[ Git種類 2](#_Toc478033753)

[ GitHub 2](#_Toc478033754)

[ 安裝Git for Windows 注意事項 3](#_Toc478033755)

[ Git敘述 4](#_Toc478033756)

[ 使用數據庫管理歷史記錄 4](#_Toc478033757)

[ 遠端數據庫和本地端數據庫 4](#_Toc478033758)

[ 建立本地端數據庫的方法有兩種： 4](#_Toc478033759)

[一種是重新建數據庫。另一種是複製遠端數據庫。 4](#_Toc478033760)

[ 記錄修改的提交 4](#_Toc478033761)

[ 工作目錄與索引 4](#_Toc478033762)

[ Tips（小撇步 ） 5](#_Toc478033763)

[ 指令 6](#_Toc478033764)

[ 開發分版觀念：Master、Branch和Merge 6](#_Toc478033765)

[ 版本線圖（commit graph） 6](#_Toc478033766)

[ 專案分版手法：Clone和Fork 6](#_Toc478033767)

[ Pull & Push 6](#_Toc478033768)

[ Pull Request 6](#_Toc478033769)

* Git種類

msysGit

Git for Windows

Github (https://desktop.github.com/)

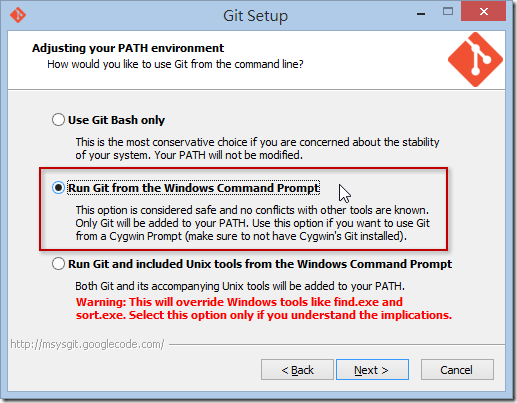
* GitHub

是以Git為核心的線上協作平臺

使用議題（Issue）、新開分支（Branch）、維基（Wiki）、拉取要求（Pull Request）等功能

* 安裝Git for Windows 注意事項

安裝 [Git for Windows](http://git-scm.com/download/win" \o "http://git-scm.com/download/win" \t "_blank)時，安裝到 **Adjusting your PATH environment** 步驟時，請選擇 **Run Git from the Windows Command Prompt** 這個選項，相容性比較高，問題也會少很多：



