第 01 天：認識 Git 版本控管

筆者使用 Subversion (SVN) 已經將近 10 年，從來都不覺得有任何必要轉換至其他版本控管平台，直到前幾年因應雲端化的改變，慢慢導入 TFS 版本控管 (TFS Service)，轉換的過程還算順利，只因為 SVN 與 TFS 的版本控管概念相近，都屬於集中式版本控管系統。這類集中式版本控管系統，使用上簡單、直覺且容易進行權限控管，說真的，在大部分的開發情境下，Subversion 或 TFS 已經相當足夠，那又是甚麼契機或是需求，迫使我們一定要轉換到 Git 版本控管呢？我相信，不同人採用 Git 一定有他的理由，有些人覺得好玩、有些人覺得新鮮、有些人覺得功能強大，無論如何，只要這個理由能夠支持你去主動認識一個陌生技術，都是好的，本篇文章除了帶大家認識 Git 版本控管機制外，也會說說我想轉換到 Git 的理由。

文章目的

在軟體開發領域，對原始碼進行版本控管是非常重要的一件事，有別於 Subversion 或 TFVC (Team Foundation Version Control) 這類集中式版本控管系統，Git 是一套分散式版本控管系統(DVCS; Distributed Version Control System)，並帶來許多版本控管上的各種優勢與解決傳統集中式版本控管的缺失，例如支援本地操作、備份容易、功能強大且彈性的分支與合併等等。不過，由於 Git 版本控管無論在版控觀念與工具使用上，都與傳統集中式版控工具差異甚大，因此造成了不小的學習門檻。

雖然說本次文章的主題是「30 天精通 Git 版本控管」，不過，說實在的，還真有點言過其實了，因為 Git 博大精深，有非常多細節可以探究，如果真的要應用在工作上，學幾天可以真正上手呢？每天學一點，連續學習 30 天，似乎是個合理的數字 (或太多?)，如果有一個工具大家都要用，而且要立刻上手的工具，如果學 30 天都還不知道怎麼活用，那這學習門檻也太高了些。因此我想，這個系列的文章，主要還是專注於「如何在 30 天內學會 Git 版本控管，而且必須要能熟練的應用在實務開發工作上」，這才是本系列的真正目的，那些繁瑣的細節，我不會特別強調，但總是有些重要的概念與細節還是不能錯過，我會嘗試在每一個主題中提到一部份，一有機會就會深入探討，希望大家可以透過做中學，深刻體會 Git 版本控管的強大魅力。

轉換的契機

這幾個月，公司因為有個大型專案，參與開發人數超過 12 人，最後大家決議採用 Git 作為本次專案的版本控管機制，與其說我們採用了 Git 版本控管，其實真正採用的原因是「我們選擇使用 GitHub 當成我們的版控平台」，原因就是 GitHub 平台實在整合得太好，完整的 Git 版控支援、議題追蹤與管理、線上 Wiki 文件管理、友善的原始碼審核(Code Review)介面。這些特性，都能有效協助我們在多人協同開發的過程中，減少團隊溝通的問題。

剛開始接觸 Git 說實在挺辛苦的，因為 Git 版本控管的觀念，實在與 Subversion 差太多，沒有辦法很直覺的去體會其差異，就算給了你 GUI 圖形化工具介面，你也不見得就會使用。你知道的，一個強大又好用的工具在你手上，「錯誤的使用方式」比「不會用」還可怕！說穿了，就是你必須先建立一套思維模式(Mindset)，了解 Git 的運作原理，然後再上手使用 Git 相關工具(無論是指令列工具或圖形化介面工具)，才是正途！

學習的方法

我在剛學習 Git 的時候，看了好幾本書 (其實是挑重點看)，也看了許多線上的文章與簡報，甚至還看了好幾部教學影片，看著看著，確實可以學會如何使用 Git 工具，我覺得並不會太過艱深。不過，Git 的指令與參數非常多，完全超出大腦能記憶的範圍，除非每天使用，否則哪有可能一天到晚打指令進行版控，如果每次要使用 Git 指令都要查書的話，那這也太沒效率了點，當下的我就直覺地認為，學習 Git 最終還是要回歸到好用的 GUI 工具，否則這東西在團隊中可能不容易推廣。

