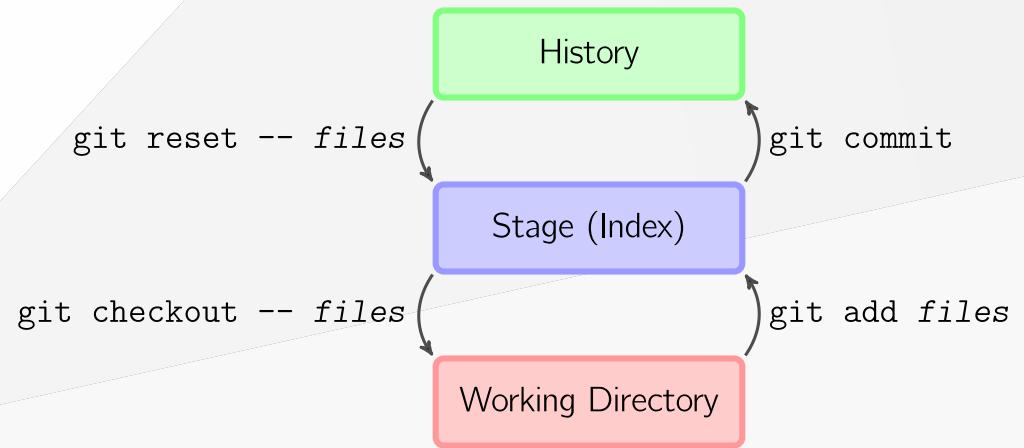
```
commit 919f6bee71fad71cc011f3e2b25e7ed15b667443 (HEAD -> master, tag: git-slides-v01)  
Author: FF9999 <FF9999@example.com.tw>  
Date: Mon Jun 3 19:14:48 2024 +0800
```

commit **vs** tag

| 特性 | git commit | git tag |
|----|-----------------|----------------------------------|
| 定義 | 提交代表對代碼庫的一次快照。 | 標籤用於標記提交歷史中的重要點，如版本發佈。 |
| 用途 | 紀錄代碼變更。 | 標記特定的提交，如版本號。 |
| 變動 | 提交會隨著新的變更而進行更新。 | 標籤一旦創建，通常不會變動。 |
| 資訊 | 包含作者、日期和變更訊息。 | 附註標籤可以包含創建者信息、日期和訊息。輕量標籤不包含額外資訊。 |

Git Reset

- `git reset` 會將 HEAD 指針移動到指定的提交，並可以選擇性地更新暫存區和工作區以匹配該提交。
- 在[Git 遊樂園](#)裡盡情嘗試吧！



git reset 有三種模式：

不要輕易使用!

1. `git reset --soft <commit>`：將 HEAD 移動到 `<commit>`，但保留暫存區和工作區的修改。
2. `git reset --mixed <commit>`：將 HEAD 和暫存區移動到 `<commit>`，但保留工作區的修改。執行 `git reset` 時的預設行為。
3. `git reset --hard <commit>`：將 HEAD、暫存區和工作區都移動到 `<commit>`，但要小心使用，因為 `git reset --hard` 可能會丟失修改。

Git Revert

- `git revert` 是一個用來「恢復 commit」的指令。他的本意會把某個 commit 「反著做」。不會動到 commit history，可以抽掉不想要的 commit，而不會動到其他已經 commit 的部分。
- 在[Git 遊樂園](#)裡盡情嘗試吧！

```
git revert <commit-hash> #還原特定的commit  
git revert HEAD #還原最近的一個commit
```

還原最近一次提交

- 執行最近一次提交

```
$ git commit -m "Added git snapshot to git slides"  
[master e1772a9] Added git snapshot to git slides  
 1 file changed, 74 insertions(+), 8 deletions(-)
```

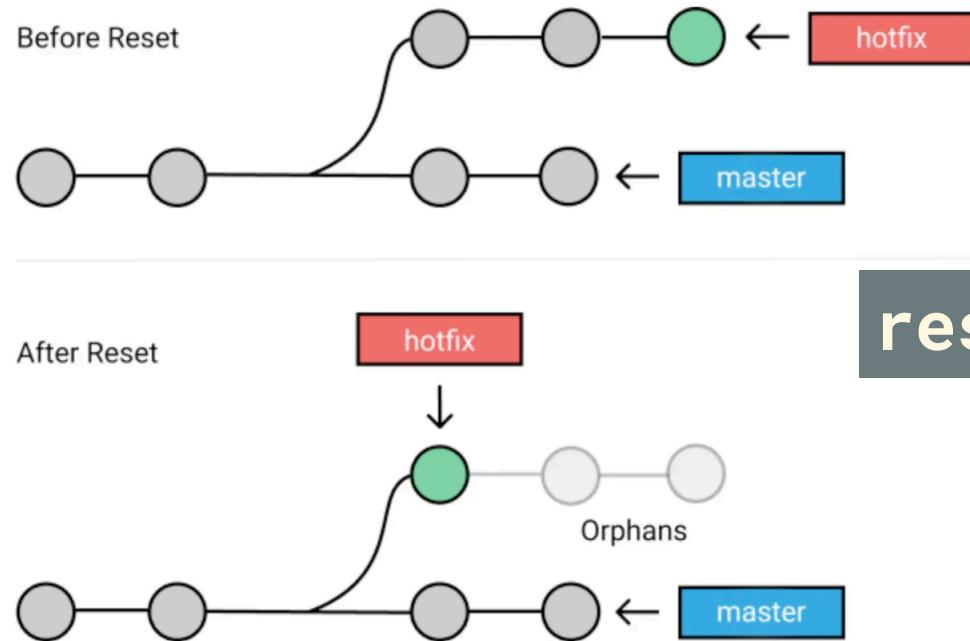
- 還原最近一次提交

```
$ git revert HEAD  
[master c2265e9] Revert "Added git snapshot to git slides"  
 1 file changed, 8 insertions(+), 74 deletions(-)
```

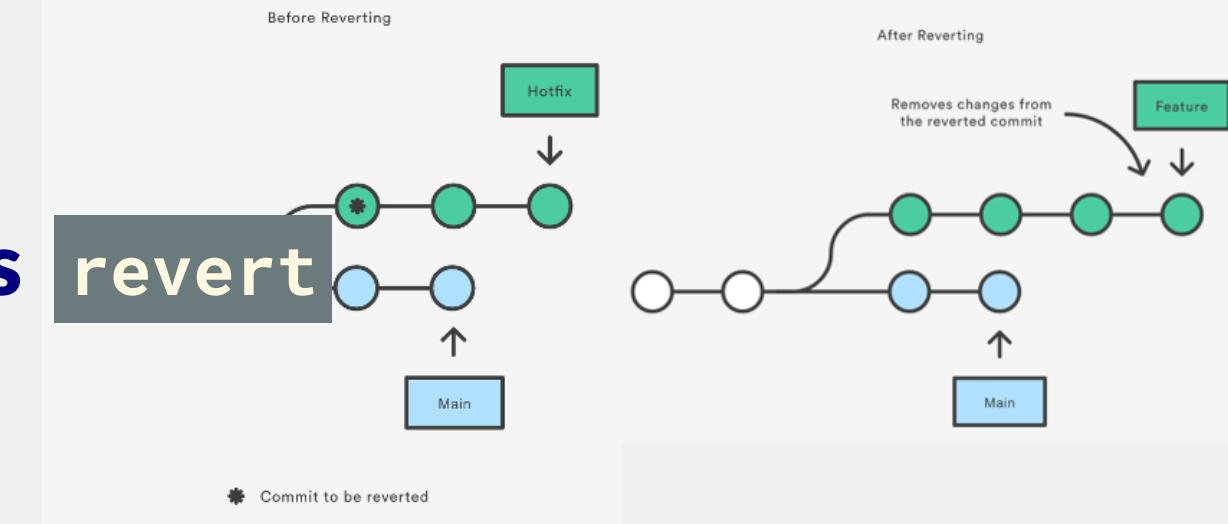
撤銷還原

- 目標：回到執行 `git revert HEAD` 之前的狀態，並且丟棄所有工作目錄中的變更。
- 命令

```
$ git reset --hard HEAD~1  
HEAD is now at e1772a9 Added git snapshot to git slides
```