* Git敘述

Git為分散式版本控制系統，是為了更好管理Linux內核而開發的。

Git可以把檔案的狀態作為更新歷史記錄保存起來。因此可以把編輯過的檔案復原到以前的狀態，也可以顯示編輯過內容的差異。

而且，當有人想將編輯過的舊檔案上傳到伺服器、覆蓋其他人的最新檔案時，系統會發出警告，因此可以避免在無意中覆蓋他人的編輯內容。

在Git管理檔案的話，更新歷史會保存在Git。所以不需複製備用的檔案

* 使用數據庫管理歷史記錄

數據庫 (Repository) 是記錄檔案或目錄狀態的地方，儲存內容的修改歷史記錄。在數據庫的管理下除了儲存修改歷史記錄外，還可以追蹤內容的狀態和版本喔。

* 遠端數據庫和本地端數據庫

首先，Git的數據庫分為遠端數據庫和本地端數據庫。

• 遠端數據庫: 配有專用的伺服器，為了讓多人共享而建立的數據庫。

• 本地端數據庫：為了方便用戶個人使用，在自己的機器上配置的數據庫。

您可以在本地端數據庫上使用像是還原更改、跟踪更改等所有Git版本控制功能。

不過，如果想要公開在本地端數據庫的修改內容，就需要將內容上傳到遠端數據庫了。另外，透過遠端數據庫還可以取得其他人修改的內容。

* 建立本地端數據庫的方法有兩種：

重新建數據庫。

複製遠端數據庫。

* 記錄修改的提交

若想把變更與新增的檔案/目錄儲存到數據庫中，您需要執行提交（Commit）。

執行提交後，數據庫裡會產生上次提交的狀態與現在狀態的差異記錄（也被稱為Revision）。

如下圖，提交是以時間順序排列狀態被儲存到數據庫中的。憑藉該提交和最新的檔案狀態，就可以知道過去的修改記錄以及內容。

為了區分提交，系統會產生一組識別碼來給提交命名，識別碼會根據提交修改的內容計算出不重複的40位英文數字。指定識別碼，就可以在數據庫中找到對應的提交。

* 工作目錄與索引

工作目錄（Working Tree）是保存您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。

索引(Index )位於工作目錄和數據庫之間，是為了向數據庫提交作準備的暫存區域。

所以在工作目錄上做的任何變更並不會直接提交到數據庫的。Git在執行提交的時候，不是直接將工作目錄的狀態儲存到數據庫，而是將索引的狀態儲存到數據庫。因此，要提交變更，首先必需要把變更內容加入到索引中。

索引的存在可以排除工作目錄裡不必要的檔案提交，還可以只將檔案變更內容的一部分加入索引並提交。

* 本地端版本控制



* 集中式版本控制系統



* 分散式版本控制系統



* Tips（小撇步 ）

像錯誤修復或功能添加之類不同含義的更改，要盡量分開來提交。這樣可以方便事後從歷史記錄裡找出特定的修改內容。

執行提交時，系統會要求輸入提交訊息。請務必輸入提交訊息，因為在訊息空白的狀態下執行提交是會失敗的。

提交訊息是查看其他人提交的修改內容或自己檢查歷史記錄時重要的資料。所以要用心填寫讓人容易理解的提交訊息。

Git的標準提交訊息：

第1行：提交時修改內容的摘要

第2行：空行

第3行以後：修改的理由

 SSH副檔名有.pub的表示是公鑰，沒有副檔名的是私鑰

* 指令
  + 開發分版觀念：Master、Branch和Merge

主幹（Master）與分支（Branch）是稱呼專案的主要版本和分支版本。

在GitHub上，通常第一個建立的專案版本會稱為Master版本。

但對Git而言，開發專案的版本並無主幹（Master）與分支（Branch）的分別，每個分支都是等價關係，而開發者自行調整版本命名來區分。

一般習慣將穩定版本稱主幹，其餘的變動、開發中版本則都稱作分支。

Branch也是一個建立分支的Git指令，可將某個歷史版本複製一份，獨立成為另一個新的分支版本，而合併（Merge）指令剛好相反，則是把兩個不同的分支版本，合併到其中一個分支上。

* + 版本線圖（commit graph）

Git則提供了一個視覺化的版本線圖（commit graph），來呈現出主幹與各分支連結形成的樹狀架構。不過隨著專案越來越大，版本線圖會變得越來越複雜。

* + 專案分版手法：Clone和Fork

複製（Clone）指令是把專案在遠端儲存庫上的所有內容複製到本地，建立起本機儲存庫及工作目錄，而叉（Fork）則是把別人專案的遠端儲存庫內容複製一份到自己的遠端儲存庫

* + Pull & Push

可以執行Pull取得其他人開放授權的遠端儲存庫上的程式碼，也可以將自己修改的程式碼Push到可存取的第三者遠端儲存庫上。透過推（Push）跟拉（Pull）兩個指令，開發者就能互相分享原始碼。

* + Pull Request

Pull Request則是更主動地要求第三方開發者納入自己開發的程式，將本地端儲存庫上的程式碼，整併到對方的儲存庫上。例如A成員可透過Pull Request要求B成員，將A成員修改後的程式碼併入B成員的開發專案中。(1060323)

$ git init

建立本地儲存區 (local repository) 初始化 Git 目錄

$ git status

取得目前 Git 工作目錄的狀態

$ git add frlename

新增檔案(將尚未被Git追蹤的新增檔案加進去追蹤名單)

$ git add .