再者，因為 Git 是屬於「分散式版本控管」機制，當開發人數開始變多，版本庫又開始變成一人一份時，在第一次進行多人分支與合併的過程時，大家都飽受煎熬，而且持續一段不短的時間。雖然公司內部有先進行技術分享，不過由於大家都是第一次學，那些 Git 的抽象概念，還沒辦法深植人心，只能基於 Git 的使用方法進行分享，例如工具怎麼用、有哪些常用的指令、甚麼特殊的情況下應該下甚麼指令，諸如此類的。過程中就算說出了複雜的原理，由於大家對於 Git 的認知還很模糊，不同人對 Git 版控方式的理解也不盡相同，所吸收到的知識與概念，也不一定一致。所以，雖然上完課了，大家還是需要好幾天的時間不斷磨合，相互討論，互相解決問題，如果你只有一人使用 Git 的話，確實不容易感受 Git 帶來的好處，也恐怕不容易堅持下去。

所以，我認為，要學好 Git 版本控管，若先知道以下幾點，也許比較容易學會：

- 先擁有 Git 基礎觀念，透過下指令的方式學習是最快的方式，不要跳過這一段
- 找多一點人跟你一起學 Git 版本控管，最好能直接用在實務的開發工作上
- 團隊中最好要有幾個先遣部隊，可以多學一點 Git 觀念，好分享給其他人，或有人卡關時，能適時提供協助
- 了解 Git 屬於「分散式版本控管」，每個人都有一份完整的儲存庫(Repository)，所以必須經常合併檔案
- 使用 Git 的時候，分支與合併是常態，但只要有合併，就會有衝突，要學會如何解決衝突

認識 Git 版本控管

Git 的出現，來自於 Linux 之父 "Linus Torvalds" 開發 Linux kernel 的時候，因為早期的版本控制方法非常沒有效率，屬集中式控管，當 Linux kernel 這類複雜又龐大的專案在進行版本控管時，出現了許多問題。最早期 Linux kernel 採用 BitKeeper 進行版本控管，但後來 Linus Torvalds 基於 BitKeeper 與 Monotone 的使用經驗，設計出更棒的 Git 版控系統。原先 Git 只被設計成一個低階的版控工具，用來當做其他版控系統(SCM)的操作工具，後來才漸漸演變成一套完整的版本控制系統。

有趣的是，Linus Torvalds 改採 Git 進行版本控管初期，由於 Git 太過複雜，許多版控觀念跟以往差異太大，也受到世界各地開放原始碼社群的反對，但經過幾年的努力與發展，操作 Git 的相關工具也越來越成熟，才漸漸平撫反對的壓力，從 2013 年的市場調查看來，全世界已有 30% 的開放原始碼專案改採 Git 進行版本控管，這是個非常驚人的市占率，意謂著 Git 絕對有其驚豔之處，不好好研究一番還不行呢！

講到 Git 的架構，完全是基於 Linus Torvalds 在維護 Linux kernel 這個大型專案時得到的經驗，以及他本身在檔案系統優化方面的豐富經驗進行設計，也因為這樣，Git 包含了以下幾個重要的設計：

- 強力支援非線性開發模式 (分散式開發模式)
  - Git 擁有快速的分支與合併機制，還包括圖形化的工具顯示版本變更的歷史路徑。
  - Git 非常強調分支與合併，所以版本控管的過程中，你會不斷的在執行分支與合併動作。
  - Git 的分支機制非常輕量，沒有負擔，每一次的分支只是某個 commit 的參考指標而已。
- 分散式開發模型
  - 參與 Git 開發的每個人，都將擁有完整的開發歷史紀錄。
  - 當開發人員第一次將 Git 版本庫複製(clone)下來後，完全等同於這份 Git 版本庫的「完整備份」。
  - 整個版本庫中所有變更過的檔案與歷史紀錄，通通都會儲存在本機儲存庫(local repository)。
- 相容於現有作業系統
  - Git 版本庫其實就只是一個資料夾而已，資料夾中有許多相關的設定檔與各種 blob 物件檔案而已。
  - Git 版本庫可以用任何方式發布，所以你用 HTTP, FTP, rsync, SSH 甚至於用 Git protocol 都可以當成存取 Git 版本庫的媒介，相容性極高。
- 有效率的處理大型專案
  - 由於完整的版本庫會複製(clone)一份在本機，該版本庫包含完整的檔案與版本變更紀錄，所以針對版本控管中的各種檔案操作速度，將會比直接從遠端存取來的快上百倍之多。
  - 這也代表著，Git 版本控管不會因為專案越來越大、檔案越來越多，而導致速度變慢。
- 歷史紀錄保護
  - Git 版控的過程，每次 commit 都會產生一組 hash id 編號，而且每個版本在變化的過程都會參考到這個 hash id，只要 hash id 無法比對的上，Git 就會無法運作，所以當專案越來越大，版本庫複製(clone)的越來越多份，你幾乎無法竄改檔案的內容或版本紀錄。
  - **請記得: 每個人都有一份完整的版本庫，你改了原始的那份，所有人的版本庫就無法再合併回原本的版本庫了，所以你幾乎不可能任意竄改版本紀錄。**
- 以工具集為主的設計 (Toolkit-based design)
  - Git 被設計成一個一個的工具軟體(指令列工具)，你可以很輕易的組合不同工具的使用，使用上非常彈性。
- 彈性的合併策略 (Pluggable merge strategies)
  - Git 有一個擁有良好設計的「不完整合併(incomplete merge)」 機制，以及多種可以完成合併的演算法，並在最後告知使用者為何無法自動完成合併，或通知你需要手動進行合併動作。
- 被動的垃圾回收機制

    - 在使用 Git 的時候，若想要中斷目前的操作或回復上一個操作，都是可以的，你完全可以不必擔心可能有其中一個指令下錯，或指令執行到一半當機等問題。
    - Git 的垃圾回收機制，其實就是那些殘留在檔案系統中的無用檔案，這個垃圾回收機制只會在這些無用的物件累積一段時間後自動執行，或你也可以自行下達指令清空它。例如: git gc --prune
- 定期的封裝物件
    - 我們在 Git 中提到的 "物件" 其實就是代表版本庫中的一個檔案。而在版本異動的過程中，專案中的程式碼或其他檔案會被更新，每次更新時，只要檔案內容不一樣，就會建立一個新的 "物件"，這些不同內容的檔案全部都會保留下來。
    - 你應該可以想像，當一個專案越來越大、版本越來越多時，這個物件會越來越多，雖然每個檔案都可以各自壓縮讓檔案變小，不過過多的檔案還是會檔案存取變得越來越沒效率。因此 Git 的設計有個機制可以將一群老舊的 "物件" 自動封裝進一個封裝檔(packfile)中，以改善檔案存取效率。
    - 那些新增的檔案還是會以單一檔案的方式存在著，也代表一個 Git 版本庫中的 "檔案" 就是一個 Git "物件"，但每隔一段時間就會需要重新封裝(repacking)。
    - 照理說 Git 會自動執行重新封裝等動作，但你依然可以自行下達指令執行。例如: git gc
    - 如果你要檢查 Git 維護的檔案系統是否完整，可以執行以下指令: git fsck

關於 Git 的分散式版控系統，我再重申幾件事：

- Git 完全不需要伺服器端的支援就可以運作版本控制，因為每個人都有一份完整的儲存庫副本。
- 因為每個人都有一份完整的儲存庫副本，所以每次提交版本變更時，都僅提交到本地的儲存庫而已，因此提交速度非常快，也不用網路連線，可大幅節省開發時間。
- 由於每個人都有一份完整的儲存庫副本，代表著在使用 Git 版本控管時，沒有所謂的「權限控管」這件事，每個成員都能把儲存庫複製(clone)回來，也都可以在本地提交變更，沒有任何權限可以限制。使用 Git 時，唯一能設定的權限是，你有沒有權利存取上層儲存庫(upstream repository)或遠端儲存庫(remote repository)的權限。
- 如果需要跟別人交換變更後的版本，隨時可以透過「合併」的方式進行，Git 擁有非常強悍的合併追蹤（merge tracing）能力。
- 要合併多人的版本，你只要有存取共用儲存庫(shared repository)的權限或管道即可。 **例如：在同一台伺服器上可以透過資料夾權限進行共用，或透過 SSH 遠端存取另一台伺服器的 Git 儲存庫，也可以透過 Web 伺服器等方式來共用 Git 儲存庫。**

今日小結

今天這篇只是個大致介紹，若看不太懂 Git 的設計理念沒關係，你可以用一段時間之後再回來看這篇文章，或許會有更深一層的體會。

我覺得要寫「認識 Git 版本控管」比教大家怎麼用還難許多，以下我在列出一些 Git 相關連結，供大家進一步學習。

參考連結

- Git (software) - Wikipedia, the free encyclopedia
- Pro Git Book
- Git Magic - 繁體中文版
- 簡介 Git 及使用
- 版本控制 - 維基百科，自由的百科全書

- HOME
- 回目錄
- 下一天：在 Windows 平台必裝的三套 Git 工具