Git Basic

Git 基礎篇

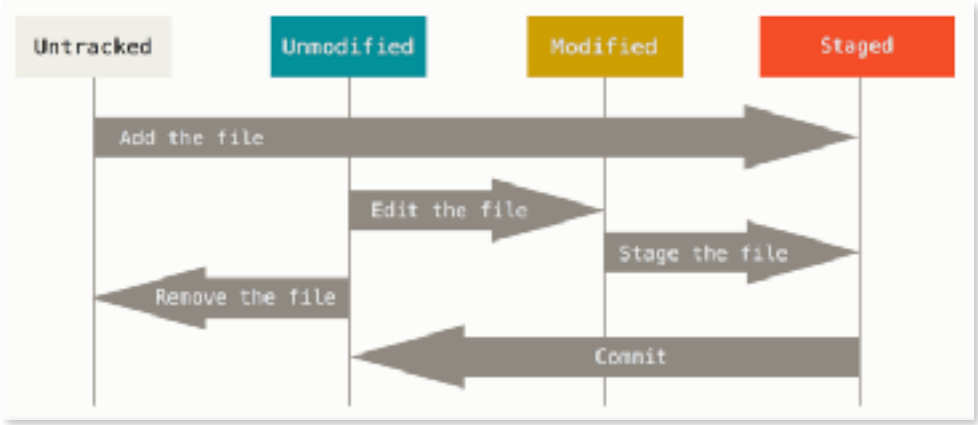
Jean Hu - 11 September 2017

不同類型的 VCS

集中式版本控制系統 Centralized Version Control Systems, CVCS	分散式版本控制系統 Distributed Version Control Systems, DVSC
採主從式架構， Repository 集中放置於中央伺服器	每個用戶端都擁有完整的 Repository
CVS、SVN、Perforce	Git、Mercurial、Bazaar
• 管理員以目錄結構掌控每個開發者的權限	• 若有發生伺服器故障，可以用任何一個用戶端的鏡像來還原。 因為每個地方都有完整的資料備份。 • 開發者不須透過網路，便能提交程式碼到本機端 Repository ，等到需要與遠端 Repositroy 同步時，才需要使用網路，降低對網路的依賴程度。
• 資料皆在單一伺服器上，如果想要提交新版本、查詢各版本差異，都要連到伺服器才能進行。 • 單點故障，如果中心檔案庫的硬碟發生損壞，又沒有做適當的備份，那就會遺失所有資料。	• 專案較大的情況下拉取較慢

除此之外，Git 還有幾項特點：

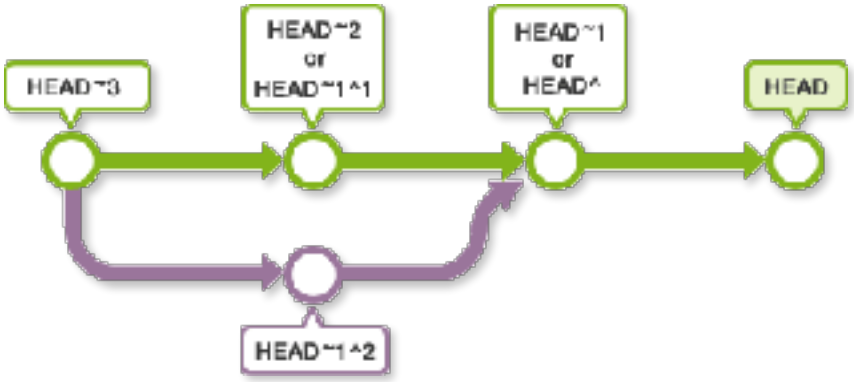
- 紀錄檔案快照，而不是差異。
- 大部分操作皆可以在本地端完成。
- 檢查完整性，Git 用來計算校驗碼的機制稱為 SHA-1 雜湊演算法。
- 在很多 VCS 中使用分支是個昂貴的過程，需要對原始程式碼目錄建立完整的副本。Git 分支非常羽量級，新建與切換分支的操作非常快。分支實際上只是一個檔案，該檔案內容是這個分支指向的提交的雜湊值。
- 鼓勵在工作流程中頻繁地使用分支與合併（merge）。
- 工作區域
  - Git 倉庫：用來保存項目的元數據和對象資料庫的地方。
  - Working directory（工作目錄）：對某版本提取出來的內容，即工作分支。
  - Staging area（預存區）：也稱為 index（索引），保存下次要提交的資訊。