加上一個小數點.表目前目錄，將「新增」與「變更」檔案 (Untracked files) 加入追蹤清單中

$git add . –u

把變更過的檔案加入追蹤。不加上 -u 則代表「新增」與「變更」都會一併提交追蹤。

$git config --global --list

使用者層級 (User-level configuration) (設定於目前登入的使用者)

$git commit

提交變更(將Git狀態中所有異動的部分送進儲存區Repository )

$git commit -m "<版本紀錄訊息>"

提交變更，附上紀錄

$git log

查看歷史紀錄

$ git log –stat

$ git log --pretty=oneline

commit 後面接著一串亂碼 ( SHA1 Hash ) 代表這個版本的唯一記號

Author 代表提交這次變更的作者，這裡的資料是由提交的人當下設定的 user.name 與 user.email 參數，所以如果有人有改名過，從這裡可能不好看出到底是誰改的，或覺得紀錄有點亂，這可能一開始大家要先講好比較不會亂。

Date 代表這個版本被提交的日期

最後，就是當時留下的紀錄訊息。

**$ git rm '*<filename>*'**

**刪除檔案**

**$ git reset HEAD [<path\filename>]**

**重置檔案狀態 (不會變更檔案內容)**

**$ git checkout --**

**復原所有檔案變更 (Undo Changes)**

**$ git checkout -- <filename.ext>**

**復原檔案變更 (Undo Changes)**

**--** 等同於 **HEAD** 的意思，是 **HEAD** 的簡寫

**$ git clean**

**清除未追蹤的檔案 ( Clean untracked files )**

**$ git clean –n**

**(僅列出要清空的檔案清單)**

**$ git clean -f**

**(執行刪除動作，把那些要清空的檔案逐一刪除)**

**$ git clone https://github.com/Lolikitty/Example.git(GITHUB URL)**

**複製到本機資料夾**

* **For example**

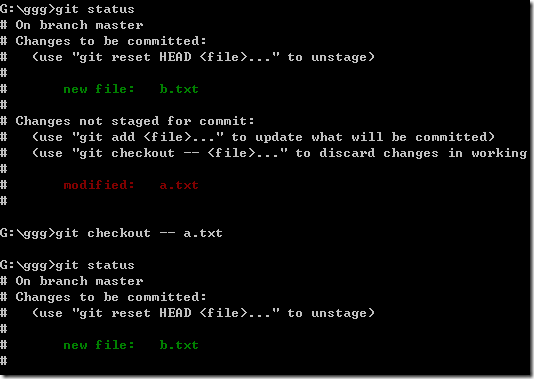
**假設你是一個舞團指揮，目前正在編舞 (程式開發中)，排練時有個舞台可以練習，而這個舞台就是你的 Git 工作區 (目前這個目錄)，該舞團有個工具可以控制舞台上所有事情 (就是你的 Git 工具)，而舞台上所有 舞者 (目錄下的所有檔案) 都會聽指揮的命令。**

**今天，舞台上有一個 舞者 (a.txt) 自己發明了一個新舞步 (檔案被更新)，但 指揮 並不知道，當 指揮 詢問這位舞者的狀態 ( git status ) 時，發現了這個舞者 (a.txt) 發明了新舞步，他必須決定是否要把舞步編進舞冊上 ( git commit )。**