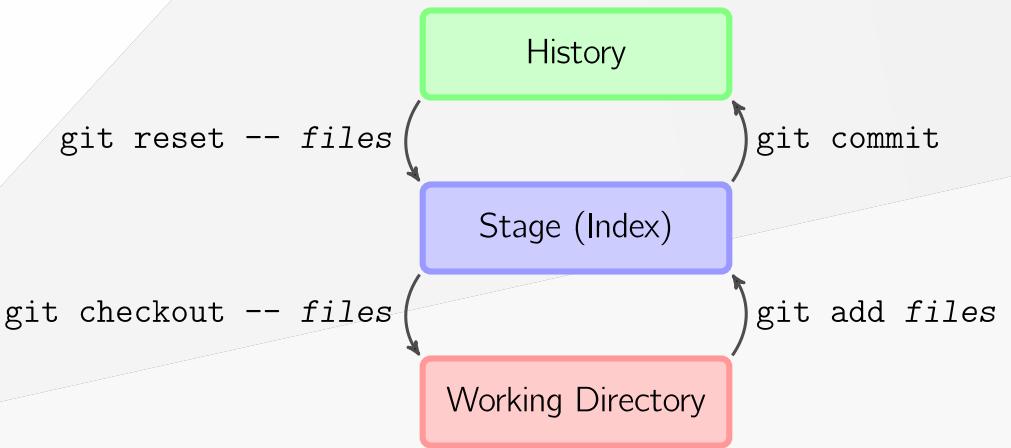


reset vs revert



Git Checkout

`git checkout` 是一個用來切換分支或恢復工作區文件的指令。



Git Checkout

`git checkout` 有三種常見的使用方式：

1. `git checkout <branch>` : 切換到 `<branch>` 分支
2. `git checkout <commit_hash>` : 切換到 `<commit_hash>` 提交。
3. `git checkout -- <file>` : 恢復 `<file>` 到最近一次提交的狀態。是一個**危險**的命令，因為它會丟棄所有未提交的變更。經常用來撤銷尚未加入暫存區的修改。

當前提交紀錄

```
$ git log --oneline
e1772a9 (HEAD -> master) Added git snapshot to git slides
919f6be (tag: git-slides-v01) Added git-get-got
1552525 Added .gitignore
851ed94 Added marp-sample
```

切換到特定提交紀錄

- 使用 哈希值

```
git checkout 919f6be  
Note: switching to '919f6be'.
```

You are **in 'detached HEAD'** state. You can look around, make experimental changes and commit them, and you can discard any commits you make **in** this state without impacting any branches by switching back to a branch.

...

```
HEAD is now at 919f6be Added git-get-got
```

切換到特定提交紀錄

- 使用 HEAD的相對位置

```
git checkout HEAD~2  
Note: switching to 'HEAD~2'.
```

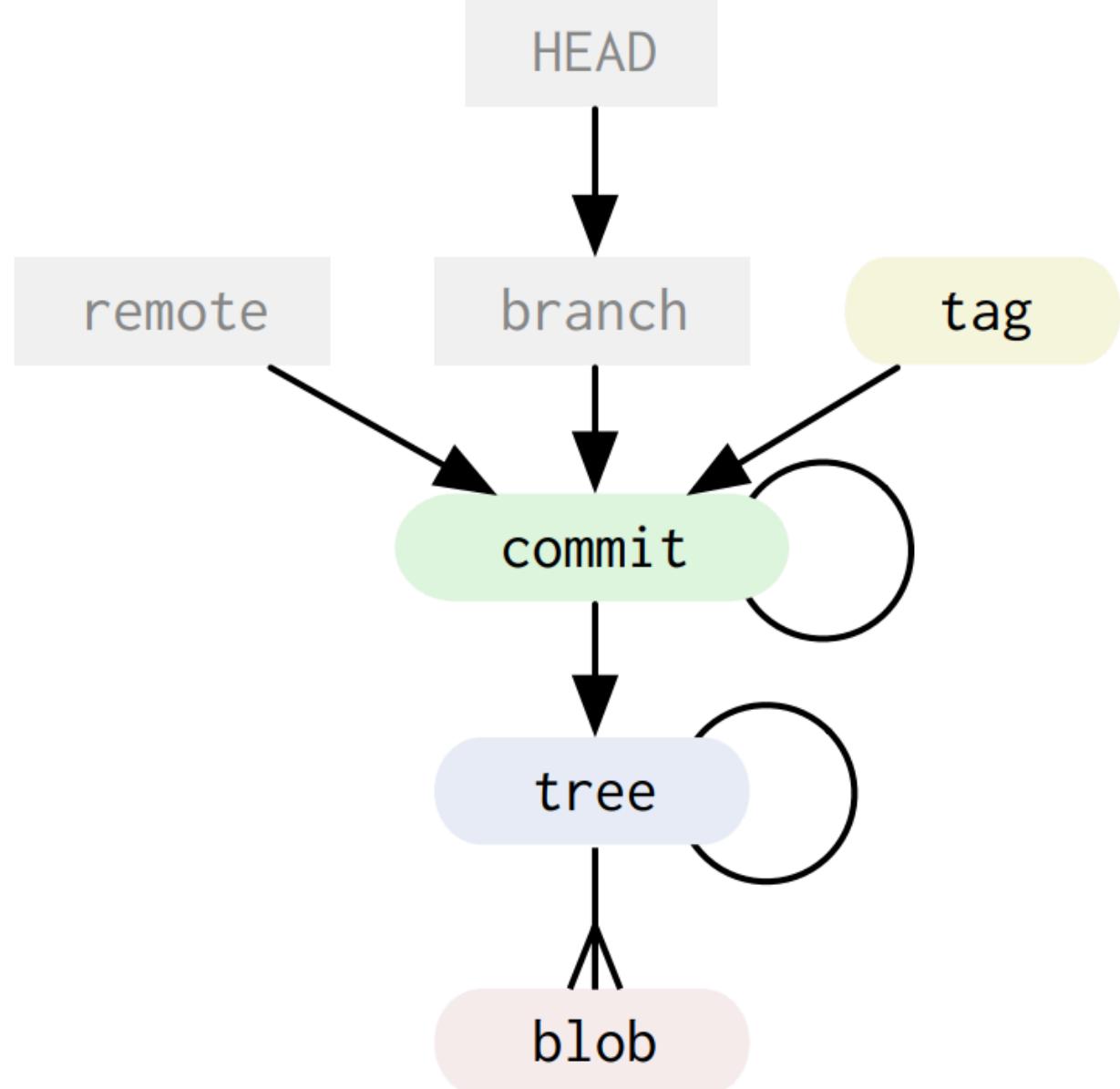
You are **in 'detached HEAD'** state. You can look around, make experimental changes and commit them, and you can discard any commits you make **in** this state without impacting any branches by switching back to a branch.

...

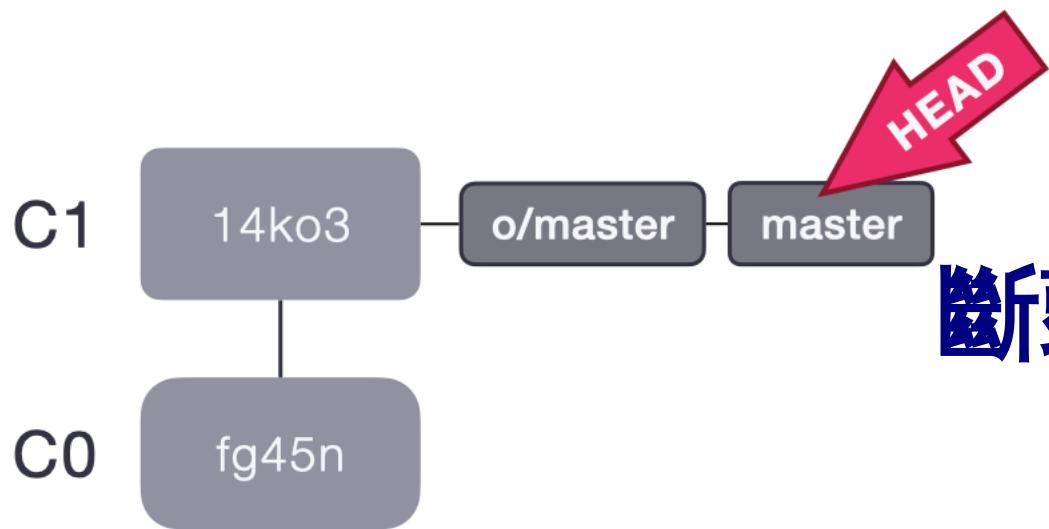
```
HEAD is now at 1552525 Added .gitignore
```