- 檔案狀態
  - Ignore（被排除的檔案）：被排除，不需要加入 Git 的檔案。
  - Untracked（未追蹤檔案）：尚未添加至版本控制中的檔案。
  - Modified（已修改）：在現有版本上已進行過修改的檔案但尚未提交。
  - Staged（已預存）：檔案已添加至預存區，將被存到下次提交的快照中。
  - Committed（已提交）/ Unmodified：檔案提交至 repository，尚未修改。

- 基本 Git 工作流程大致為
  1. 建立或 Clone 一個 Git repository，並在其基礎上建立工作目錄。
  2. 在工作目錄中添加或修改檔案。
  3. 預存檔案，將檔案的快照新增到預存區。
  4. 提交更新，讓存在預存區的檔案快照永久地儲存在 Git Repository。
  5. Push 至遠端 Git repository。

Commit 節點



- Git 特殊指標 HEAD
  - 永遠代表當前分支的最新的 commit ，也可以縮寫為@。

- 找尋更早的節點
  - 當節點中有一個以上的父節點時，可用來指定為哪個父節點。

    `<節點標籤或是識別碼>^<數字>`

  - 節點標籤或是識別碼~數字，用來表示那一層的父節點。

    `<節點標籤或是識別碼>~<數字>`

- Commit 節點識別碼
  - 為一串很長16進位的數字在指定 commit 節點時不需要完整列出，一般只需要最前面4碼數字即可如有重複會發生錯誤，再加長一點的數字即可解決。

Git 安裝教學

- 在 Linux 中安裝

  可直接透過所用的 distribution 內建的基礎套件管理工具。

  - Fedora

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

  - Debian

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

- 在 Mac 中安裝

  使用 Homebrew 安裝

  ```
  $ brew install git
  ```

- 在 Windows 中安裝

  在 Git 官方網站下載：https://git-scm.com/downloads

初次設定 Git

- Git 設定檔
  - Git 有三個不同層級的設定檔，優先權從低至高如下：
    - Git 程式安裝資料夾中的 /etc/gitconfig：包含該系統所有使用者和 repository 的設定。傳遞 --system 參數，Git 就會從這個檔案讀取或寫入設定。
    - 登入帳號家目錄下的 .gitconfig 檔（~/.gitconfig、~/.config/git/config）：帳號專用的設定。 傳遞 --global 參數， Git 就會從這個檔案讀取或寫入設定。
    - Repository 中 Git 資料夾 config 檔案（.git/config）：該 repository 專用設定。傳遞 --local 參數， Git 就會從這個檔案讀取或寫入設定。

  - 設定識別資料，安裝 Git 後首先應該做的事是設定使用者名稱及電子郵件。

    ```
    $ git config global user.name JeanHu
    $ git config --global user.email jean.hu@tpinformation.com.tw
    ```

  - 檢閱設定

    ```
    $ git config -l
    ```

- 說明文件

  ```
  $ git help <verb>
  $ git <verb> –help
  $ man git <verb>
  ```

Git 基礎應用

- 建立 Git repository
  - 在現有的資料夾中創建 repository

    ```
    $ git init
    ```

    - 這個命令將會建立一個 .git 子資料夾，其中包含 Git 所有必需的倉儲檔案。
    - 若專案資料夾原本已經有檔案，可以追蹤這些原本就有的檔案，且進行提交。

    ```
    $ git add .
    $ git commit -m 'initial project'
    ```

  - Clone 現有的 repository

    ```
    $ git clone <遠端檔案庫> <本機檔案庫>
    ```

    - 工作目錄下的每個檔案不外乎兩種狀態，
      - 已追蹤：檔案狀態可能是未修改、已修改、已預存（staged）。
      - 未追蹤：在工作目錄中，不包含在上次快照中及預存區中的任何檔案。
    - 第一次 Clone 一個檔案庫時，所有檔案都是「已追蹤」且「未修改」。

- 忽略不需要加入 Git 的檔案
  - 新建一個名為 .gitignore 的檔案，在該檔中列舉符合這些檔名的 pattern。
  - 編寫 .gitignore 檔案的模式規則如下：
    - 空白列，或者以 # 開頭的列會被忽略。
    - 可使用標準的 Glob 模式。
  - 可以參考 GitHub 提供的 Template，或是 gitignore.io，產生符合得配置。

- 檢查檔案狀態
  - 檢視檔案狀態

    ```
    $ git status
    ```

    - 區塊顯示各種檔案狀態，並提示相對應下一步指令。

  - 精簡模式檢視

    ```
    $ git status -s
    $ git status --short
    ```

    - 標記有二個欄位，左邊欄位表示預存區狀態，右邊則是工作目錄狀態。
    - 未追蹤的新檔案在開頭被標示為 ??
    - 被加入預存區的新檔案被標為 A
    - 已修改檔案則是 M

- 加入預存區
  - 追蹤新的檔案、預存修改過後的檔案（或像是將「標記合併衝突（merge-conflicted）的檔案設為已解決」），皆使用 git add 命令。
  - 把它想成「把檔案內容加入下一個提交中」會比較容易理解。
  - 常用指令
    - 使用 Interactive mode
      ```
      $ git add -i
      ```
    - 將新增、修改的檔案加入索引
      ```
      $ git add .
      ```
    - 將修改、刪除的檔案加入索引
      ```
      $ git add -u
      ```
    - 將新增、修改、刪除的檔案加入索引
      ```
      $ git add -A
      ```

- 比較檔案差異
  - 先看索引，再看檔案庫
    ```
    $ git diff <檔案名稱>
    ```
  - 比對資料夾中兩個檔案
    ```
    $ git diff –no-index <檔案1> <檔案2>
    ```
  - 比對資料夾與檔案庫
    ```
    $ git diff <commit 節點> <檔案名稱>
    ```
  - 先看索引，再看檔案庫
    ```
    $ git diff --cached <檔案名稱>
    $ git diff --staged <檔案名稱>
    ```
  - 比對檔案庫中的兩個commit
    ```
    $ git diff <commit 節點1> <commit 節點2> <檔案名稱>
    ```

- 提交修改至檔案庫
  - 啟動選定的編輯器編輯 comment，Git 會利用這些提交訊息（註解和差異內容會被濾除）產生新的提交。
    ```
    $ git commit
    ```
  - 直接輸入 comment
    ```
    $ git commit -m <comment>
    ```
  - 顯示修改的差異內容
    ```
    $ git commit -v
    ```

- 略過預存區
  ```
  $ git commit -a
  ```
- 修改最新的 commit
  ```
  $ git commit --amend
  ```

- 移除檔案
  - 將檔案狀態從tracked 改為 untracked，索引中的檔案內容會被移除
    ```
    $ git rm --cached <檔案名稱>
    ```
  - 刪除檔案庫中的檔案，或是索引中的檔案，也可刪除資料夾中的檔案
    ```
    $ git rm <檔案名稱>
    ```
    - 檢查索引中有無該檔案及檢查檔案內容是否與資料庫中的一致，若皆通過，會刪除該檔案，並在索引中紀錄要從檔案庫刪除檔案。
  - 或者手動刪除不需要的檔案後，執行 `$ git add -A .`，將刪除的檔案記錄在索引當中。

- 移動檔案
  - 重新命名檔案
    ```
    $ git mv <原來檔案名稱> <新檔案名稱>
    ```
    - 相當於執行下列命令：
      ```
      $ mv <原來檔案名稱> <新檔案名稱>
      $ git rm <原來檔案名稱>
      $ git add <新檔案名稱>
      ```

- 復原
  - 復原被修改且尚未加入預存區的檔案
    ```
    $ git checkout -- <檔案名稱>
    ```
  - 將已經預存的檔案移出預存區
    ```
    $ git reset HEAD <檔案名稱>
    ```
  - 前面有提過，修改最新的 commit
    ```
    $ git commit --amend
    ```
    - 提交後才意識到想把某些忘記預存的修改也一併加入到上一個提交中，可執行：
      ```
      $ git commit -m <comment>
      $ git add <忘記的檔案>
      $ git commit –amend
      ```
  - 回復到某個 commit 節點
    ```
    $ git reset <選項> <commit 節點>
    ```

- --soft：只有檔案庫裡頭的資料會變更。
- --mixed：預設選項，索引也會回到指定節點的狀態，但資料夾中檔案不受影響。
- --hard：檔案庫、索引、資料夾中檔案，都會回復成指定節點的狀態。

- 撤銷某次提交

```
$ git revert <commit 節點>
```

- 不會刪除 Git 檔案庫中的 commit節點。
- 回到指定 commit節點的前一個節點。

- 暫存目前資料夾的檔案狀態
  - 暫存資料夾中檔案狀態

```
$ git stash save
```

- 儲存資料夾中被追蹤的檔案和檔案庫中最新版本的差異。
- 把資料夾中被追蹤的檔案還原至檔案庫中的最新的版本。

  - 顯示資訊及執行 save 時比對的基礎檔案版本

```
$ git stash list
```

  - 回復成 $ git stash save 時的狀態
    - 取出 $ git stash list 時比對的檔案版本

```
$ git checkout <commit 節點>
```

    - 取出暫存檔案，並且合併到目前資料夾中的檔案

```
$ git stash pop
$ git stash apply
```

- 標籤
  - 列出所有標籤

```
$ git tag
```

    - -l <pattern>：使用特定的 pattern 來搜尋標籤。

  - 查詢標籤

```
$ git show <標籤名稱>
```

  - 建立輕量級的標籤

```
$ git tag <標籤名稱>
```

    - 這種標籤只會指向一個特定的提交

  - 建立有註解的標籤

```
$ git tag -a <標籤名稱> <commit 節點>
```

- 有註解的標籤，會儲存成完整的物件。
- 計算校驗碼，包含貼標籤那個人的名字、電子郵件和日期。
- 能夠紀錄一個標籤訊息。
- 並且可以簽署及透過 GNU Privacy Guard (GPG) 驗證。
- 通常建議建立一個有註解的標籤，以便保留跟這個標籤有關的所有資訊。

- 刪除標籤

```
$ git tag -d <標籤名稱>
```

- 分享指定標籤

```
$ git push <標籤名稱>
```

- 將所有不在伺服器上面的標籤傳送給遠端伺服器

```
$ git push --tag
```

查詢記錄

- 檢視提交歷史紀錄
  - `$ git log`
    - -p：顯示每筆提交所做的修改內容。
    - -<筆數>：限制輸出最後幾筆內容。
    - --stat：檢視每筆提交簡略的統計資訊。
    - --oneline 選項將每一筆提交顯示成單獨一行。
    - --graph：畫出 commit 節點演進圖。
    - --decorate：標示出分支的名稱。
    - --all：顯示所有分支。
    - --pretty：用來改變原本預設輸出的格式。
      - short、full、fuller： 輸出的格式大致相同，分別會少一些或者多一些資訊。
      - format：可以指定自訂的輸出格式。
    - 限制日誌輸出
      - --since 和 --until：限制時間的選項。這個命令支援各種格式，可以指定特定的日期格式（例如："2008-01-15"），或者相對日期格式。
      - --after / --before：限制時間區間。
      - -S：用來尋找所修改的內容中被加入或移除某字串的提交。
  - `$ git shortlog`
    - 列出每人執行 commit 的次數及說明。
    - -n：commit 次數由高至低排序。
    - -s：不需要顯示 commit 說明。

- `$ git reflog HEAD or <分支名稱>`
  - 列出對 repository 執行的詳細指令對應。

- 查詢特定字串
  - 搜尋指定 commit 中的所有檔案

```
$ git grep <找尋的字串> <commit 節點>
```

    - 若沒有指定commit 節點，則搜尋資料夾中所有檔案。
    - -i：不分大小寫。
    - -c：列出檔案中出現次數。
    - -l：僅列出檔案名稱。

  - 若要找字串不只一個，用 -e 分別指定字串

```
$ git grep -e '找尋的字串1' -e –and '找尋的字串2' …
```

    - 預設是以 or 方式結合，若想使用 and 方式需加上 --and。

- 查詢修改資訊

```
$ git blame <檔案名稱>
```

  - 若想知道每一行最後是由誰修改的

```
$ git blame -L 起始行,結束行 檔案名稱
$ git blame -L 起始行 檔案名稱
$ git blame -L ,結束行 檔案名稱
```

與遠端協同工作

- 管理遠端檔案庫
  - 顯示遠端檔案庫

```
$ git remote -v
$ git remote show <遠端簡稱>
```

    - 若使用 $ git clone 取得遠端檔案庫，會建立簡稱為 origin 的遠端檔案庫對應。
    - 若指令不輸入遠端簡稱，預設為 origin。

  - 列出本地檔案庫對應的所有遠端檔案庫

```
$ git ls-remote
```

  - 新增遠端檔案庫

```
$ git remote add <簡稱> <url>
```

  - 移除或重新命名遠端

```
$ git remote rename <舊名稱> <新名稱>
$ git remote rm <簡稱>
```

- 更新遠端連線資訊
  ```
  $ git remote set-url <遠端簡稱> <新URL>
  ```

- 同步遠端檔案庫至本地檔案庫
  - 從遠端獲取
    ```
    $ git fetch <遠端簡稱>
    ```

    - 連結到遠端專案，將本地還沒有的資料全部拉下來。
    - 執行完成後會有那個遠端檔案庫中所有分支的 reference ，可用來合併或檢視。

  - 從遠端拉取
    ```
    $ git pull <遠端簡稱>
    ```

    - 自動「獲取」並「合併」那個遠端分支到目前的分支，相當於執行下列命令：
      ```
      $ git fetch
      $ git merge
      ```

    - 使用 rebase 的方式合併
      ```
      $ git pull --rebase
      $ git pull -r
      ```

- 推送至遠端
  ```
  $ git push <遠端簡稱> <分支名稱>
  ```

  - 推送分支至遠端儲存庫並設定追蹤該上游分支
    ```
    $ git push –set-upstream <遠端簡稱> <分支名稱>
    $ git push -u <遠端簡稱> <分支名稱>
    ```

    - 且將會在設定檔中紀錄本地分支與遠端分支的對應關係。

分支（Branches）

- 分支管理
  - 查詢分支
    ```
    $ git branch
    $ git branch --list 分支名稱
    ```

  - 查看各個分支最後一個提交
    ```
    $ git branch -v
    ```

    - --merged 和 --no-merged：從該清單中篩選出已經合併或尚未合併到目前分支的分支。

- 建立分支
  ```
  $ git branch <分支名稱> <commit 節點>
  ```
  - 若未指定 commit，則由最新的 commit 切出分支。
- 切換分支
  ```
  $ git checkout <分支名稱>
  ```
  - 切換分支時，工作目錄內的檔案將會被修改。
  - 若無法乾淨地切換過去，將無法切換。所以最好先保持乾淨的工作區域。
- 同時新建及切換分支
  ```
  $ git checkout -b <分支名稱>
  ```
- 刪除分支
  ```
  $ git branch -d <分支名稱>
  ```
- 變更分支名稱
  ```
  $ git branch -m <新分支名稱>
  ```

- 合併分支
  - 切回合併目的地的分支，然後執行：
    ```
    $ git merge <被合併分支>
    ```
    - --no-ff：強制使用 3-way merge。
  - 利用 Rebase 更新分支的起始點
    ```
    $ git rebase <被合併分支>
    ```
    - 若是開一個 feature 分支，將主分支合併回去保持一制性，比較推薦使用 rebase，避免兩分支相互交織的情況發生。
    - 但若有與他人共用此分支，就不適合執行 rebase 指令，因為會更動到舊的 commit 節點，會造成跟其他人檔案庫資料不一致的問題。
  - 執行 Rebase 想反悔
    ```
    $ git reflog HEAD 或是<任何分支的名稱>
    ```
    - 必須先找到執行 rebase 前，HEAD 所在的 commit節點。

      ```
      $ git reset --hard HEAD@{<前幾個commit>}
      ```
    - 回復到 rebase 之前的節點。
  - 把指定 commit 節點的檔案版本，合併到資料夾中的檔案
    ```
    $ git cherry-pick <commit 節點>
    ```

- 回復到未合併前的分支

```
$ git reset –hard HEAD^
$ git reset --merge ORIG_HEAD
```

- 從檔案庫中取出檔案
  - 取得某版本檔案

```
$ git checkout <commit 節點> <檔案1> <檔案2>...
```

    - 從檔案庫中指定 commit 節點取出檔案，若沒有包含指定檔案，自動往更舊commit 節點尋找，若皆無檔案則為錯誤。
    - 資料夾中的檔案會被取出的檔案覆蓋。
    - 若取出檔案與檔案庫中最新的 commit 內容不同，取出的檔案自動被記錄在索引。若又執行 git commit ，則取出的檔案內容就會存入檔案庫中成為最新的版本，可在執行checkout 後立即執行 git reset HEAD，清除索引。

  - Detached HEAD
    - 取得某版本檔案 `$ git checkout <commit 節點>`。變更資料夾內容，改變 HEAD 指向該 commit 節點，形成 detached HEAD。
    - 放棄更改內容。
      - 賦予無名分支一個名稱

```
$ git branch <無名分支名稱>
```

      - 切換到另外一個分支

```
$ git checkout <分支名稱>
```

      - 刪除無名分支

```
$ git branch -D <無名分支名稱>。
```

    - 將內容合併回原來的分支。
      - 賦予無名分支一個名稱

```
$ git branch <無名分支名稱>
```

      - 切換到另外一個分支

```
$ git checkout <分支名稱>
```

      - 合併無名分支

```
$ git merge <無名分支名稱>
```

其他提升工作效率指令

- 別名（Aliases）
  - 使用 git config 來替指令設定別名

```
$ git config alias.<指令別名> <正式指令和選項>
```

- 常用的設定如下：

```
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
$ git config --global alias.co checkout
$ git config --global alias.unstage 'reset HEAD --'
$ git config --global alias.last 'log -1 HEAD'
```

- 清理 Git 檔案庫

```
$ git gc
```

- --aggressive：會比較仔細去檢查且清理，需花較久時間。
- --auto：先判斷是否需要清理，若判斷良好則不清理。
- --no-prune：要求Git 不要清除檔案中不會用到的資料，只要整理它們即可。

參考資料

- Pro Git 2nd Edition
- Git 重點手札.pdf
- 完整學會Git GitHub Git Server的24堂課
- 連猴子都能懂的 Git 入門指南
- 學習分支的教學網站