**此時，假設指揮同意了這個舞步變更，他就必須把這個檔案加到編舞裡去，你就要輸入 git add -u 命令讓他上台試跳，如果跳的不錯，這時輸入 git commit 才會真正把舞步編進舞冊上。**

**設定 Git 執行環境的三個層級** ( **註**: 執行 **git config --help** 可以取得完整指令說明 )

* **系統層級** (System-level configuration) (設定於整台電腦，所有使用者的預設值)
  + git config --system --list
  + 預設路徑: **C:\Program Files (x86)\Git\etc\gitconfig**
  + 或是路徑: **%LOCALAPPDATA%\VirtualStore\Program Files (x86)\Git\etc\gitconfig**   
    **補充參考**: [VirtualStore - Inside Windows Vista User Account Control](http://technet.microsoft.com/zh-tw/magazine/2007.06.uac(en-us).aspx" \t "_blank)
* **使用者層級** (User-level configuration) (設定於目前登入的使用者)
  + git config --global --list
  + 預設路徑: C:\Users\<使用者帳號>\.gitconfig      (這是個檔案)
  + 常用指令
    - 最常見的設定就是 user 區段下的 name 與 email 參數，第一次用 Git 一定要設定!
      * git config --global user.name "John Doe"
      * git config --global user.email "johndoe@example.com"
    - 在 Linux 底下可以指定編輯器
      * git config --global core.editor vim
    - 在Windows底下，最好打開 core.autocrlf 選項，讓 commit 的檔案沒有 CR 字元   
      ( 純 Windows 開發環境，可以設定為 false；跨平台協同開發，建議設定為 true )
      * git config --global core.autocrlf true
    - 自動訂正打錯的參數，例如 git statsu 會自動修正為 git status 讓你不用重打一次
      * git config --global help.autocorrect 1
* **儲存區層級** (Repository-level configuration)
  + git config --list
  + 預設路徑: <Git儲存區>\.git\config



filter.lfs.clean=git-lfs clean -- %f

filter.lfs.smudge=git-lfs smudge -- %f

filter.lfs.process=git-lfs filter-process

filter.lfs.required=true

user.name=papa3321

user.email=richard.hsieh@askeycloud.com

core.autocrlf=true

core.excludesfile=C:\Users\richard1\_hsieh\Documents\gitignore\_global.txt

* Git local建置步驟

1. $ git init

=>Initialized empty Git repository in /.git/

=>Success!

1. $ git status

=># On branch master

=># Initial commit

=>nothing to commit (create/copy files and use "git add" to track)

=>Success!

1. $ git status

=># On branch master

=># Initial commit

=># Untracked files:

=># (use "git add <file>..." to include in what will be committed)

=># octocat.txt

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

=>Success!

1. $ git add octocat.txt

=>Nice job, you've added octocat.txt to the Staging Area

=>#Success!

1. $ git status

=># On branch master

=># Initial commit

=># Changes to be committed:

=># (use "git rm --cached <file>..." to unstage)

=># new file: octocat.txt

=>#Success!

1. $ git commit -m "Add cute octocat story"

=>[master (root-commit) 20b5ccd] Add cute octocat story

=> 1 file changed, 1 insertion(+)

=> create mode 100644 octocat.txt

=>#Success!

1. $ git add '\*.txt'

=>#Success!

1. $ git commit -m 'Add all the octocat txt files'

=>[master 3852b4d] Add all the octocat txt files

=> 4 files changed, 4 insertions(+)

=> create mode 100644 blue\_octocat.txt

=> create mode 100644 octofamily/baby\_octocat.txt

=> create mode 100644 octofamily/momma\_octocat.txt

=> create mode 100644 red\_octocat.txt

=>Success!

1. $ git log

=>commit 3852b4db1634463d0bb4d267edb7b3f9cd02ace1

=>Author: Try Git <try\_git@github.com>

=>Date: Sat Oct 10 08:30:00 2020 -0500

=> Add all the octocat txt files

=>commit b652edfd888cd3d5e7fcb857d0dabc5a0fcb5e28

=>Author: Try Git <try\_git@github.com>

=>Date: Sat Oct 10 08:30:00 2020 -0500

=> Added cute octocat story

=>Success!

1. $ git remote add origin https://github.com/try-git/try\_git.git

=>Success!

1. $ git push -u origin master

=>Branch master set up to track remote branch master from origin.

=>Success!

1. $ git pull origin master

=>Updating 3852b4d..3e70b0f

=>Fast-forward

=> yellow\_octocat.txt | 1 +

=> 1 file changed, 1 insertion(+)

=> create mode 100644 yellow\_octocat.txt

=>Success!

1. $ git diff HEAD

=>diff --git a/octocat.txt b/octocat.txt

=>index 7d8d808..e725ef6 100644

=>--- a/octocat.txt

=>+++ b/octocat.txt

=>@@ -1 +1 @@

=>-A Tale of Two Octocats

=>+[mA Tale of Two Octocats and an Octodog

=>Success!