Git 重要元素

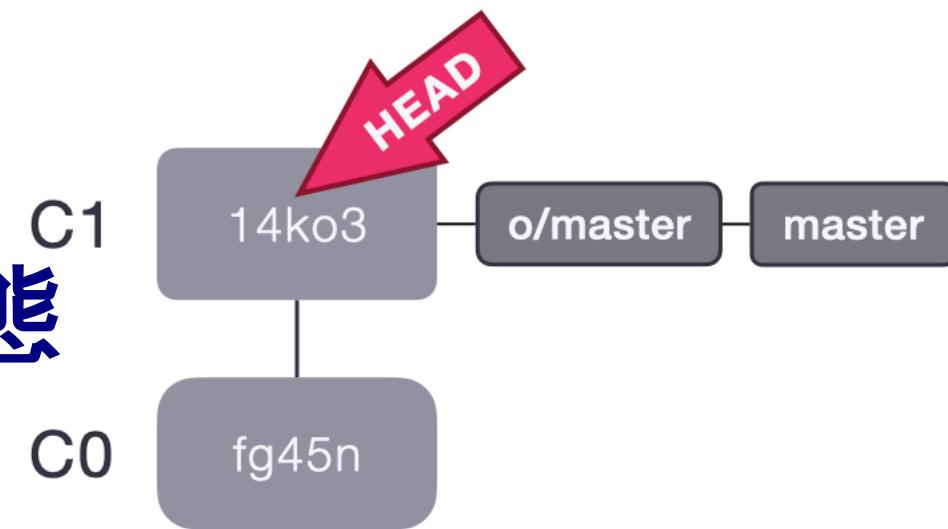
- **Commit:** 代表在特定時間點的專案狀態。
- **Branch:** 為了將修改記錄的歷史分開儲存而設計的。
- **HEAD:** 指向你當前工作中的最新提交的指標。
- **Tag:** 一種標記，用於指向特定的提交。



Attached HEAD



Detached HEAD



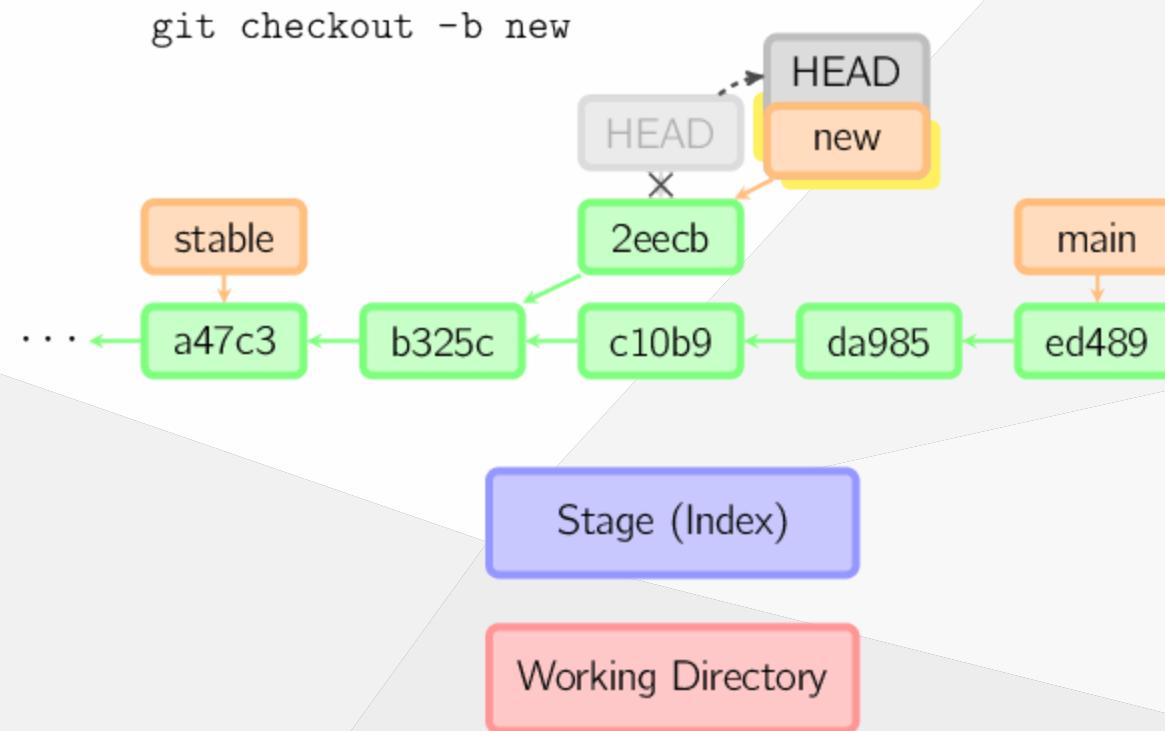
斷頭狀態

斷頭狀態

- 在 detached HEAD 狀態中，提交將不屬於任何分支，除非創建一個新分支來保存這些提交。
- 如果只是想查看特定提交的代碼，而不打算進行更改或新提交，這是安全的。
- 如果你想要在特定提交上創建一個新分支並切換到該分支，可以使用：

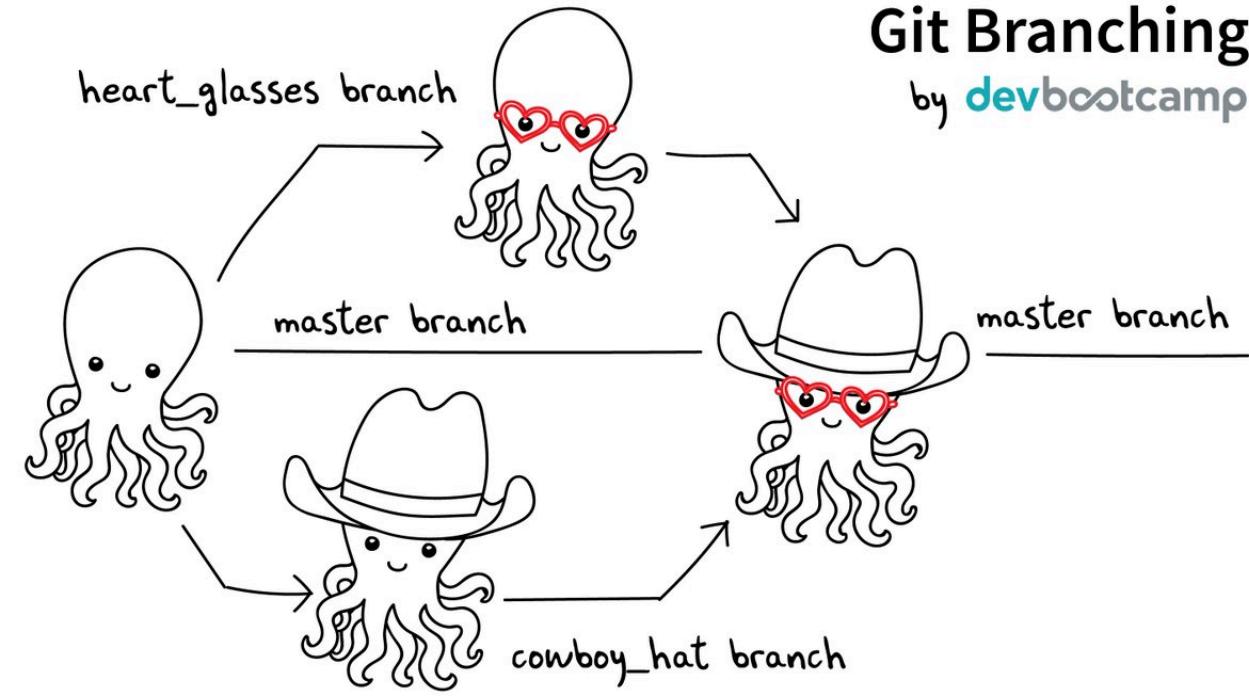
```
git checkout -b <new-branch-name> <commit-hash>
```

創建並切換分支



分支

- 是什麼？
 - 分支是指向提交的指標
 - 它允許你在不同的開發線路上工作
- 為什麼要使用？
 - 同時進行多個功能的開發
 - 隔離實驗性的更改
 - 便於協作和代碼審查



分支策略

- 功能分支：針對每個新功能創建分支
- 修復分支：針對每個錯誤修復創建分支
- 發布分支：用於準備新的產品發布

GIT, GET, GOT!

Git Branch

新增並切換分支

```
# 創建一個名為 'add-branch-section' 的新分支  
git branch add-branch-section
```

```
# 切換到 'add-branch-section' 分支  
$ git checkout add-branch-section  
Switched to branch 'add-branch-section'
```

Git Branch

使用 `git branch -a` 查看所有分支

```
git branch -a
* add-branch-section
  master
```

Git Branch

使用 `git branch -d <branch-name>` 刪除分支

```
$ git branch -d add-branch-section  
Deleted branch add-branch-section (was 4fb5873).
```

Git Merge

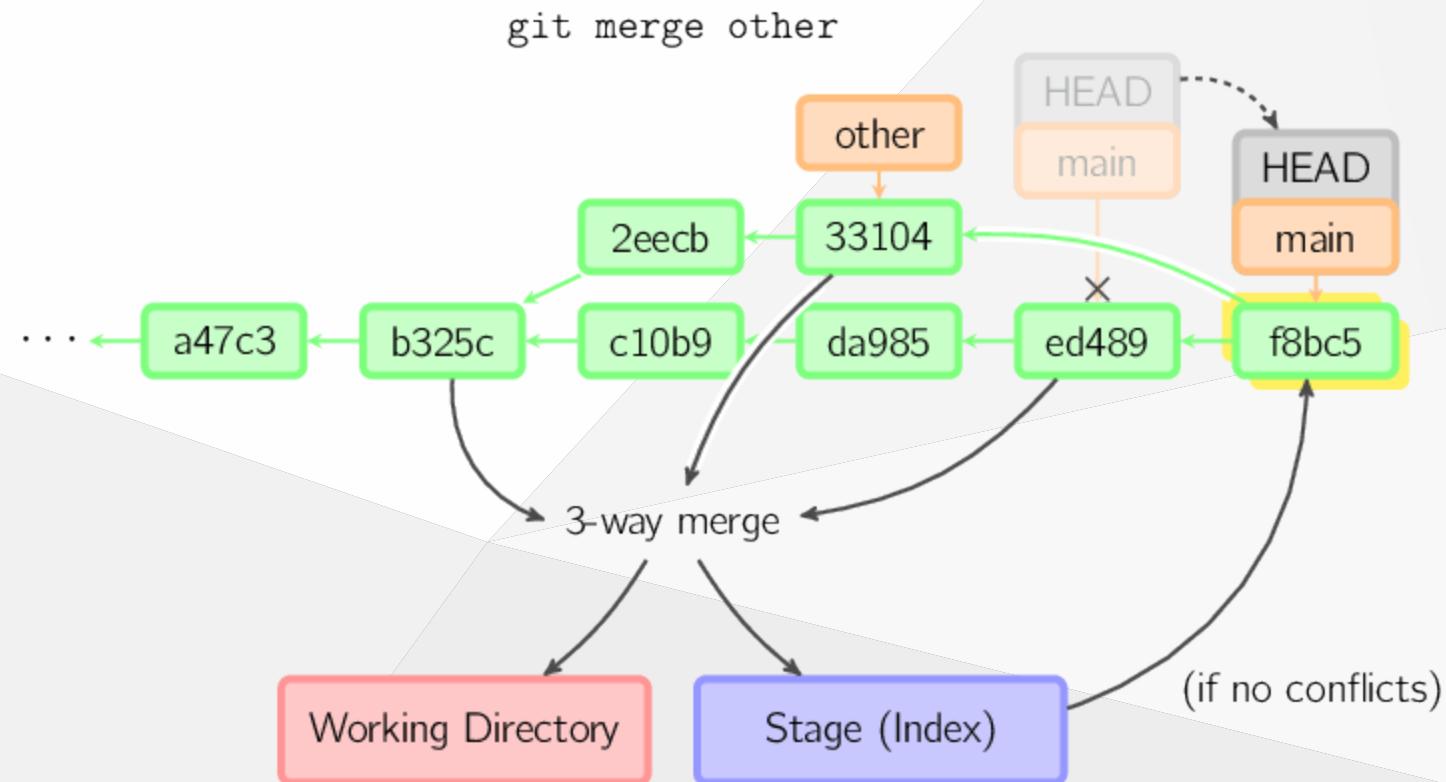
- 先切換回主分支: `git checkout master`
- 再將 `new-section` 分支的更改合併到 `master` 分支

```
git merge new-section
Updating 74fb1f5..8fa8a78
Fast-forward
 git-get-got.md | 22 ++++++=====
 1 file changed, 22 insertions(+)
```

查看合併後記錄

```
$ git log --oneline
8fa8a78 (HEAD -> master, new-section) Added git merge command in git slides
74fb1f5 Added git branch command in git slides
4fb5873 Completed section 1 through 2 in git slides
e1772a9 Added git snapshot to git slides
919f6be (tag: git-slides-v01) Added git-get-got
1552525 Added .gitignore
851ed94 Added marp-sample
```

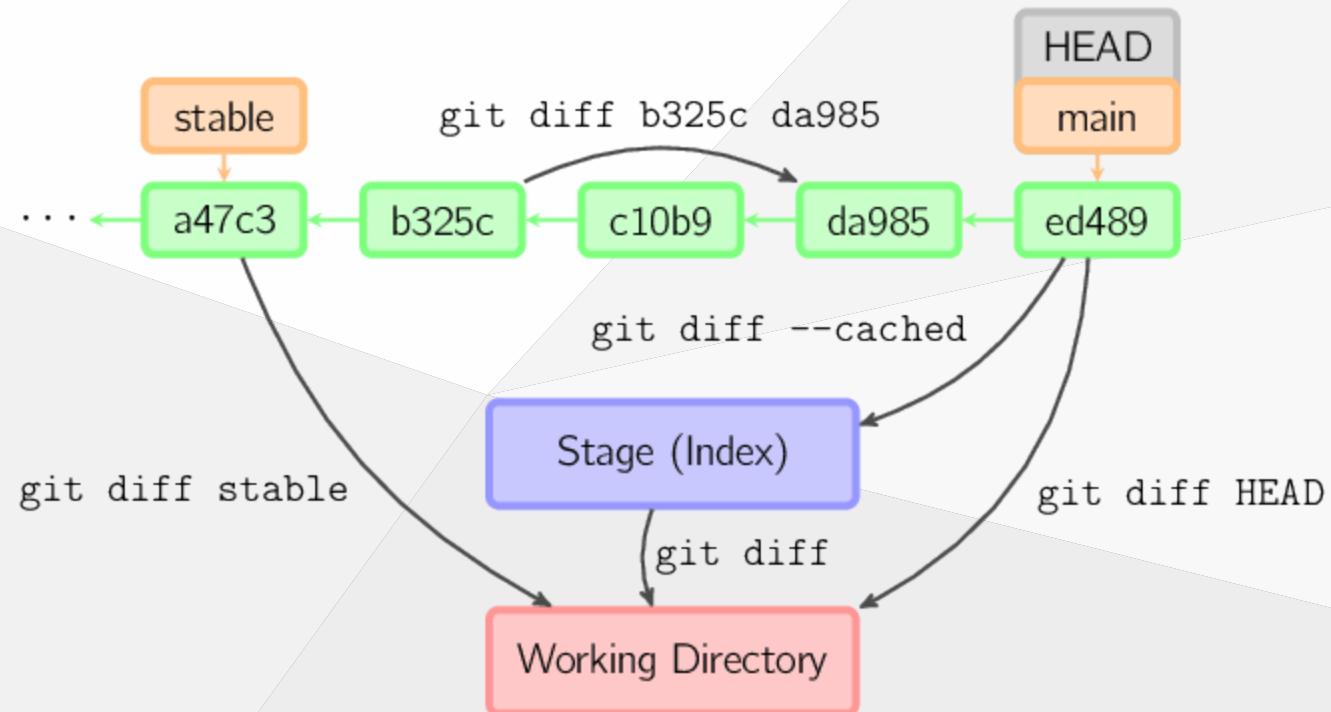
合併分支



Git Diff

- 用途
 - 用於顯示文件之間的差異
 - 可比較工作目錄、暫存區和提交之間的變更
- 使用情境
 - 查看工作目錄和暫存區的差異(git vs get) : `git diff`
 - 查看工作目錄和分支的差異 : `git diff <branch>`
 - 查看暫存區和最後一次提交的差異(get vs. got) : `git diff --cached`
 - 比較兩個提交之間的差異 : `git diff <commit1> <commit2>`

Git Diff



Git Diff

- 在新增修改至暫存區前，查看工作目錄和暫存區的差異(git vs get)

```
$ git diff
diff --git a/git-get-got.md b/git-get-got.md
index 79f204d..c2fd7e6 100644
--- a/git-get-got.md
+++ b/git-get-got.md
@@ -690,16 +690,30 @@ Deleted branch add-branch-section (was 4fb5873).
```

```
# Git Merge
-- 切換回主分支
+- 先切換回主分支：`git checkout master`
+- 再將`new-section`分支的更改合併到`master`分支
```

Git Diff

- 在新增修改至暫存區後，查看暫存區和最後一次提交的差異(get vs. got)

```
$ git diff --cached
diff --git a/git-get-got.md b/git-get-got.md
index 79f204d..c2fd7e6 100644
--- a/git-get-got.md
+++ b/git-get-got.md
@@ -690,16 +690,30 @@ Deleted branch add-branch-section (was 4fb5873).
```

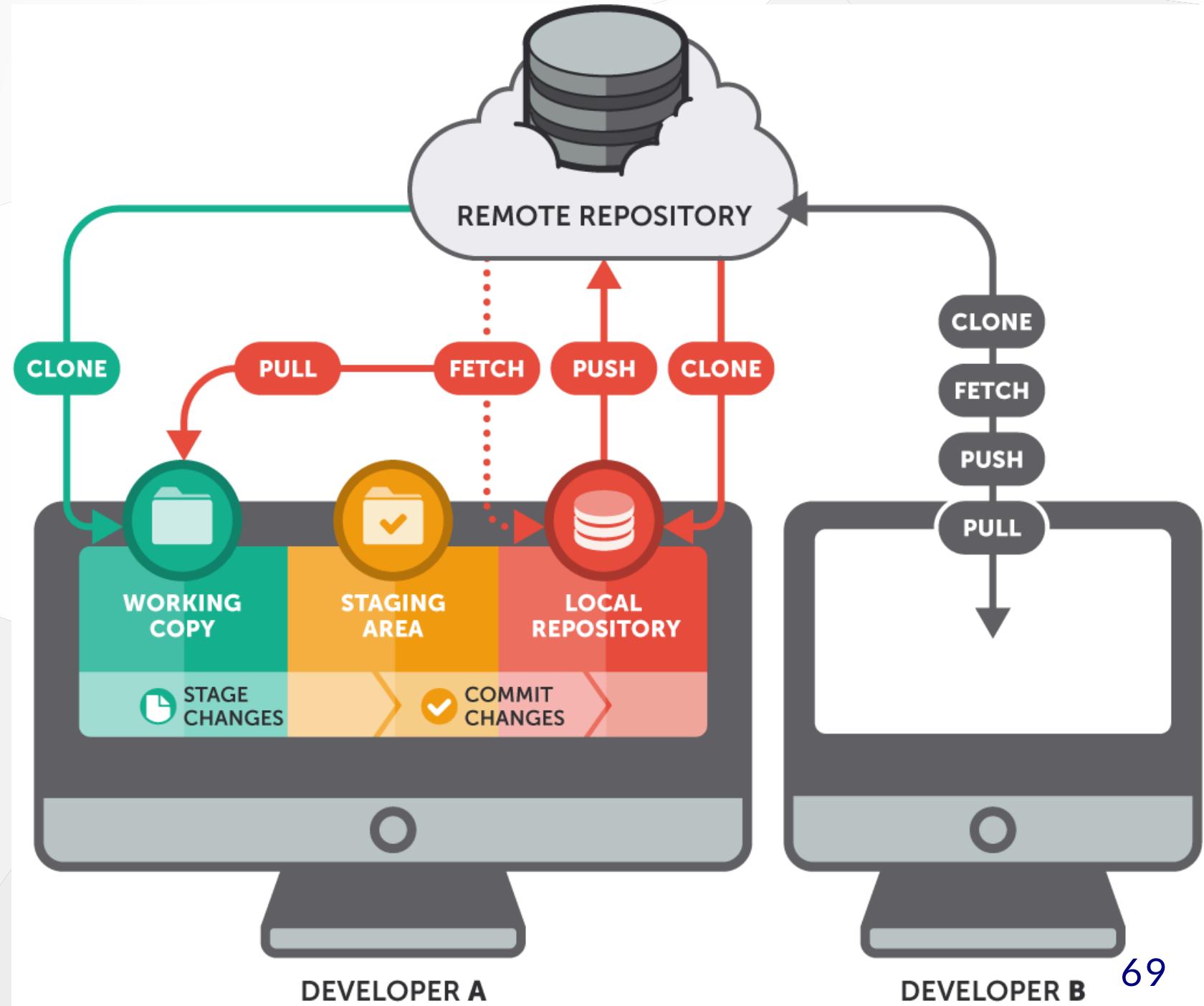
```
# Git Merge
-- 切換回主分支
+- 先切換回主分支：`git checkout master`
+- 再將`new-section`分支的更改合併到`master`分支
```

- Remote > Local

- clone
- fetch
- pull

- Local > Remote

- push



添加 SSH 密鑰

1. 在本地機器上生成 RSA 密鑰對。

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

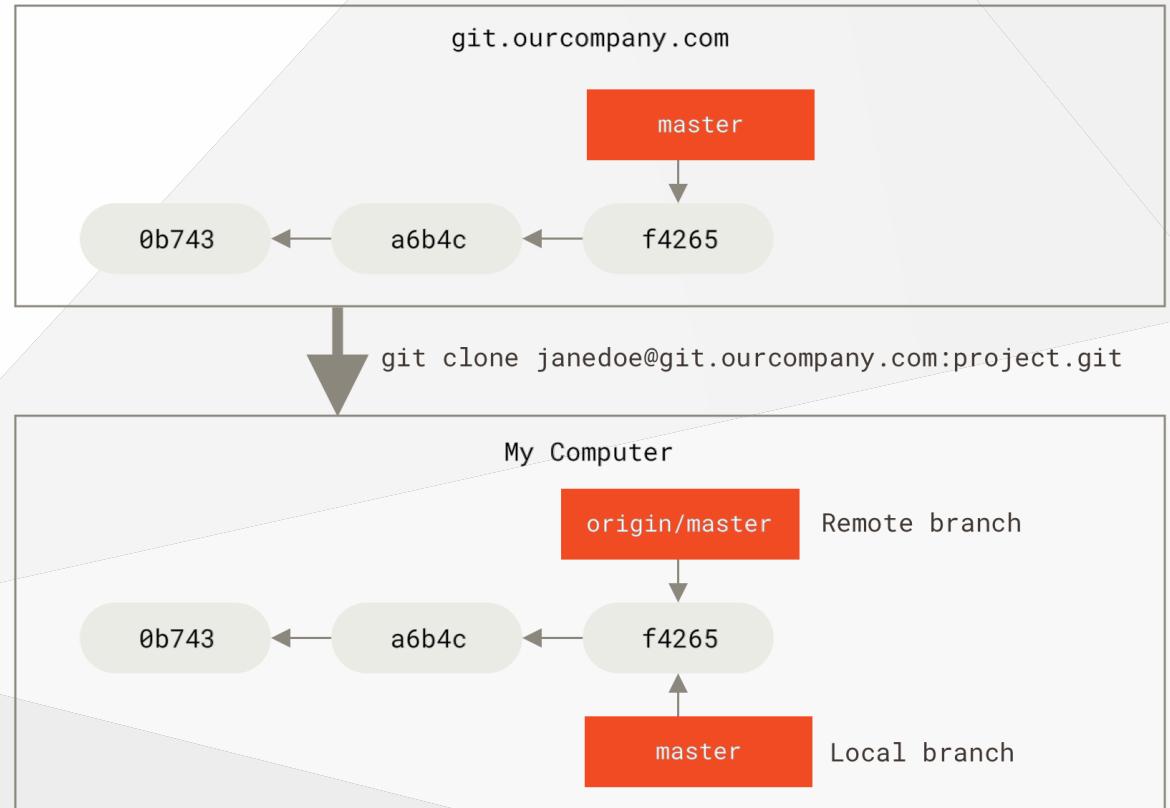
2. 將公鑰添加到 Git Server，使用以下命令打開公鑰文件。

```
cat ~/.ssh/id_rsa.pub
```

3. 複製公鑰，並貼到 Git Server 中“SSH 密鑰”區塊的“添加公鑰”表單中。

Git Remote

- `git remote` 是一個用來管理遠端儲存庫的指令，可以用來查看遠端儲存庫的列表，或添加、移除、重命名遠端儲存庫。



Git Remote

git remote 的常用指令

- `git remote add <remote-name> <remote-url>` : 添加一個新的遠端儲存庫
- `git remote -v` : 查看遠端儲存庫的列表
- `git remote remove <remote-name>` : 移除一個遠端儲存庫
- `git remote rename <old-name> <new-name>` : 重新命名遠端儲存庫

Git Remote

- 新增遠端空白儲存庫

```
$ git remote add origin http://ff9999.com.tw:3000/gitea/marp-slides.git
```

- 檢視遠端儲存庫

```
$ git remote -v
origin  http://ff9999.com.tw:3000/gitea/marp-slides.git (fetch)
origin  http://ff9999.com.tw:3000/gitea/marp-slides.git (push)
```

Git Remote

預設遠端儲存庫的名稱為 origin

- 使用以下指令來查詢預設的遠端儲存庫：

```
git remote show origin
```

- 使用以下指令來修改預設的遠端儲存庫：

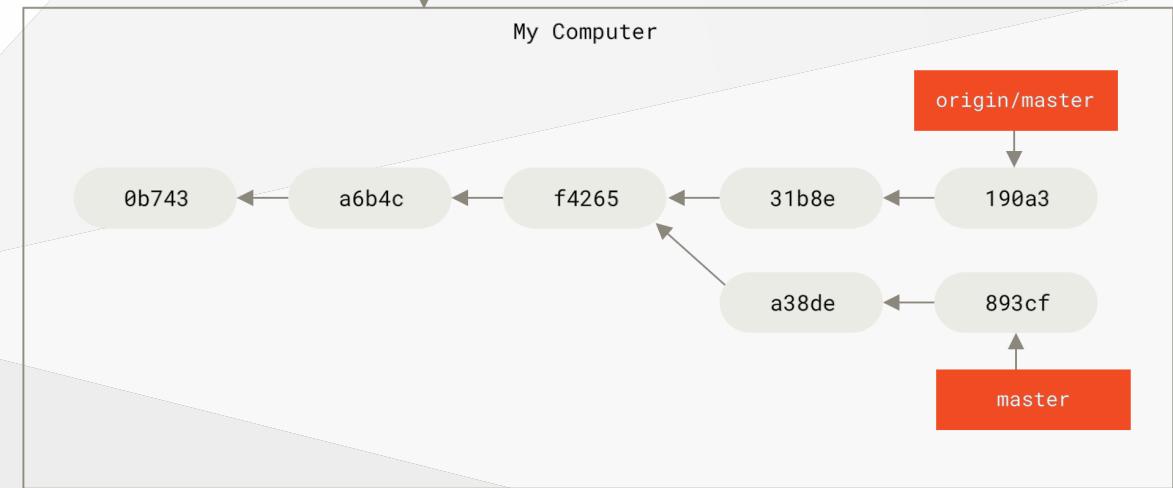
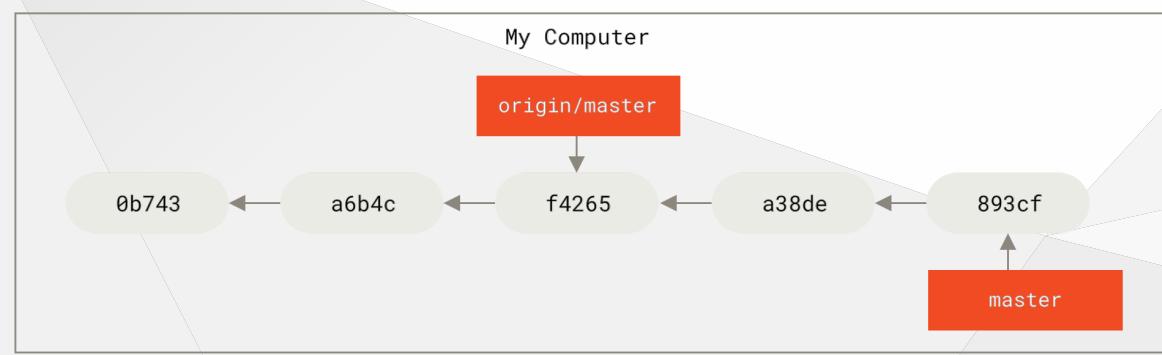
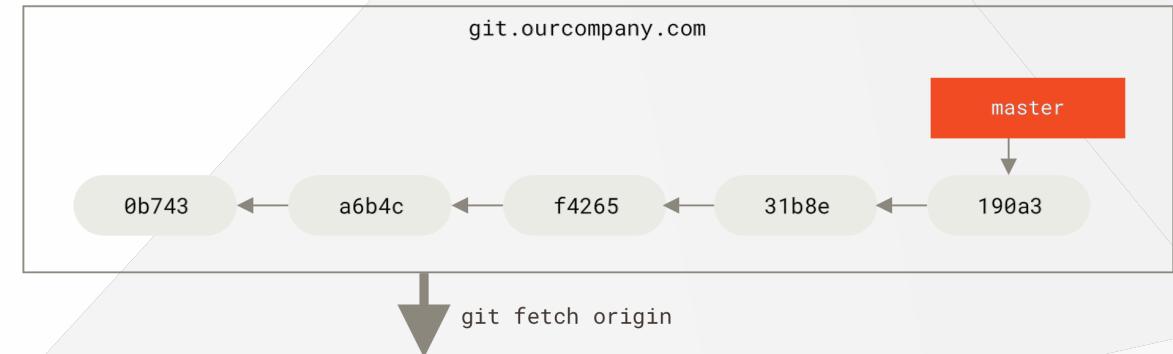
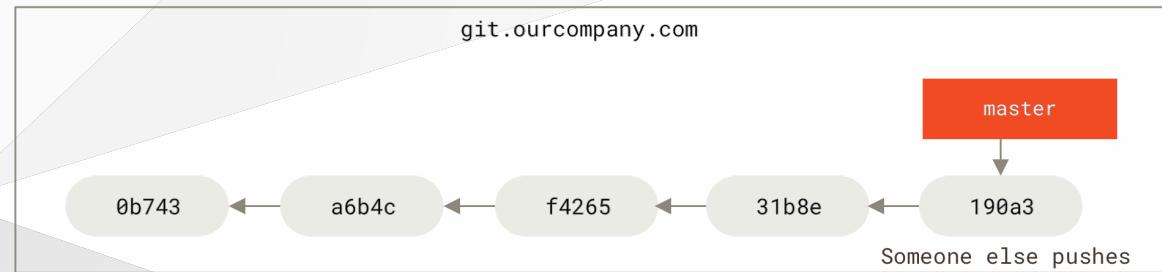
```
git remote set-url origin <new-url>
```

Git Fetch

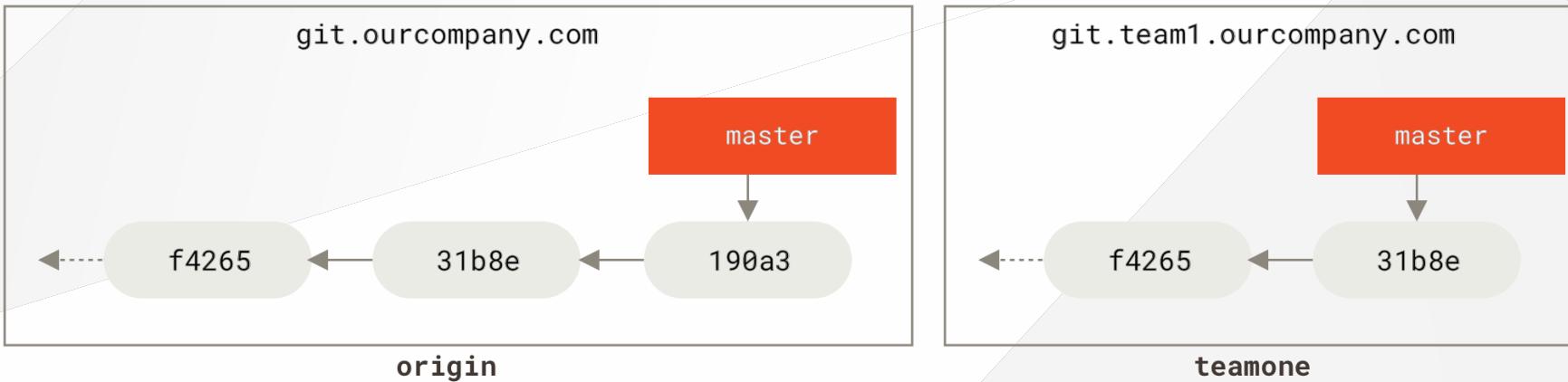
- `git fetch` 是一個用來從遠端儲存庫獲取變更，但不自動合併的指令。
- 你可以在 `git fetch` 之後使用 `git merge` 來合併變更。
- `git fetch` 預設會從設定的遠端儲存庫獲取所有分支的變更。

```
git fetch <remote-name>
```

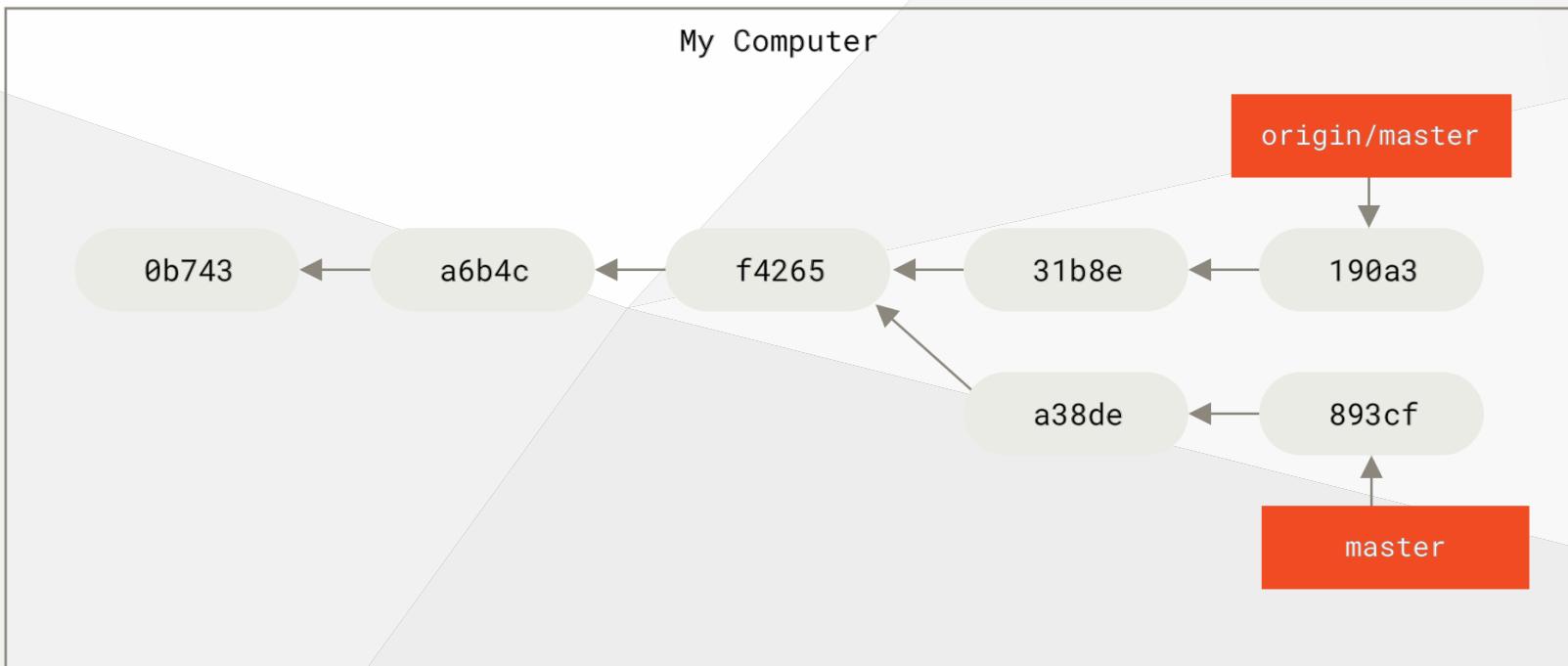
GIT, GET, GOT!



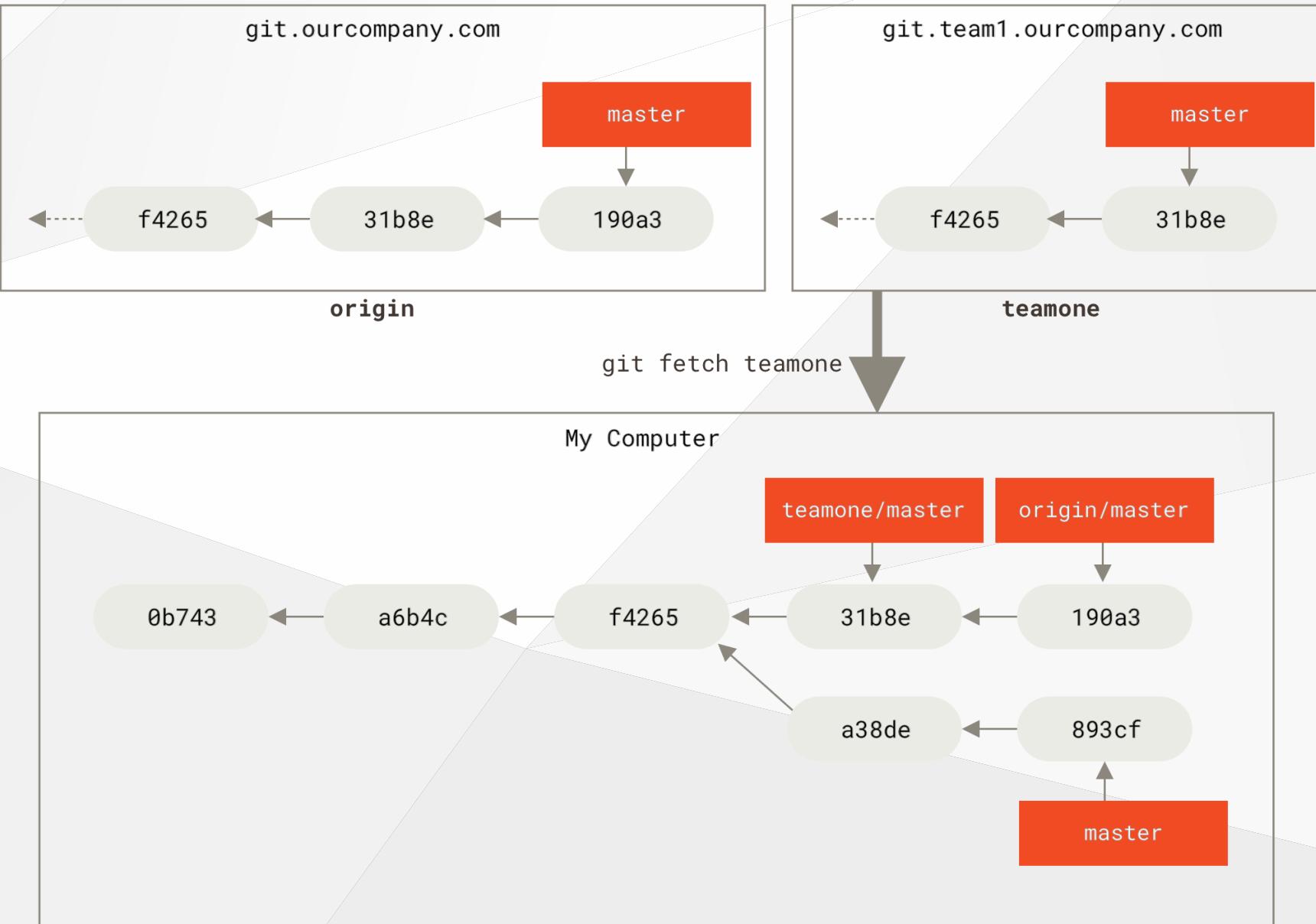
GIT, GET, GOT!



git remote add teamone git://git.team1.ourcompany.com



GIT, GET, GOT!

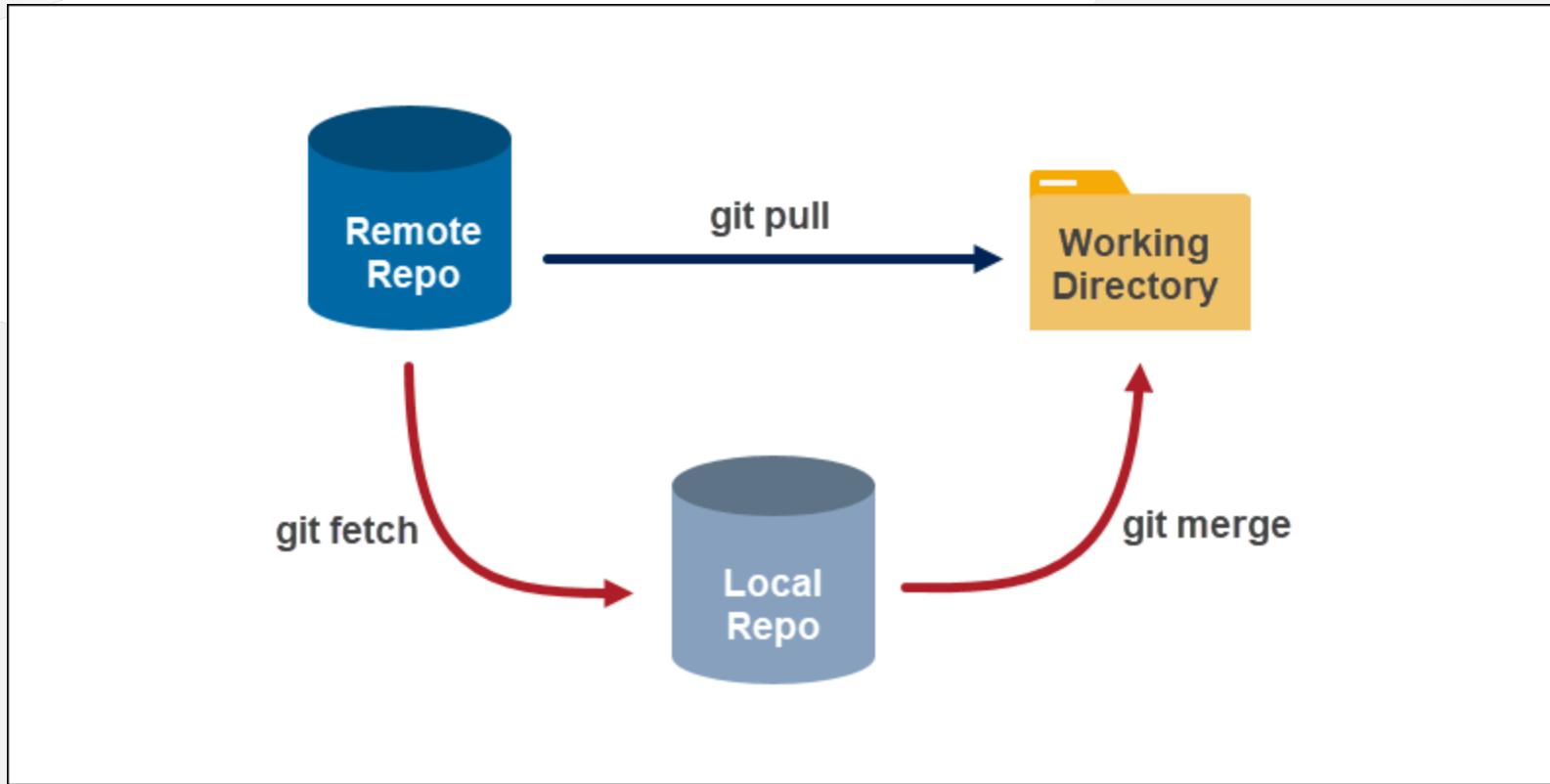


Git Pull

- `git pull` 用來從遠端儲存庫的分支獲取變更並自動合併到當前分支。

```
git pull <remote-name> <branch-name>
```

git pull 與 git fetch 的差異



Git Push

- 將本地的變更推送到遠端儲存庫的指令。
 - `git push origin main`: 這個命令會將你的本地 `main` 分支的變更推送到遠端倉庫 `origin` 的 `main` 分支。如果遠端倉庫 `origin` 的 `main` 分支已經存在，那麼這個命令會更新這個分支。如果遠端倉庫 `origin` 的 `main` 分支不存在，那麼這個命令會創建這個分支。
 - `git push -u origin main`: 這個命令與上面的命令相同，但是它還會設定遠端倉庫 `origin` 的 `main` 分支為你的本地 `main` 分支的上游分支 (`up-stream`)。這意味著在未來，當你在本地 `main` 分支上執行 `git pull` 或 `git push` 命令時，Git 會知道這些命令應該與哪個遠端分支進行交互。

Git Push

- 初次推送至遠端空白儲存庫

```
$ git push -u origin main
Enumerating objects: 33, done.
Counting objects: 100% (33/33), done.
Delta compression using up to 8 threads
Compressing objects: 100% (31/31), done.
Writing objects: 100% (33/33), 14.95 KiB | 478.00 KiB/s, done.
Total 33 (delta 17), reused 0 (delta 0), pack-reused 0
remote: . Processing 1 references
remote: Processed 1 references in total
To http://ff9999.com.tw:3000/gitea/marp-slides.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

協同開發流程(1/2)

1. `git clone <repository>` : 初級工程師 從遠端存儲庫複製代碼到本地。
2. `git checkout -b <branch-name>` : 初級工程師創建一個新的分支來開發新功能或修復 bug。
3. `git add .` : 初級工程師將修改的文件添加到暫存區。
4. `git commit -m "commit message"` : 初級工程師將暫存區的更改提交到當前分支。
5. `git push origin <branch-name>` : 初級工程師將本地分支的更改推送到遠端存儲庫。
6. 在 Gitea 或其他 Git 平台上，初級工程師創建一個新的 Pull Request。

協同開發流程(2/2)

7. `git checkout <branch-name>` : 主任工程師 切換到該分支以進行審查。
8. `git pull origin <branch-name>` : 主任工程師拉取最新的更改。
9. 主任工程師審查代碼，並在 Pull Request 中留下評論。
10. `git merge <branch-name>` : 如果代碼審查通過，主任工程師將分支合併到主分支。
11. `git push origin master` : 主任工程師將合併後的更改推送到遠端的主分支。

處理檔案

- 移動檔案
- 刪除檔案



Git Mv

- `git mv` 是一個用來移動或重命名工作區中的文件的指令。以下是一些基本的使用方式：
 1. `git mv <file> <new-file>`：將文件 `<file>` 重命名為 `<new-file>`
 2. `git mv <file> <directory>`：將文件 `<file>` 移動到 `<directory>` 目錄下
- 這個指令不僅會在工作區中移動或重命名文件，還會將這個操作添加到暫存區，準備提交。

git mv 與 mv 的差異

git mv 和 mv 都可以用來移動或重命名文件，但它們的主要差異在於 Git 的追蹤。

- mv 是一個基本的 Unix 指令，用於移動或重命名文件或目錄。當你使用 mv 重命名或移動文件時，Git 會認為原文件已被刪除，並且有一個新文件被創建。
- git mv 不僅會在工作區中移動或重命名文件，還會將這個操作添加到暫存區，準備提交。這意味著 Git 會追蹤這個文件的變更歷史。

Git Rm

`git rm` 命令用於從 工作樹 和 索引 中移除檔案。

```
git rm <file>
```

git rm 與 rm 的差異

git rm 和 rm 都可以用來刪除文件，但它們的主要差異在於 Git 的追蹤。

- rm 是一個基本的 Unix 指令，用於刪除文件或目錄。當你使用 rm 刪除文件時，Git 不會自動知道這個文件已經被刪除，你需要使用 git add 命令將這個變更添加到暫存區。
- git rm 不僅會在工作區中刪除文件，還會將這個操作添加到暫存區，準備提交。這意味著 Git 會追蹤這個文件的刪除歷史。這樣，當你下次提交時，Git 會知道這個文件已經被刪除。這對於保持你的版本控制歷史的完整性非常有用。

git rm --cached

- 功能

- 用於從 Git 的 索引 中移除檔案，但卻不刪除工作目錄中的檔案。

- 使用範例

- 若你不小心將一些大型的 log 檔案或者包含敏感資料的檔案加入了儲存庫中的 _sites 資料夾，可以使用這個命令來移除。

How To Clear Git Cache

```
:> git rm -r --cached _sites
```

removes the
specified folder
from the repository
and the working
directory

enables
recursive
deletion
of contents
within the
specified
folder

enables deletion
only on the
repository and
keeps the contents
of the working
directory intact

name of
the folder
to be
deleted



JunosNotes.com

熱門的 Git GUI

1. [GitHub Desktop](#): 提供了一個直觀的界面，不需要手動輸入命令。
2. [GitKraken](#): 提供美觀的視覺效果、直觀的使用者介面以及內建的整合工具。
3. [Sourcetree](#): 一個免費的 Git GUI 客戶端，提供了視覺化的介面來管理 Git 儲存庫。
4. [TortoiseGit](#): 一個 Windows 下的 Git 版本控制客戶端，集成在資源管理器中，提供了圖形化界面。
5. [SmartGit](#): 一個強大的 Git 客戶端，有豐富的功能，包括支援 GitHub, Bitbucket 和 GitLab。

熱門的 Git TUI

1. [gitui](#): 提供了一個快速的終端機使用者介面，讓你可以在終端機中享受 Git GUI 的舒適感。
2. [tig](#): 一個簡單的命令行介面，用於瀏覽和搜尋 Git 存儲庫的歷史。
3. [lazygit](#): 一個簡單的終端機 UI，用於 git 命令，讓你可以更輕鬆地管理 git 存儲庫。
4. [git-tui](#): 提供了一個人性化的終端機介面，讓你可以更方便地使用 git。

參考資料

- [圖解 Git](#)
- [30 天內學會 Git](#)
- [在 Git 遊樂園裡學習分支](#)
- [Git 快速小抄](#)
- [Git 分區指令大全](#)
- [Pro Git](#)