1. $ git add octofamily/octodog.txt

=>Success!

1. $ git diff --staged

=>diff --git a/octofamily/octodog.txt b/octofamily/octodog.txt

=>new file mode 100644

=>index 0000000..cfbc74a

=>--- /dev/null

=>+++ b/octofamily/octodog.txt

=>@@ -0,0 +1 @@

=>+[mwoof

=>Success!

1. $ git reset octofamily/octodog.txt

=>Success!

1. $ git checkout -- octocat.txt

=>Success!

1. $ git branch clean\_up

=>Success!

1. $ git checkout clean\_up

=>Switched to branch 'clean\_up

=>Success!

1. $ git rm '\*.txt'

=>rm 'blue\_octocat.txt'

=>rm 'octocat.txt'

=>rm 'octofamily/baby\_octocat.txt'

=>rm 'octofamily/momma\_octocat.txt'

=>rm 'red\_octocat.txt'

=>Success!

1. $ git commit -m "Remove all the cats"

=>[clean\_up 63540fe] Remove all the cats

=> 5 files changed, 5 deletions(-)

=> delete mode 100644 blue\_octocat.txt

=> delete mode 100644 octocat.txt

=> delete mode 100644 octofamily/baby\_octocat.txt

=> delete mode 100644 octofamily/momma\_octocat.txt

=> delete mode 100644 red\_octocat.txt

=>Success!

1. $ git checkout master

=>Switched to branch 'master'

=>Success!

1. $ git merge clean\_up

=>Updating 3852b4d..ec6888b

=>Fast-forward

=> blue\_octocat.txt | 1 -

=> octocat.txt | 1 -

=> octofamily/baby\_octocat.txt | 1 -

=> octofamily/momma\_octocat.txt | 1 -

=> red\_octocat.txt | 1 -

=> 5 files changed, 5 deletions(-)

=> delete mode 100644 blue\_octocat.txt

=> delete mode 100644 octocat.txt

=> delete mode 100644 octofamily/baby\_octocat.txt

=> delete mode 100644 octofamily/momma\_octocat.txt

=> delete mode 100644 red\_octocat.txt

=>Success!

1. $ git branch -d clean\_up

Deleted branch clean\_up (was ec6888b).

=>Success!

1. $ git push

=>To https://github.com/try-git/try\_git.git

=>3e70b0f..95b8bbc master -> master

=>Success!

git rm -r folder\_of\_cats

* AWS CodeCommit

AWS CodeCommit 提供多種其他 Git 原始程式碼控制系統沒有提供的功能：

* + 全受管 – AWS CodeCommit 讓您無須託管、維護、備份和擴展自己的原始程式碼控制伺服器。
  + 安全 – AWS CodeCommit 會自動加密傳輸中和靜態的檔案。AWS CodeCommit 已經與 AWS Identity and Access Management (IAM) 整合，讓您能夠指派使用者特定權限給儲存庫。
  + 高可用性 – AWS CodeCommit 建置在擴展性高、冗餘且耐用的 AWS 服務上，例如 Amazon S3 和 Amazon DynamoDB。
  + 可擴展性 – AWS CodeCommit 可讓您存放任意數量的檔案，而且儲存庫沒有大小限制。
  + 更快速的開發生命週期 – AWS CodeCommit 讓儲存庫更接近 AWS 雲端中的建置、分段和生產環境。這能讓您增加開發生命週期的速度和頻率。
* 架設

Step 1: Create an AWS CodeCommit Repository

Step 2: Create a Local Repo

Step 3: Create Your First Commit

Step 4: Push Your First Commit

Step 5: Share the AWS CodeCommit Repository and Push and Pull Another Commit

Step 6: Create and Share a Branch

Step 7: Create and Share a Tag

Step 8: Set Up Access Permissions

Step 9: Clean Up

* 環境設定

$ git config --global user.email "you@example.com"

$ git config --global user.name "Your Name"

$ git config --global color.ui true

$ git config --global core.editor vim

$ git config --global alias.co commit

$ git config --global alias.lg "log --color --graph --all --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --"