Git 魔法

Ben Lynn

2007 年 8 月

前言

Git 堪稱版本控制瑞士軍刀。這個可靠、多才多藝、用途多樣的校訂工具異常靈活，以致不易掌握，更別說精通了。

正如 Arthur C. Clarke 所說，足夠先進的技術與魔法無二。這是學習 Git 的好辦法：新手不妨忽略 Git 的內部機理，只當小把戲玩，借助 Git 其奇妙的能力，逗逗朋友，氣氣敵人。

為了不陷入細節，我們對特定功能提供大面上的講解。在反覆應用之後，慢慢地你會理解每個小技巧如何工作，以及如何組合這些技巧以滿足你的需求。

- 簡體中文: 俊傑，萌和江薇。正體中文 由 + cconv -f UTF8-CN -t UTF8-TW + 轉換。
- 法文: Alexandre Garel。也在 itaapy。
- 德文: Benjamin Bellee 和 Armin Stebich；也在 Armin 的網站。
- 葡萄牙文: Leonardo Siqueira Rodrigues [ODT 版]。
- 俄文: Tikhon Tarnavsky, Mikhail Dymskov, 和其他人。
- 西班牙: Rodrigo Toledo 和 Ariset Llerena Tapia。
- 越南文: Trần Ngọc Quân; 也在 他的網站.
- 單一檔案: 純 HTML，無 CSS。
- PDF 檔案: 打印效果好.
- Debian 包，Ubuntu 包: 本站快速本地拷貝。如果 下線了會方便些。
- 紙質書 [Amazon.com]: 64 頁, 15.24cm x 22.86cm, 黑白。沒有電子設備的時候會方便些。

致謝！

那麼多人對本文檔的翻譯讓我受寵若驚。他們的付出拓寬了讀者群，我非常感激。

Dustin Sallings、Alberto Bertogli、James Cameron、Douglas Livingstone、Michael Budde、Richard Albury、Tarmigan、Derek Mahar、Frode Aannevik、Keith Rarick、Andy Somerville、Ralf Recker、Øyvind A. Holm、Miklos Vajna、Sébastien Hinderer、Thomas Miedema、Joe Malin、和 Tyler Breisacher 對本文檔正確性和優化做出了貢獻。

François Marier 維護 Debian 包，該 Debian 包起初由 Daniel Baumann 創建。

感謝其他很多提供幫助和鼓勵的人。名單太長了我無法一一寫下。

如果我不小心把你的名字落下，請告訴我或者發一個補丁。

免費 **Git** 主機: 底下網站可以免費放置公開專案。非常感謝底下網站贊助放置手冊。

- http://repo.or.cz/
- http://gitorious.org/
- http://github.com/ 私有項目收錢。
- Assembla: 私有項目收錢, 雖然有 1 gigabyte 空間免費.

許可

入門

我將用類比方式來介紹版本控制的概念。更嚴謹的解釋參見 維基百科版本修訂控制條目。

工作是玩

我從小就玩電腦遊戲。相反，我只是在長大後才開始使用版本控制系統。我想我並不特殊，並且，對比兩者工作方式可使這些概念更易解釋，也易於理解。

編寫代碼，或編輯文檔和玩遊戲差不多。在你做出了很多進展之後，你最好保存一下。去做這個，會點擊你所信任的編輯器保存按鈕就好了。

但這將覆蓋老版本。就像那些學校裡玩的老遊戲，只有一個存檔：你確實可以保存，但你不能回到更老的狀態了。這真讓人掃興，因為那個狀態可能恰好保存了這個遊戲特別有意思一關，說不定哪天你想再玩一下呢。或者更糟糕的，你當前的保存是個必敗局，這樣你就不得不從頭開始玩了。

版本控制

在編輯的時候，如果想保留舊版本，你可以將檔案“另存為”一個不同的檔案，或在保存之前將檔案拷貝到別處。你可能壓縮這些檔案以節省空間。這是一個初級的靠手工的版本控制方式。遊戲軟件早就提高了這塊，很多都提供多個基于時間戳的自動存檔槽。

讓我們看看稍稍複雜的情況。比如你有很多放在一起的檔案，比如項目源碼，或網站文件。現在如你想保留舊版本，你不得不把整個目錄存檔。手工保存多個版本很不方便，而且很快會耗費巨大。

在一些電腦遊戲裡，一個存檔真的包含在一個充滿檔案的目錄裡。這些遊戲為玩家屏蔽了這些細節，並提供一個方便易用的界面來管理該目錄的不同版本。

版本控制系統也沒有兩樣。兩者提供友好的界面，來管理目錄裡的東西。你可以頻繁保存，也可以之後加載任一保存。不像大多計算機遊戲，版本控制系統通常精於節省存儲空間。一般情況如果兩個版本間只有少數檔案的變更，每個檔案的變更也不大，那就只存儲差異的部分，而不是把全部拷貝的都保存下來，以節省存儲空間。

分佈控制

現在設想一個很難的遊戲。太難打了，以至于世界各地很多骨灰級玩家決定組隊，分享他們遊戲存檔以攻克它。Speedrun 們就是實際中的例子：在同一個遊戲裡，玩家們分別攻克不同的等級，協同工作以創造驚人戰績。

你如何搭建一個系統，使得他們易於得到彼此的存檔？並易於上載新的存檔？

在過去，每個項目都使用中心式版本控制。某個伺服器上放所有保存的遊戲記錄。其他人就不用了。每個玩家在他們機器上最多保留幾個遊戲記錄。當一個玩家想更新進度時候，他們需要把最新進度從主伺服器下載下來，玩一會兒，保存並上載到主伺服器以供其他人使用。

假如一個玩家由於某種原因，想得到一個較舊版本的遊戲進度怎麼樣？或許當前保存的遊戲是一個注定的敗局，因為某人在第三級忘記撿某個物品；他們希望能找到最近一個可以完成的遊戲記錄。或者他們想比較兩個舊版本間的差異，來估算某個特定玩家幹了多少活。

查看舊版本的理由有很多，但檢查的辦法都是一樣的。他們必須去問中心伺服器要那個舊版本的記錄。需要的舊版本越多，和伺服器的交互就越多。

新一代的版本控制系統，Git 就是其中之一，是分散式的，可以被認作廣義上的中心式系統。從主伺服器下載時玩家會得到所有保存的記錄，而不僅是最新版。這看起來他們好像把中心伺服器做了個鏡像。

最初的克隆操作可能比較費時，特別當有很長歷史的時，但從長遠看這是值得的。一個顯而易見的好處是，當查看一個舊版本時，不再需要和中心伺服器通訊了。

一個誤區

一個很常見的錯誤觀念是，分散式系統不適合需要官方中心倉庫的項目。這與事實並不相符。給誰照相也不會偷走他們的靈魂。類似地，克隆主倉庫並不降低它的重要性。

一般來說，一個中心版本控制系統能做的任何事，一個良好設計的分散式系統都能做得更好。網絡資源總要比本地資源耗費更費。不過我們應該在稍後分析分散式方案的缺點，這樣人們才不會按照習慣做出錯誤的比較。

一個小項目或許只需要分散式系統提供的一小部分功能，但是，在項目很小的時候，應該用規劃不好的系統？就好比說，在計算較小數目的時候應該使用羅馬數字？

而且，你的項目的增長可能會超出你最初的預期。從一開始就使用 Git 好似帶著一把瑞士軍刀，儘管你很多時候只是用它來開開瓶蓋。某天你迫切需要一把改錐，你就會慶幸你所有的不單單是一個啟瓶器。

合併衝突

對於這個話題，電腦遊戲的類比顯得不夠用。那讓我們再來看看文檔編輯的情況吧。

假設 Alice 在文檔開頭插入一行，並且 Bob 在文檔末尾添加一行。他們都上傳了他們的改動。大多數系統將自動給出一個合理的處理方式：接受且合併他們的改動，這樣 Alice 和 Bob 兩人的改動都會生效。

現在假設 Alice 和 Bob 對檔案的同一行做了不同的改動。如果沒有人工參與的話，這個沖突是無法解決的。第二個人在上載檔案時，會收到 合併衝突的通知，要麼用一個人的改動覆蓋另一個的，要麼完全修訂這一行。

更複雜的情況也可能出現。版本控制系統自己處理相對簡單的情況，把困難的情況留給人來處理。它們的行為通常是可配置的。

基本技巧

與其一頭紮進 Git 命令的海洋中，不如來點基本的例子試試手。它們簡單而且實用。實際上，在開始使用 Git 的頭幾個月，我所用的從來沒超出本章介紹的內容。

保存狀態

要不來點猛的？在做之前，先為當前目錄所有檔案做個快照，使用：

```
$ git init
$ git add .
$ git commit -m "My first backup"
```

現在如果你的編輯亂了套，恢復之前的版本：

```
$ git reset --hard
```

再次保存狀態：

```
$ git commit -a -m "Another backup"
```

添加、刪除、重命名

以上命令將只跟蹤你第一次運行 **git add** 命令時就已經存在的檔案。如果要添加新文件或子目錄，你需要告訴 Git：

```
$ git add readme.txt Documentation
```

類似，如果你想讓 Git 忘記某些檔案：

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/
```

這些檔案如果還沒刪除，Git 刪除它們。

重命名檔案和先刪除舊檔案，再添加新檔案的一樣。也有一個快捷方式 **git mv** ，和 **mv** 命令的用法一樣。例如：

```
$ git mv bug.c feature.c
```

進階撤銷/重做

有時候你只想把某個時間點之後的所有改動都回滾掉，因為這些的改動是不正確的。那麼：

```
$ git log
```

來顯示最近提交列表，以及他們的 SHA1 哈希值:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800

    Replace printf() with write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Initial commit.
```

哈希值的前幾個字元足夠確定一個提交；也可以拷貝粘貼完整的哈希值，鍵入：

```
$ git reset --hard 766f
```

來恢復到一個指定的提交狀態，並從記錄裡永久抹掉所有比該記錄新一些的提交。

另一些時候你想簡單地跳到一個舊狀態。這種情況，鍵入：

```
$ git checkout 82f5
```

這個操作將把你帶回過去，同時也保留較新提交。然而，像科幻電影裡時光旅行一樣，如果你這時編輯並提交的話，你將身處另一個現實裡，因為你的動作與開始時相比是不同的。

這另一個現實叫作 “分支” (branch)，之後我們會對這點多討論一些。至于現在，只要記住：

```
$ git checkout master
```

會把你帶到當下來就可以了。另外，為避免 Git 的抱怨，應該在每次運行 checkout 之前提交 (commit) 或重置 (reset) 你的改動。

還以電腦遊戲作為類比：

- **git reset --hard**: 加載一個舊記錄並刪除所有比之新的記錄。
- **git checkout**: 加載一個舊記錄，但如果你在這個記錄上玩，遊戲狀態將偏離第一輪的較新狀態。你現在打的所有遊戲記錄會在你剛進入的、代表另一個真實的分支裡。我們稍後論述。

你可以選擇只恢復特定檔案和目錄，通過將其加在命令之後：

```
$ git checkout 82f5 some.file another.file
```

小心，這種形式的 **checkout** 會不聲不響地覆蓋檔案。為阻止意外發生，在運行任何 checkout 命令之前做提交，尤其在初學 Git 的時候。通常，任何時候你覺得對運行某個命令不放心，無論 Git 命令還是不是 Git 命令，就先運行一下 **git commit -a** 。

不喜歡拷貝站題哈希值？那就用：

```
$ git checkout :/"My first b"
```

來跳到以特定字元串開頭的提交。你也可以回到倒數第五個保存狀態：

```
$ git checkout master~5
```

撤銷

在法庭上，事件可以從法庭記錄裡敲出來。同樣，你可以檢出特定提交以撤銷。

```
$ git commit -a
$ git revert 1b6d
```

講撤銷給定哈希值的提交。本撤銷被記錄為一個新的提交，你可以通過運行 **git log** 來確認這一點。

變更日誌生成

一些項目要求生成變更日誌changelog. 生成一個，通過鍵入：

```
$ git log > ChangeLog
```

下載檔案

得到一個由 Git 管理的項目的拷貝，通過鍵入：

```
$ git clone git://server/path/to/files
```

例如，得到我用來創建該站的所有檔案：

```
$ git clone git://git.or.cz/gitmagic.git
```

我們很快會對 **clone** 命令談的很多。

到最新

如果你已經使用 **git clone** 命令得到了一個項目的一份拷貝，你可以更新到最新版，通過：

```
$ git pull
```

快速發佈

假設你寫了一個腳本，想和他人分享。你可以只告訴他們從你的計算機下載，但如果此時你正在改進你的腳本，或加入試驗性質的改動，他們下載了你的腳本，他們可能由此陷入困境。當然，這就是發佈周期存在的原因。開發人員可能頻繁進行項目修改，但他們只在他們覺得代碼可以見人的時候才擇時發佈。

用 Git 來完成這項，需要進入你的腳本所在目錄：

```
$ git init
$ git add .
$ git commit -m "First release"
```

然後告訴你的用戶去運行：

```
$ git clone your.computer:/path/to/script
```

來下載你的腳本。這要假定他們有 ssh 訪問權限。如果沒有，需要運行 **git daemon** 並告訴你的用戶去運行：

```
$ git clone git://your.computer/path/to/script
```

從現在開始，每次你的腳本準備好發佈時，就運行：

```
$ git commit -a -m "Next release"
```

並且你的用戶可以通過進入包含你腳本的目錄，並鍵入下列命令，來更新他們的版本：

```
$ git pull
```

你的用戶永遠也不會取到你不想讓他們看到的腳本版本。顯然這個技巧對所有的東西都是可以，不僅是對腳本。

我們已經做了什麼？

找出自從上次提交之後你已經做了什麼改變：

```
$ git diff
```

或者自昨天的改變：

```
$ git diff "@{yesterday}"
```

或者一個特定版本與倒數第二個變更之間：

```
$ git diff 1b6d "master~2"
```

輸出結果都是補丁格式，可以用 **git apply** 來把補丁打上。也可以試一下：

```
$ git whatchanged --since="2 weeks ago"
```

我也經常用qgit 瀏覽歷史, 因為他的圖形界面很養眼，或者 tig ，一個文本界面的東西，很慢的網絡狀況下也工作的很好。也可以安裝 web 伺服器，運行 **git instaweb** ，就可以用任何瀏覽器瀏覽了。

練習

比方 A，B，C，D 是四個連續的提交，其中 B 與 A 一樣，除了一些檔案刪除了。我們想把這些刪除的檔案加回 D。我們如何做到這個呢？

至少有三個解決方案。假設我們在 D：

1. A 與 B 的差別是那些刪除的檔案。我們可以創建一個補丁代表這些差別，然後吧補丁打上：

   ```
   $ git diff B A | git apply
   ```

2. 既然這些檔案存在 A，我們可以把它們拿出來：

   ```
   $ git checkout A foo.c bar.h
   ```

3. 我們可以把從 A 到 B 的變化視為可撤銷的變更：

   ```
   $ git revert B
   ```

哪個選擇最好？這取決於你的喜好。利用 Git 滿足自己需求是容易，經常還有多個方法。

克隆周邊

在較老一代的版本控制系統裡，checkout 是獲取檔案的標準操作。你將獲得一組特定保存狀態的檔案。

在 Git 和其他分散式版本控制系統裡，克隆是標準的操作。通過創建整個倉庫的克隆來獲得檔案。或者說，你實際上把整個中心伺服器做了個鏡像。凡是主倉庫上能做的事，你都能做。

計算機間同步

我可以忍受製作 tar 包或利用 rsync 來作備份和基本同步。但我有時在我筆記本上編輯，其他時間在台式機上，而且這倆之間也許並不交互。

在一個機器上初始化一個 Git 倉庫並提交你的檔案。然後轉到另一台機器上：

```
$ git clone other.computer:/path/to/files
```

以創建這些檔案和 Git 倉庫的第二個拷貝。從現在開始，

```
$ git commit -a
$ git pull other.computer:/path/to/files HEAD
```

將把另一台機器上特定狀態的檔案“拉”到你正工作的機器上。如果你最近對同一個文件做了有衝突的修改，Git 將通知你，而你也應該在解決衝突之後再次提交。

典型源碼控制

為你的檔案初始化 Git 倉庫：

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

在中心伺服器，在某個目錄初始化一個“裸倉庫”：

```
$ mkdir proj.git
$ cd proj.git
$ git init --bare
$ touch proj.git/git-daemon-export-ok
```

如需要的話，啟動 Git 守護進程：

```
$ git daemon --detach  # 它也許已經在運行了
```

對一些 Git 伺服服務，按照其指導來初始化空 Git 倉庫。一般是在網頁上填一個表單。

把你的項目“推”到中心伺服器：$ git push central.server/path/to/proj.git HEAD

撿出源碼，可以鍵入：

```
$ git clone central.server/path/to/proj.git
```

做了改動之後，開發保存變更到本地：

```
$ git commit -a
```

更新到最近版本：

```
$ git pull
```

所有衝突應被處理，然後提交：

```
$ git commit -a
```

把本地改動撿入到中心倉庫：

```
$ git push
```

如果主伺服器由於其他開發的活動，有了新的變更，這個撿入會失敗，該開發應該把最新版本拿下來，解決合併衝突，然後重試。

為使用上面 pull 和 push 命令，開發必須有 SSH 訪問權限。不過，通過鍵入以下命令，任何人都可以看到源碼：

```
$ git clone git://central.server/path/to/proj.git
```

本地 git 協議和 HTTP 類似：並無安全驗證，因此任何人都能拿到項目。因此，預設情況 git 協議禁止推操作。

封閉源碼

閉源項目不要執行 touch 命令，並確保你從未創建 ‘git-daemon-export-ok‘檔案。資源庫不再可以通過 git 協議獲取；只有那些有 SSH 訪問權限的人才能看到。如果你所有的資源庫都是封閉的，那也沒必要運行運行 git 守護了，因為所有溝通都走 SSH。

裸倉庫

之所以叫裸倉庫是因為其沒有工作目錄；它只包含正常情況下隱藏在 ‘.git‘子目錄下的檔案。換句話說，它維護項目歷史，而且從不保存任何給定版本的快照。

裸倉庫扮演的角色和中心版本控制系統中中心伺服器的角色類似：你項目的中心。開發從其中克隆項目，撿入新近改動。典型地裸倉庫存在一個伺服器上，該伺服器除了分散數據外並不做啥。開發活動發生在克隆上，因此中心倉庫沒有工作目錄也行。

很多 Git 命令在裸倉庫上失敗，除非指定倉庫路徑到環境變數 ‘GIT_DIR‘，或者指定 ‘--bare‘選項。

推還是拽

為什麼我們介紹了 push 命令，而不是依賴熟悉的 pull 命令？首先，在裸倉庫上 pull 會失敗：除非你必須 “fetch”，一個之後我們要討論的命令。但即使我們在中心伺服器上保持一個正常的倉庫，拽些東西進去仍然很繁瑣。我們不得不登陸伺服器先，給 pull 命令我們要拽自機器的網絡地址。防火牆會阻礙，並且首先如果我們沒有到伺服器的 shell 訪問怎麼辦呢？

然而，除了這個案例，我們反對推進倉庫，因為當目標有工作目錄時，困惑隨之而來。

簡短截說，學習 Git 的時候，只在目標是裸倉庫的時候 push，否則用 pull 的方式。

項目分叉

項目走歪了嗎？或者認為你可以做得更好？那麼在伺服器上：

```
$ git clone git://main.server/path/to/files
```

之後告訴每個相關的人你伺服器上項目的分支。

在之後的時間，你可以合併來自原先項目的改變，使用命令：

```
$ git pull
```

終極備份

會有很多散佈在各處，禁止篡改的冗餘存檔嗎？如果你的項目有很多開發，那乾脆啥也別做了。你的每份代碼克隆是一個有效備份。不僅當前狀態，還包括你項目整個歷史。感謝哈希加密算法，如果任何人的克隆被損壞，只要他們與其他的交互，這個克隆就會被修好。

如果你的項目並不是那麼流行，那就找儘可能多的伺服來放克隆吧。

真正的偏執狂應該總是把 HEAD 最近 20 位元組的 SHA1 哈希值寫到安全的地方。應該保證安全，而不是把它藏起來。比如，把它發佈到報紙上就不錯，因為對攻擊者而言，更改每份報紙是很難的。

輕快多任務

比如你想並行開發多個功能。那麼提交你的項目並運行：

```
$ git clone . /some/new/directory
```

Git 使用硬連結和檔案共享來儘可能安全地創建克隆，因此它一眨眼就完成了，因此你現在可以並行操作兩個沒有相互依賴的功能。例如，你可以編輯一個克隆，同時編譯另一個。感謝 hardlinking, 本地克隆比簡單備份省時省地。

現在你可以同時工作在兩個彼此獨立的特性上。比如，你可以在編譯一個克隆的時候編輯另一個克隆。任何時候，你都可以從其它克隆提交並拖拽變更。

```
$ git pull /the/other/clone HEAD
```

游擊版本控制

你正做一個使用其他版本控制系統的項目，而你非常思念 Git？那麼在你的工作目錄初始化一個 Git 倉庫：

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

然後克隆它：

```
$ git clone . /some/new/directory
```

並在這個目錄工作，按你所想在使用 Git。過一會，一旦你想和其他每個人同步，在這種情況下，轉到原來的目錄，用其他的版本控制工具同步，並鍵入：

```
$ git add .
$ git commit -m "Sync with everyone else"
```

現在轉到新目錄運行:

```
$ git commit -a -m "Description of my changes"
$ git pull
```

把你的變更提交給他人的過程依賴于其他版本控制系統。這個新目錄包含你的改動的文件。需要運行其他版本控制系統的命令來上載這些變更到中心倉庫。

Subversion, 或許是最好的中心式版本控制系統,為無數項目所用。**git svn** 命令為 Subversion 倉庫自動化了上面的操作，並且也可以用作 導出 Git 項目到 Subversion 倉庫 的替代。

Mercurial

Mercurial 是一個類似的的版本控制系統，几乎可以和 Git 一起無縫工作。使用 'hg-git'插件，一個 Mercurial 用戶可以無損地往 Git 倉庫推送，從 Git 倉庫拖拽。

使用 Git 獲得 'hg-git'插件:

```
$ git clone git://github.com/schacon/hg-git.git
```

或使用 Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

不好意思，我沒注意 Git 有類似的插件。因此, 我主張使用 Git 而不是 Mercurial 作為主資源庫，即使你偏愛 Mercurial。使用 Mercurial 項目，通常一個自願者維護一個平行的 Git 項目以適應 Git 用戶，然而感謝 'hg-git'插件，一個 Git 項目自動地適應 Mercurial 用戶。

儘管該插件可以把一個 Mercurial 倉庫轉成一個 Git 倉庫，通過推到一個空的倉庫，這個差事交給 'hg-fast-export.sh'腳本還是更容易些。來自:

```
$ git clone git://repo.or.cz/fast-export.git
```

要轉化，只需在一個空目錄運行:

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

注意該腳本應加入你的 '$PATH'。

Bazaar

我們簡略提一下 Bazaar，它畢竟是緊跟 Git 和 Mercurial 之後最流行的自由分散式版本控制系統。

Bazaar 有後來者的優勢，它相對年輕些；它的設計者可以從前人的錯誤中學習，並且躲過去翻歷史上犯過的錯誤。另外，它的開發人員對可移植性以及和與其它版本控制系統的互操作性也考慮周全。

一個 ‘bzr-git‘插件讓 Bazaar 用戶在一定程度下可以工作在 Git 倉庫。‘tailor‘程序轉換 Bazaar 倉庫到 Git 倉庫，並且可以遞增的方式做，要知道 ‘bzr-fast-export‘只是在一次性轉換性情況下工作良好。

我偏愛 Git 的原因

我起先選擇 Git 是因為我聽說它能管理不可想象地不可管理的 Linux 內核源碼。我從來沒覺得有離開的必要。Git 已經服侍的很好了，並且我也沒有被其瑕疵所困擾。因為我主要使用 Linux，其他平台上的問題與我無關。

還有，我偏愛 C 程序和 bash 腳本，以及諸如 Python 的可執行可腳本：較少依賴，並且我也沉迷于快速的執行時間。

我考慮過 Git 才能如何提高，甚至自己寫類似的工具，但只作為研究練練手。即使完成這個項目，我也無論如何會繼續使用 Git，因為使用一個古裡古怪的系統所獲甚微。

自然地，你的需求和期望可能不同，並且你可能使用另一個系統會好些。儘管如此，使用 Git 你都錯不太遠。

分支巫術

即時分支合併是 Git 最給力的殺手鐧。

問題：外部因素要求必須切換場景。在發佈版本中突然蹦出個嚴重缺陷。某個特性完成的截至日期就要來臨。在項目關鍵部分可以提供幫助的一個開發正打算離職。所有情況逼迫你停下所有手頭工作，全力撲到到這個完全不同的任務上。

打斷思維的連續性會使你的生產力大大降低，並且切換上下文也更麻煩，更大的損失。使用中心版本控制我們必須從中心伺服器下載一個新的工作拷貝。分散式系統的情況就好多了，因為我們能夠在本地克隆所需要的版本。

但是克隆仍然需要拷貝整個工作目錄，還有直到給定點的整個歷史記錄。儘管 Git 使用文件共享和硬連結減少了花費，項目檔案自身還是必須在新的工作目錄裡重建。

方案：Git 有一個更好的工具對付這種情況，比克隆快多了而且節省空間：**git branch** 。

使用這個魔咒，目錄裡的檔案突然從一個版本變到另一個。除了只是在歷史記錄裡上跳下竄外，這個轉換還可以做更多。你的檔案可以從上一個發佈版變到實驗版本到當前開發版本到你朋友的版本等等。

老闆鍵

曾經玩過那樣的遊戲嗎？按一個鍵（“老闆鍵”），屏幕立即顯示一個電子表格或別的？那麼如果老闆走進辦公室，而你正在玩遊戲，就可以快速將遊戲藏起來。

在某個目錄：

```
$ echo "I'm smarter than my boss" > myfile.txt
$ git init
$ git add .
$ git commit -m "Initial commit"
```

我們已經創建了一個 Git 倉庫，該倉庫記錄一個包含特定信息的檔案。現在我們鍵入：

```
$ git checkout -b boss  # 之後似乎沒啥變化
$ echo "My boss is smarter than me" > myfile.txt
$ git commit -a -m "Another commit"
```

看起來我們剛剛只是覆蓋了原來的檔案並提交了它。但這是個錯覺。鍵入：

```
$ git checkout master  # 切到檔案的原先版本
```

嘿真快！這個檔案就恢復了。並且如果老闆決定窺視這個目錄，鍵入：

```
$ git checkout boss  # 切到適合老闆看的版本
```

你可以在兩個版本之間相切多少次就切多少次，而且每個版本都可以獨立提交。

骯髒的工作

比如你正在開發某個特性，並且由於某種原因，你需要回退三個版本，臨時加進幾行打印語句來，來看看一些東西是如何工作的。那麼：

```
$ git commit -a
$ git checkout HEAD~3
```

現在你可以到處加醜陋的臨時代碼。你甚至可以提交這些改動。當你做完的時候，

```
$ git checkout master
```

來返回到你原來的工作。看，所有未提交變更都結轉了。

如果你後來想保存臨時變更怎麼辦？簡單：

```
$ git checkout -b dirty
```

只要在切換到主分支之前提交就可以了。無論你什麼時候想回到髒的變更，只需鍵入：

```
$ git checkout dirty
```

我們在前面章節討論加載舊狀態的時候，曾經接觸過這個命令。最終我們把故事說全：檔案改變成請求的狀態，但我們必須離開主分支。從現在開始的任何提交都會將你的文件提交到另一條不同的路，這個路可以之後命名。

換一個說法，在 checkout 一個舊狀態之後，Git 自動把你放到一個新的，未命名的分支，這個分支可以使用 **git checkout -b** 來命名和保存。

快速修訂

你正在做某件事的當間，被告知先停所有的事情，去修理一個新近發現的臭蟲，這個臭蟲在提交‘1b6d...‘：

```
$ git commit -a
$ git checkout -b fixes 1b6d
```

那麼一旦你修正了這個臭蟲：

```
$ git commit -a -m "Bug fixed"
$ git checkout master
```

並可以繼續你原來的任務。你甚至可以“合併”到最新修訂：

```
$ git merge fixes
```

合併

一些版本控制系統，創建分支很容易，但把分支合併回來很難。使用 Git，合併簡直是家常便飯，以至于甚至你可能對其發生沒有察覺。

我們很久之前就遇到合併了。**pull** 命令取出提交併合並它們到你的當前分支。如果你沒有本地變更，那這個合併就是一個“快進”，相當於中心式版本控制系統裡的一個弱化的獲取最新版本操作。但如有本地變更，Git 將自動合併，並報告任何衝突。

通常，一個提交只有一個“父提交”，也叫前一個提交。合併分支到一起產生一個至少有兩個父的提交。這就引出了問題：HEAD~10 真正指哪個提交？一個提交可能有多個父，那我們跟哪個呢？

原來這個表示每次選擇第一個父。這是可取的，因為在合併時候當前分支成了第一個父；多數情況下我們只關注我們在當前分支都改了什麼，而不是從其他分支合併來的變更。

你可以用插入符號來特別指定父。比如，顯示來自第二個父的日誌：

```
$ git log HEAD^2
```

你可以忽略數字以指代第一個父。比如，顯示與第一個父的差別：

```
$ git diff HEAD^
```

你可以結合其他類型使用這個記號。比如：

```
$ git checkout 1b6d^^2~10 -b ancient
```

開始一個新分支“ancient”，表示第一個父的第二個父的倒數第十次提交的狀態。

不間斷工作流

經常在硬件項目裡，計劃的第二步必須等第一步完成才能開始。待修的汽車傻等在車庫裡，直到特定的零件從工廠運來。一個原型在其可以構建之前，可能苦等晶片成型。

軟件項目可能也類似。新功能的第二部分不得不等待，直到第一部分發佈並通過測試。一些項目要求你的代碼需要審批才能接受，因此你可能需要等待第一部分得到批准，才能開始第二部分。

多虧了無痛分支合併，我們可以不必遵循這些規則，在第一部分正式準備好前開始第二部分的工作。假設你已經將第一部分提交並發去審批，比如說你現在在主分支。那麼分岔：

```
$ git checkout -b part2
```

接下來，做第二部分，隨時可以提交變更。只要是人就可能犯錯誤，經常你將回到第一部分在修修補補。如果你非常幸運，或者超級棒，你可能不必做這幾行：

```
$ git checkout master  # 回到第一部分
$ 修復問題
$ git commit -a        # 提交變更
$ git checkout part2   # 回到第二部分
$ git merge master     # 合併這些改動
```

最終，第一部分獲得批准：

```
$ git checkout master  # 回到第一部分
$ submit files         # 對世界發佈
$ git merge part2      # 合併第二部分
$ git branch -d part2  # 刪除分支“part2”
```

現在你再次處在主分支，第二部分的代碼也在工作目錄。

很容易擴展這個技巧，應用到任意數目的部分。它也很容易追溯分支：假如你很晚才意識到你本應在 7 次提交前就創建分支。那麼鍵入：

```
$ git branch -m master part2  # 重命名“master”分支為“part2”。
$ git branch master HEAD~7    # 以七次前提交建一個新的“master”。
```

分支 master 只有第一部分內容，其他內容在分支 part2 。我們現在後一個分支；我們創建了 master 分支還沒有切換過去，因為我們想繼續工作在 part2 。這是不尋常的。直到現在，我們已經在創建之後切換到分支，如：

```
$ git checkout HEAD~7 -b master  # 創建分支，並切換過去。
```

重組雜亂

或許你喜歡在同一個分支下完成工作的方方面面。你想為自己保留工作進度並希望其他人只能看到你仔細整理過後的提交。開啟一對分支：

```
$ git branch sanitized    # 為乾淨提交創建分支
$ git checkout -b medley  # 創建並切換分支以進去工作
```

接下來，做任何事情：修臭蟲，加特性，加臨時代碼，諸如此類，經常按這種方式提交。然後：

```
$ git checkout sanitized
$ git cherry-pick medley^^
```

應用分支“medley”的祖父提交到分支“sanitized”。通過合適的挑選（像選櫻桃那樣）你可以構建一個只包含成熟代碼的分支，而且相關的提交也組織在一起。

管理分支

列出所有分支：

```
$ git branch
```

預設你從叫“master”的分支開始。一些人主張別碰“master”分支，而是創建你自己版本的新分支。

選項 **-d** 和 **-m** 允許你來刪除和移動（重命名）分支。參見 **git help branch**。

分支“master”是一個有用的慣例。其他人可能假定你的倉庫有一個叫這個名字的分支，並且該分支包含你項目的官方版本。儘管你可以重命名或抹殺“master”分支，你最好還是尊重這個約定。

臨時分支

很快你會發現你經常會因為一些相似的原因創建短期的分支：每個其它分支只是為了保存當前狀態，那樣你就可以直接跳到較老狀態以修復高優先順序的臭蟲之類。

可以和電視的換台做類比，臨時切到別的頻道，來看看其它台那正放什麼。但並不是簡單地按幾個按鈕，你不得不創建，檢出，合併，以及刪除臨時分支。幸運的是，Git 已經有了和電視機遙控器一樣方便的快捷方式：

```
$ git stash
```

這個命令保存當前狀態到一個臨時的地方（一個隱藏的地方）並且恢復之前狀態。你的工作目錄看起來和你開始編輯之前一樣，並且你可以修復臭蟲，引入之前變更等。當你想回到隱藏狀態的時候，鍵入：

```
$ git stash apply  # 你可能需要解決一些衝突
```

你可以有多個隱藏，並用不同的方式來操作他們。參見 **git help slash**。也許你已經猜到，Git 維護在這個場景之後的分支以執行魔法技巧.

按你希望的方式工作

你可能猶疑于分支是否值得一試。畢竟，克隆也几乎一樣快，並且你可以用 **cd** 來在彼此之間切換，而不是用 Git 深奧的命令。

考慮一下瀏覽器。為什麼同時支持多標籤和多窗口？因為允許兩者同時接納納了多種風格的用戶。一些用戶喜歡只保持一個打開的窗口，然後用標籤瀏覽多個網頁。一些可能堅持另一個極端：任何地方都沒有標籤的多窗口。一些喜好處在兩者之間。

分支類似你工作目錄的標籤，克隆類似打開的瀏覽器新窗口。這些是本地操作很快，那為什麼不試着找出最適合你的組合呢？Git 讓你按你確實所希望的那樣工作。

關於歷史

Git 分散式本性使得歷史可以輕易編輯。但你若篡改過去，需要小心：只重寫你獨自擁有的那部分。正如民族間會無休止的爭論誰犯下了什麼暴行一樣，如果在另一個人的克隆裡，歷史版本與你的不同，當你們的樹互操作時，你會遇到一致性方面的問題。

一些開發人員強烈地感覺歷史應該永遠不變，不好的部分也不變所有都不變。另一些覺得代碼樹在向外發佈之前，應該整得漂漂亮亮的。Git 同時支持兩者的觀點。像克隆，分支和合併一樣，重寫歷史只是 Git 給你的另一強大功能，至于如何明智地使用它，那是你的事了。

我認錯

剛提交，但你期望你輸入的是一條不同的信息？那麼鍵入：

```
$ git commit --amend
```

來改變上一條信息。意識到你還忘記了加一個檔案？運行 git add 來加，然後運行上面的命令。

希望在上次提交裡包括多一點的改動？那麼就做這些改動並運行：

```
$ git commit --amend -a
```

更複雜情況

假設前面的問題還要糟糕十倍。在漫長的時間裡我們提交了一堆。但你不太喜歡他們的組織方式，而且一些提交信息需要重寫。那麼鍵入：

```
$ git rebase -i HEAD~10
```

並且後 10 個提交會出現在你喜愛的 $EDITOR。一個例子：

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

之後：

- 通過刪除行來移去提交。
- 通過為行重新排序行來重新排序提交。
- 替換 pick 使用：
  - edit 標記一個提交需要修訂。
  - reword 改變日誌信息。
  - squash 將一個提交與其和前一個合併。
  - fixup 將一個提交與其和前一個合併，並丟棄日誌信息。

保存退出。如果你把一個提交標記為可編輯，那麼運行

```
$ git commit --amend
```

否則，運行：

```
$ git rebase --continue
```

這樣儘早提交，經常提交：你之後還可以用 rebase 來規整。

本地變更之後

你正在一個活躍的項目上工作。隨着時間推移，你做了幾個本地提交，然後你使用合併與官方版本同步。在你準備好提交到中心分支之前，這個循環會重複幾次。

但現在你本地 Git 克隆摻雜了你的改動和官方改動。你更期望在變更列表裡，你所有的變更能夠連續。

這就是上面提到的 **git rebase** 所做的工作。在很多情況下你可以使用 **--onto** 標記以避免交互。

另外參見 **git help rebase** 以獲取這個讓人驚奇的命令更詳細的例子。你可以拆分提交。你甚至可以重新組織一棵樹的分支。

重寫歷史

偶爾，你需要做一些代碼控制，好比從正式的照片中去除一些人一樣，需要從歷史記錄裡面徹底的抹掉他們。例如，假設我們要發佈一個項目，但由於一些原因，項目中的某個檔案不能公開。或許我把我的信用卡號記錄在了一個文本檔案裡，而我又意外的把它加入到了這個項目中。僅僅刪除這個檔案是不夠的，因為從別的提交記錄中還是可以訪問到這個檔案。因此我們必須從所有的提交記錄中徹底刪除這個檔案。

```
$ git filter-branch --tree-filter 'rm top/secret/file' HEAD
```

參見 **git help filter-branch** ，那裡討論了這個例子並給出一個更快的方法。一般地，**filter-branch** 允許你使用一個單一命令來大範圍地更改歷史。

此後，+.git/refs/original+ 目錄描述操作之前的狀態。檢查命令 filter-branch 的確做了你想要做的，然後刪除此目錄，如果你想運行多次 filter-branch 命令。

最後，用你修訂過的版本替換你的項目克隆，如果你想之後和它們交互的話。

製造歷史

想把一個項目遷移到 Git 嗎？如果這個項目是在用比較有名氣的系統，那可以使用一些其他人已經寫好的腳本，把整個項目歷史記錄導出來放到 Git 裡。

否則，查一下 **git fast-import** ，這個命令會從一個特定格式的文本讀入，從頭來創建 Git 歷史記錄。通常可以用這個命令很快寫一個腳本運行一次，一次遷移整個項目。

作為一個例子，粘貼以下所列到臨時檔案，比如/tmp/history：

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT


commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Replace printf() with write().
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

之後從這個臨時檔案創建一個 Git 倉庫，鍵入：

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

你可以從這個項目 checkout 出最新的版本，使用：

```
$ git checkout master .
```

命令 **git fast-export** 轉換任意倉庫到 **git fast-import** 格式，你可以研究其輸出來寫導出程序，也以可讀格式傳送倉庫。的確，這些命令可以發送倉庫文本檔案通過只接受文本的渠道。

哪兒錯了？

你剛剛發現程序裡有一個功能出錯了，而你十分確定幾個月以前它運行的很正常。天啊！這個臭蟲是從哪裡冒出來的？要是那時候能按照開發的內容進行過測試該多好啊。

現在說這個已經太晚了。然而，即使你過去經常提交變更，Git 還是可以精確的找出問題所在：

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git 從歷史記錄中檢出一個中間的狀態。在這個狀態上測試功能，如果還是有問題：

```
$ git bisect bad
```

如果可以工作了，則把"bad" 替換成"good"。Git 會再次幫你找到一個以確定的好版本和壞版本之間的狀態，通過這種方式縮小範圍。經過一系列的迭代，這種二分搜索會幫你找到導致這個錯誤的那次提交。一旦完成了問題定位的調查，你可以返回到原始狀態，鍵入：

```
$ git bisect reset
```

不需要手工測試每一次改動，執行如下命令可以自動的完成上面的搜索：

```
$ git bisect run my_script
```

Git 使用指定命令（通常是一個一次性的腳本）的返回值來決定一次改動是否是正確的：命令退出時的代碼 0 代表改動是正確的，125 代表要跳過對這次改動的檢查，1 到 127 之間的其他數值代表改動是錯誤的。返回負數將會中斷整個 bisect 的檢查。

你還能做更多的事情：幫助文檔解釋了如何展示 bisects, 檢查或重放 bisect 的日誌, 並可以通過排除對已知正確改動的檢查，得到更好的搜索速度。

誰讓事情變糟了？

和其他許多版本控制系統一樣，Git 也有一個"blame" 命令：

```
$ git blame bug.c
```

這個命令可以標註出一個指定的檔案裡每一行內容的最後修改者，和最後修改時間。但不像其他版本控制系統，Git 的這個操作是在綫下完成的，它只需要從本地磁碟讀取信息。

個人經驗

在一個中心版本控制系統裡，歷史的更改是一個困難的操作，並且只有管理員才有權這麼做。沒有網絡，克隆，分支和合併都沒法做。像一些基本的操作如瀏覽歷史，或提交變更也是如此。在一些系統裡，用戶使用網絡連接僅僅是為了查看他們自己的變更，或打開檔案進行編輯。

中心繫統排斥離線工作，也需要更昂貴的網絡設施，特別是當開發人員增多的時候。最重要的是，所有操作都一定程度變慢，一般在用戶避免使用那些能不用則不用的高級命令時。在極端的情況

下，即使是最基本的命令也會變慢。當用戶必須運行緩慢的命令的時候，由於工作流被打斷，生產力降低。

我有這些的一手經驗。Git 是我使用的第一個版本控制系統。我很快學會適應了它，用了它提供的許多功能。我簡單地假設其他系統也是相似的：選擇一個版本控制系統應該和選擇一個編輯器或瀏覽器沒啥兩樣。

在我之後被迫使用中心繫統的時候，我被震驚了。我那有些脆弱的網絡沒給 Git 帶來大麻煩，但是當它需要像本地硬碟一樣穩定的時候，它使開發困難重重。另外，我發現我自己有選擇地避免特定的命令，以避免踏雷，這極大地影響了我，使我不能按照我喜歡的方式工作。

當我不得不運行一個慢的命令的時候，這種等待極大地破壞了我思緒連續性。在等待服務器通訊完成的時候，我選擇做其他的事情以度過這段時光，比如查看郵件或寫其他的文檔。當我返回我原先的工作場景的時候，這個命令早已結束，並且我還需要浪費時間試圖記起我之前正在做什麼。人類不擅長場景間的切換。

還有一個有意思的大眾悲劇效應：預料到網絡擁擠，為了減少將來的等待時間，每個人將比以往消費更多的頻寬在各種操作上。共同的努力加劇了擁擠，這等於是鼓勵個人下次消費更多頻寬以避免更長時間的等待。

多人 Git

我最初在一個私人項目上使用 Git，那裡我是唯一的開發。在與 Git 分散式本性有關的命令中，我只用到了 **pull** 和 *clone*，用以在不同地方保持同一個項目。

後來我想用 Git 發佈我的代碼，並且包括其他貢獻者的變更。我不得不學習如何管理有來自世界各地的多個開發的項目，幸運的是，這是 Git 的長處，也可以說是其存在的理由。

我是誰？

每個提交都有一個作者姓名和電子信箱，這顯示在 **git log** 裡。預設，Git 使用系統設定來填充這些域。要顯示地設定，鍵入：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

去掉 global 選項設定只對當前倉庫生效。

Git 在 SSH, HTTP 上

假設你有 ssh 訪問權限，以訪問一個網頁伺服器，但上面並沒有安裝 Git。儘管比着它的原生協議效率低，Git 也是可以通過 HTTP 來進行通信的。

那麼在你的帳戶下，下載，編譯並安裝 Git。在你的網頁目錄裡創建一個 Git 倉庫：

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

對較老版本的 Git，只拷貝還不夠，你應運行：

```
$ chmod a+x hooks/post-update
```

現在你可以通過 SSH 從隨便哪個克隆發佈你的最新版本：

```
$ git push web.server:/path/to/proj.git master
```

那隨便誰都可以通過如下命令得到你的項目：

```
$ git clone http://web.server/proj.git
```

Git 在隨便什麼上

想無需伺服器，甚至無需網絡連接的時候同步倉庫？需要在緊急時期湊合一下？我們已經看過**git fast-export** 和 **git fast-import** 可以轉換資源庫到一個單一檔案以及轉回來。我們可以來來會會傳送這些檔案以傳輸 git 倉庫，通過任何媒介，但一個更有效率的工具是 **git bundle** 。

發送者創建一個“檔案包”：

```
$ git bundle create somefile HEAD
```

然後傳輸這個檔案包，somefile ，給某個其他參與者：電子郵件，優盤，一個 **xxd** 打印品和一個 OCR 掃描器，通過電話讀位元組，狼煙，等等。接收者通過鍵入如下命令從檔案包獲取提交：

```
$ git pull somefile
```

接收者甚至可以在一個空倉庫做這個。不考慮大小，somefile 可以包含整個原先 git 倉庫。

在較大的項目裡，可以通過只打包其他倉庫缺少的變更消除浪費。例如，假設提交 “1b6d...” 是兩個參與者共享的最近提交：

```
$ git bundle create somefile HEAD ^1b6d
```

如果做的頻繁，人可能容易忘記剛發了哪個提交。幫助頁面建議使用標籤解決這個問題。即，在你發了一個檔案包後，鍵入：

```
$ git tag -f lastbundle HEAD
```

並創建較新檔案包，使用：

```
$ git bundle create newbundle HEAD ^lastbundle
```

補丁：全球貨幣

補丁是變更的文本形式，易於計算機理解，人也類似。補丁可以通吃。你可以給開發電郵一個補丁，不用管他們用的什麼版本控制系統。只要你的觀眾可以讀電子郵件，他們就能看到你的修改。類似，

在你這邊，你只需要一個電子郵件帳號：不必搭建一個在綫的 Git 倉庫。

回想一下第一章：

```
$ git diff 1b6d > my.patch
```

輸出是一個補丁，可以粘貼到電子郵件裡用以討論。在一個 Git 倉庫，鍵入：

```
$ git apply < my.patch
```

來打這個補丁。

在更正式些的設置裡，當作者名字以及或許簽名應該記錄下的時候，為過去某一刻生成補丁，鍵入：

```
$ git format-patch 1b6d
```

結果檔案可以給 **git-send-email** 發送，或者手工發送。你也可以指定一個提交範圍：

```
$ git format-patch 1b6d..HEAD^^
```

在接收一端，保存郵件到一個檔案，然後鍵入：

```
$ git am < email.txt
```

這就打了補丁並創建了一個提交，包含諸如作者之類的信息。

使用瀏覽器郵件客戶端，在保存補丁為檔案之前，你可能需要建一個按鈕，看看郵件內容原來的原始形式。

對基于 mbox 的郵件客戶端有些微不同，但如果你在使用的話，你可能是那種能輕易找出答案的那種人，不用讀教程。

對不起，移走了

克隆一個倉庫後，運行 **git push** 或 **git pull** 講自動推到或從原先 URL 拉。Git 如何做這個呢？秘密在和克隆一起創建的配置選項。讓我們看一下：

```
$ git config --list
```

選項 `remote.origin.url` 控制 URL 源；“origin” 是給源倉庫的暱稱。和 “master” 分支的慣例一樣，我們可以改變或刪除這個暱稱，但通常沒有理由這麼做。

如果原先倉庫移走，我們可以更新 URL，通過：

```
$ git config remote.origin.url git://new.url/proj.git
```

選項 `branch.master.merge` 指定 **git pull** 裡的預設遠端分支。在初始克隆的時候，它被設為原倉庫的當前分支，因此即使原倉庫之後挪到一個不同的分支，後來的 pull 也將忠實地跟着原來的分支。

這個選項只使用我們初次克隆的倉庫，它的值記錄在選項 `branch.master.remote` 。如果我們從其他倉庫拉入，我們必須顯示指定我們想要哪個分支：

```
$ git pull git://example.com/other.git master
```

以上也解釋了為什麼我們較早一些 push 和 pull 的例子沒有參數。

遠端分支

當你克隆一個倉庫，你也克隆了它的所有分支。你或許沒有注意到因為 Git 將它們隱藏起來了：你必須明確地要求。這使得遠端倉庫裡的分支不至于干擾你的分支，也使 Git 對初學者稍稍容易些。

列出遠端分支，使用：

```
$ git branch -r
```

你應該看到類似：

```
origin/HEAD
origin/master
origin/experimental
```

這顯示了遠端倉庫的分支和 HEAD，可以用在常用的 Git 命令裡。例如，假設你已經做了很多提交，並希望和最後取到的版本比較一下。你可以搜索適當的 SHA1 哈希值，但使用下面命令更容易些：

```
$ git diff origin/HEAD
```

或你可以看看 "experimental" 分支都有啥：

```
$ git log origin/experimental
```

多遠端

假設另兩個開發在同一個項目上工作，我們希望保持兩個標籤。我們可以同事跟多個倉庫：

```
$ git remote add other git://example.com/some_repo.git
$ git pull other some_branch
```

現在我們已經從第二個倉庫合併到一個分支，並且我們已容易訪問所有倉庫的所有分支。

```
$ git diff origin/experimental^ other/some_branch~5
```

但如果為了不影響自己的工作，我們只想比較他們的變更怎麼辦呢？換句話說，我們想檢查一下他們的分支，又不使他們的變更入侵我們的工作目錄。那不是運行 pull 命令，而是運行：

```
$ git fetch        # Fetch from origin, the default.
$ git fetch other  # Fetch from the second programmer.
```

這只是獲取歷史。儘管工作目錄維持不變，我們可以參考任何倉庫的任何分支，使用一個 Git 命令，因為我們現在有一個本地拷貝。

回想一下，在幕後，一個 pull 是簡單地一個 **fetch** 然後 **merge** 。通常，我們 **pull** 因為我們想在獲取後合併最近提交；這個情況是一個值得注意的例外。

關於如何去除遠端倉庫，如何忽略特定分支等更多，參見 **git help remote** 。

我的喜好

對我手頭的項目，我喜歡貢獻者去準備倉庫，這樣我可以從其中拉。一些 Git 伺服讓你點一個按鈕，擁有自己的分叉項目。

在我獲取一個樹之後，我運行 Git 命令去瀏覽並檢查這些變更，理想情況下這些變更組織良好，描述良好。我合併這些變更，也或許做些編輯。直到滿意，我才把變更推入主資源庫。

儘管我不經常收到貢獻，我相信這個方法擴展性良好。參見 這篇來自 Linus Torvalds 的博客

獸在 Git 的世界裡比補丁檔案稍更方便，因為不用我將補丁轉換到 Git 提交。更進一步，Git 處理諸如作者姓名和信箱地址的細節，還有時間和日期，以及要求作者描述他們的提交。

Git 大師技

到現在，你應該有能力查閱 **git help** 頁，並理解几乎所有東西。然而，查明解決特定問題需要的確切命令可能是乏味的。或許我可以省你點功夫：以下是我過去曾經需要的一些食譜。

源碼發佈

就我的項目而言，Git 完全跟蹤了我想打包並發佈給用戶的檔案。創建一個源碼包，我運行：

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

提交變更

對特定項目而言，告訴 Git 你增加，刪除和重命名了一些檔案很麻煩。而鍵入如下命令會容易的多：

```
$ git add .
$ git add -u
```

Git 將查找當前目錄的檔案並自己算出具體的情況。除了用第二個 add 命令，如果你也打算這時提交，可以運行 ‘git commit -a‘。關於如何指定應被忽略的檔案，參見 **git help ignore** 。

你也可以用一行命令完成以上任務：

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

這裡 **-z** 和 **-0** 選項可以消除包含特殊字元的檔案名引起的不良副作用。注意這個命令也添加應被忽略的檔案，這時你可能需要加上 `-x` 或 `-X` 選項。

我的提交太大了！

是不是忽視提交太久了？痴迷地編碼，直到現在才想起有源碼控制工具這回事？提交一系列不相關的變更，因為那是你的風格？

別擔心，運行：

```
$ git add -p
```

為你做的每次修改，Git 將展示給你變動的代碼，並詢問該變動是否應是下一次提交的一部分。回答“y”或者“n”。也有其他選項，比如延遲決定；鍵入“？”來學習更多。

一旦你滿意，鍵入

```
$ git commit
```

來精確地提交你所選擇的變更（階段變更）。確信你沒加上 **-a** 選項，否則 Git 將提交所有修改。

如果你修改了許多地方的許多檔案怎麼辦？一個一個地查看變更令人沮喪，心態麻木。這種情況下，使用 **git add -i** ，它的界面不是很直觀，但更靈活。敲幾個鍵，你可以一次決定階段或非階段性提交幾個檔案，或查看並只選擇特定檔案的變更。作為另一種選擇，你還可以運行 **git commit --interactive** ，這個命令會在你操作完後自動進行提交。

索引：Git 的中轉區域

當目前為止，我們已經忽略 Git 著名的’ 索引 ‘概念，但現在我們必須面對它，以解釋上面發生的。索引是一個臨時中轉區。Git 很少在你的項目和它的歷史之間直接倒騰數據。通常，Git 先寫數據到索引，然後拷貝索引中的數據到最終目的地。

例如，**commit -a** 實際上是一個兩步過程。第一步把每個追蹤檔案當前狀態的快照放到索引中。第二步永久記錄索引中的快照。沒有 **-a** 的提交只執行第二步，並且只在運行不知何故改變索引的命令才有意義，比如 **git add** 。

通常我們可以忽略索引並假裝從歷史中直接讀並直接寫。在這個情況下，我們希望更好地控制，因此我們操作索引。我們放我們變更的一些的快照到索引中，而不是所有的，然後永久地記錄這個小心操縱的快照。

別丟了你的 HEAD

HEAD 好似一個游標，通常指向最新提交，隨最新提交向前移動。一些 Git 命令讓你來移動它。例如：

```
$ git reset HEAD~3
```

將立即向回移動 HEAD 三個提交。這樣所有 Git 命令都表現得好似你沒有做那最後三個提交，然而你的檔案保持在現在的狀態。具體應用參見幫助頁。

但如何回到將來呢？過去的提交對將來一無所知。

如果你有原先 Head 的 SHA1 值，那麼：

```
$ git reset 1b6d
```

但假設你從來沒有記下呢？別擔心，像這些命令，Git 保存原先的 Head 為一個叫 ORGI_HEAD 的標記，你可以安全體面的返回：

```
$ git reset ORIG_HEAD
```

HEAD 捕獵

或許 ORG_HEAD 不夠。或許你剛認識到你犯了個歷史性的錯誤，你需要回到一個早已忘記分支上一個遠古的提交。

預設，Git 保存一個提交至少兩星期，即使你命令 Git 摧毀該提交所在的分支。難點是找到相應的哈希值。你可以查看在.git/objects 裡所有的哈希值並嘗試找到你期望的。但有一個更容易的辦法。

Git 把算出的提交哈希值記錄在 ".git/logs"。這個子目錄引用包括所有分支上所有活動的歷史，同時檔案 HEAD 顯示它曾經有過的所有哈希值。後者可用來發現分支上一些不小心丟掉提交的哈希值。

The reflog command provides a friendly interface to these log files. Try

命令 reflog 為訪問這些日誌檔案提供友好的介面，試試

```
$ git reflog
```

而不是從 reflog 拷貝粘貼哈希值，試一下:

```
$ git checkout "@{10 minutes ago}"
```

或者撿出後五次訪問過的提交，通過:

```
$ git checkout "@{5}"
```

更多內容參見 **git help rev-parse** 的 "Specifying Revisions" 部分。

你或許期望去為已刪除的提交設置一個更長的保存周期。例如:

```
$ git config gc.pruneexpire "30 days"
```

意思是一個被刪除的提交會在刪除 30 天後，且運行 **git gc** 以後，被永久丟棄。

你或許還想關掉 **git gc** 的自動運行:

```
$ git config gc.auto 0
```

在這種情況下提交將只在你手工運行 **git gc** 的情況下才永久刪除。

基于 Git 構建

依照真正的 UNIX 風格設計，Git 允許其易於用作其他程序的底層組件，比如圖形界面，Web 界面，可選擇的命令行界面，補丁管理工具，導入和轉換工具等等。實際上，一些 Git 命令它們自己就是站在巨人肩膀上的腳本。通過一點修補，你可以定製 Git 適應你的偏好。

一個簡單的技巧是，用 Git 內建 alias 命令來縮短你最常使用命令:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # 顯示當前別名
alias.co checkout
$ git co foo                              # 和“git checkout foo”一樣
```

另一個技巧，在提示符或窗口標題上打印當前分支。調用：

```
$ git symbolic-ref HEAD
```

顯示當前分支名。在實際應用中，你可能最想去掉 “refs/heads/” 並忽略錯誤：

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

子目錄 contrib 是一個基于 Git 工具的寶庫。它們中的一些時時會被提升為官方命令。在 Debian 和 Ubuntu，這個目錄位於 /usr/share/doc/git-core/contrib 。

一個受歡迎的居民是 workdir/git-new-workdir 。通過聰明的符號連結，這個腳本創建一個新的工作目錄，其歷史與原來的倉庫共享：

```
$ git-new-workdir an/existing/repo new/directory
```

這個新的目錄和其中的檔案可被視為一個克隆，除了既然歷史是共享的，兩者的樹自動保持同步。不必合併，推入或拉出。

大膽的特技

這些天，Git 使得用戶意外摧毀數據變得更困難。但如若你知道你在做什麼，你可以突破為通用命令所設的防衛保障。

Checkout：未提交的變更會導致撿出失敗。銷毀你的變更，並無論如何都 checkout 一個指定的提交，使用強制標記：

```
$ git checkout -f HEAD^
```

另外，如果你為撿出指定特別路徑，那就沒有安全檢查了。提供的路徑將被不加提示地覆蓋。如你使用這種方式的檢出，要小心。

Reset：如有未提交變更重置也會失敗。強制其通過，運行：

```
$ git reset --hard 1b6d
```

Branch：引起變更丟失的分支刪除會失敗。強制刪除，鍵入：

```
$ git branch -D dead_branch  # instead of -d
```

類似，通過移動試圖覆蓋分支，如果隨之而來有數據丟失，也會失敗。強制移動分支，鍵入：

```
$ git branch -M source target  # 而不是 -m
```

不像 checkout 和重置，這兩個命令延遲數據銷毀。這個變更仍然存儲在.git 的子目錄裡，並且可以通過恢復.git/logs 裡的相應哈希值獲取（參見上面上面 “HEAD 獵捕”）。默認情況下，這些數據會保存至少兩星期。

Clean: 一些 Git 命令拒絶執行，因為它們擔心會重裝未納入管理的檔案。如果你確信所有未納入管理的檔案都是消耗，那就無情地刪除它們，使用：

```
$ git clean -f -d
```

下次，那個討厭的命令就會工作！

阻止壞提交

愚蠢的錯誤污染我的倉庫。最可怕的是由於忘記 **git add** 而引起的檔案丟失。較小的罪過是行末追加空格並引起合併衝突：儘管危害少，我希望浙西永遠不要出現在公開記錄裡。

不過我購買了傻瓜保險，通過使用一個 __ 鉤子 __ 來提醒我這些問題：

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # 對舊版本Git，先運行chmod +x
```

現在 Git 放棄提交，如果檢測到無用的空格或未解決的合併衝突。

對本文檔，我最終添加以下到 **pre-commit** 鉤子的前面，來防止缺魂兒的事：

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

幾個 git 操作支持鉤子；參見 **git help hooks** 。我們早先激活了作為例子的 **post-update** 鉤子，當討論基于 HTTP 的 Git 的時候。無論 head 何時移動，這個鉤子都會運行。例子腳本 post-update 更新 Git 在基于 Git 並不知曉的傳輸協議，諸如 HTTP，通訊時所需的檔案。

揭開面紗

我們揭開 Git 神秘面紗，往裡瞧瞧它是如何創造奇蹟的。我會跳過細節。更深入的描述參見 用戶手冊。

大象無形

Git 怎麼這麼謙遜寡言呢？除了偶爾提交和合併外，你可以如常工作，就像不知道版本控制系統存在一樣。那就是，直到你需要它的時候，而且那是你歡欣的時候，Git 一直默默注視着你。

其他版本控制系統強迫你與繁文縟節和官僚主義不斷鬥爭。檔案的權限可能是隻讀的，除非你顯式地告訴中心伺服器哪些檔案你打算編輯。即使最基本的命令，隨着用戶數目的增多，也會慢的像爬一樣。中心伺服器可能正跟蹤什麼人，什麼時候 check out 了什麼代碼。當網絡連接斷了的時候，你就遭殃了。開發人員不斷地與這些版本控制系統的種種限製作鬥爭。一旦網絡或中心伺服器癱瘓，工作就嘎然而止。

與之相反，Git 簡單地在你工作目錄下的.git`目錄保存你項目的歷史。這是你自己的歷史拷貝，因此你可以保持離線，直到你想和他人溝通為止。你擁有你的檔案命運完全的控制權，因為 Git 可以輕易在任何時候從.git‘重建一個保存狀態。

數據完整性

很多人把加密和保持信息機密關聯起來，但一個同等重要的目標是保證信息安全。合理使用哈希加密功能可以防止無意或有意的數據損壞行為。

一個 SHA1 哈希值可被認為是一個唯一的 160 位 ID 數，用它可以唯一標識你一生中遇到的每個位元組串。實際上不止如此：每個位元組串可供任何人用好多輩子。

對一個檔案而言，其整體內容的哈希值可以被看作這個檔案的唯一標識 ID 數。

因為一個 SHA1 哈希值本身也是一個位元組串，我們可以哈希包括其他哈希值的位元組串。這個簡單的觀察出奇地有用：查看 “哈希鏈”。我們之後會看 Git 如何利用這一點來高效地保證數據完整性。

簡言之，Git 把你數據保存在 ‘.git/objects‘子目錄，那裡看不到正常檔案名，相反你只看到 ID。通過用 ID 作為檔案名，加上一些檔案鎖和時間戳技巧，Git 把任意一個原始的文件系統轉化為一個高效而穩定的資料庫。

智能

Git 是如何知道你重命名了一個檔案，即使你從來沒有明確提及這個事實？當然，你或許是運行了 **git mv** ，但這個命令和 **git add** 緊接 **git rm** 是完全一樣的。

Git 啟發式地找出相連版本之間的重命名和拷貝。實際上，它能檢測檔案之間代碼塊的移動或拷貝！儘管它不能覆蓋所有的情況，但它已經做的很好了，並且這個功能也總在改進中。如果它在你那兒不工作的話，可以嘗試打開開銷更高的拷貝檢測選項，並考慮升級。

索引

為每個加入管理的檔案，Git 在一個名為 “index” 的檔案裡記錄統計信息，諸如大小，創建時間和最後修改時間。為了確定檔案是否更改，Git 比較其當前統計信息與那些在索引裡的統計信息。如果一致，那 Git 就跳過重新讀檔案。

因為統計信息的調用比讀檔案內容快的很多，如果你僅僅編輯了少數幾個檔案，Git 几乎不需要什麼時間就能更新他們的統計信息。

我們前面講過索引是一個中轉區。為什麼一堆檔案的統計數據是一個中轉區？因為添加命令將檔案放到 Git 的資料庫並更新它們的統計信息，而無參數的提交命令創建一個提交，只基于這些統計信息和已經在資料庫裡的檔案。

Git 的源起

這個 Linux 內核郵件列表帖子 描述了導致 Git 的一系列事件。整個討論線索是一個令人着迷的歷史探究過程，對 Git 史學家而言。

對象資料庫

你數據的每個版本都保存在“對象資料庫”裡，其位於子目錄.git/objects`；其他位于.git/‘的較少數據：索引，分支名，標籤，配置選項，日誌，頭提交的當前位置等。對象資料庫樸素而優雅，是 Git 的力量之源。

‘.git/objects‘裡的每個檔案是一個對象。有 3 中對象跟我們有關：“blob” 對象，“tree” 對象，和 “commit” 對象。

Blob 對象

首先來一個小把戲。去一個檔案名，任意檔案名。在一個空目錄：

```
$ echo sweet > YOUR_FILENAME
$ git init
$ git add .
$ find .git/objects -type f
```

你將看到 .git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d 。

我如何在不知道檔案名的情況下知道這個？這是因為以下內容的 SHA1 哈希值：

```
"blob" SP "6" NUL "sweet" LF
```

是 aa823728ea7d592acc69b36875a482cdf3fd5c8d，這裡 SP 是一個空格，NUL 是一個 0 位元組，LF 是一個換行符。你可以驗證這一點，鍵入：

```
$ printf "blob 6\000sweet\n" | sha1sum
```

Git 基于 “內容定址”：檔案並不按它們的檔案名存儲，而是按它們包含內容的哈希值，在一個叫 “blob 對象” 的檔案裡。我們可以把檔案內容的哈希值看作一個唯一 ID，這樣在某種意義上我們通過他們內容放置檔案。開始的 “blob 6” 只是一個包含對象類型與其長度的頭；它簡化了內部存儲。

這樣我可以輕易語言你所看到的。檔案名是無關的：只有裡面的內容被用作構建 blob 對象。

你可能想知道對相同的檔案什麼會發生。試圖加一個你檔案的拷貝，什麼檔案名都行。在 .git/objects 的內容保持不變，不管你加了多少。Git 只存儲一次數據。

順便說一句，在 .git/objects 裡的檔案用 zlib 壓縮，因此你不應該直接查看他們。可以通過zpipe -d 管道，或者鍵入：

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

這漂亮地打印出給定的對象。

Tree 對象

但檔案名在哪？它們必定在某個階段保存在某個地方。Git 在提交時得到檔案名：

```
$ git commit  # 輸入一些信息。
$ find .git/objects -type f
```

你應看到 3 個對象。這次我不能告訴你這兩個新檔案是什麼，因為它部分依賴你選擇的文件名。我繼續進行，假設你選了 “rose”。如果你沒有，你可以重寫歷史以讓它看起來像似你做了：

```
$ git filter-branch --tree-filter 'mv YOUR_FILENAME rose'
$ find .git/objects -type f
```

現在你硬看到檔案 .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9，因為這是以下內容的 SHA1 哈希值：

```
"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d
```

檢查這個檔案真的包含上面內容通過鍵入：

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

使用 zpipe，驗證哈希值是容易的：

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

與查看檔案相比，哈希值驗證更技巧一些，因為其輸出不止包含原始未壓縮檔案。

這個檔案是一個 “tree” 對象：一組數據包含檔案類型，檔案名和哈希值。在我們的例子裡，檔案類型是 100644，這意味着 “rose” 是一個一般檔案，並且哈希值指 blob 對象，包含 “rose” 的內容。其他可能檔案類型有可執行，連結或者目錄。在最後一個例子裡，哈希值指向一個 tree 對象。

在一些過渡性的分支，你會有一些你不在需要的老的對象，儘管有寬限過期之後，它們會被自動清除，現在我們還是將其刪除，以使我們比較容易跟上這個玩具例子。

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

在真實項目裡你通常應該避免像這樣的命令，因為你在破換備份。如果你期望一個乾淨的倉庫，通常最好做一個新的克隆。還有，直接操作 .git 時一定要小心：如果 Git 命令同時也在運行會怎樣，或者突然停電？一般，引用應由 **git update-ref -d** 刪除，儘管通常手工刪除 refs/original 也是安全的。

Commit 對象

我們已經解釋了三個對象中的兩個。第三個是 “commit” 對象。其內容依賴于提交信息以及其創建的日期和時間。為滿足這裡我們所有的，我們不得不調整一下：

```
$ git commit --amend -m Shakespeare  # 改提交信息
$ git filter-branch --env-filter 'export
```

```
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"' # Rig timestamps and authors.
$ find .git/objects -type f
```

你現在應看到 .git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187 是下列內容的 SHA1 哈希值:

```
"commit 158" NUL
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

和前面一樣，你可以運行 zpipe 或者 cat-file 來自己看。

這是第一個提交，因此沒有父提交，但之後的提交將總有至少一行，指定一個父提交。

沒那麼神

Git 的秘密似乎太簡單。看起來似乎你可以整合幾個 shell 腳本，加幾行 C 代碼來弄起來，也就幾個小時的事：一個基本檔案操作和 SHA1 哈希化的混雜，用鎖檔案裝飾一下，檔案同步保證健壯性。實際上，這準確描述了 Git 的最早期版本。儘管如此，除了巧妙地打包以節省空間，巧妙地索引以省時間，我們現在知道 Git 如何靈巧地改造檔案系統成為一個對版本控制完美的資料庫。

例如，如果對象資料庫裡的任何一個檔案由於硬碟錯誤損毀，那麼其哈希值將不再匹配，這個錯誤會報告給我們。通過哈希化其他對象的哈希值，我們在所有層面維護數據完整性。Commit 對象是原子的，也就是說，一個提交永遠不會部分地記錄變更：在我們已經存儲所有相關 tree 對象，blob 對象和父 commit 對象之後，我們才可以計算提交的的哈希值並將其存儲在資料庫，對象資料庫不受諸如停電之類的意外中斷影響。

我們打敗即使是最狡猾的對手。假設有誰試圖悄悄修改一個項目裡一個遠古版本檔案的內容。為使對象據庫看起來健康，他們也必須修改相應 blob 對象的哈希值，既然它現在是一個不同的位元組串。這意味着他們講不得不引用這個檔案的 tree 對象的哈希值，並反過來改變所有與這個 tree 相關的 commit 對象的哈希值，還要加上這些提交所有後裔的哈希值。這暗示官方 head 的哈希值與這個壞倉庫不同。通過跟蹤不匹配哈希值線索，我們可以查明殘缺檔案，以及第一個被破壞的提交。

總之，只要 20 個位元組代表最後一次提交的是安全的，不可能篡改一個 Git 倉庫。

那麼 Git 的著名功能怎樣呢？分支？合併？標籤？單純的細節。當前 head 保存在檔案 .git/HEAD，其中包含了一個 commit 對象的哈希值。該哈希值在運行提交以及其他命令是更新。分支几乎一樣：它們是保存在 .git/refs/heads 的檔案。標籤也是：它們住在住在 .git/refs/tags，但它們由一套不同的命令更新。

附錄 A: Git 的缺點

有一些 Git 的問題，我已經藏在毯子下面了。有些可以通過腳本或回調方法輕易地解決，有些需要重組或重定義項目，少數剩下的煩惱，還只能等待。或者更好地，投入進來幫忙。

SHA1 的弱點

隨着時間的推移，密碼學家發現越來越多的 SHA1 的弱點。已經發現對對資源雄厚的組織哈希衝撞是可能的。在幾年內，或許甚至一個一般的 PC 也將有足夠計算能力悄悄摧毀一個 Git 倉庫。

希望在進一步研究摧毀 SHA1 之前，Git 能遷移到一個更好的哈希算法。

微軟 Windows

Git 在微軟 Windows 上可能有些繁瑣：

- Cygwin，, 一個 Windows 下的類 Linux 的環境，包含一個 一個 Git 在 Windows 下的移植.
- 基于 MSys 的 Git 是另一個，要求最小運行時支持，不過一些命令不能馬上工作。

不相關的檔案

如果你的項目非常大，包含很多不相關的檔案，而且正在不斷改變，Git 可能比其他系統更不管用，因為獨立的檔案是不被跟蹤的。Git 跟蹤整個項目的變更，這通常才是有益的。

一個方案是將你的項目拆成小塊，每個都由相關檔案組成。如果你仍然希望在同一個資源庫裡保存所有內容的話，可以使用 **git submodule** 。

誰在編輯什麼？

一些版本控制系統在編輯前強迫你顯示地用某個方法標記一個檔案。儘管這種要求很煩人，尤其是需要和中心伺服器通訊時，不過它還是有以下兩個好處的：

1. 比較速度快，因為只有被標記的檔案需要檢查。
2. 可以知道誰在這個檔案上工作，通過查詢在中心伺服器誰把這個檔案標記為編輯狀態。

使用適當的腳本，你也可以使 Git 達到同樣的效果。這要求程序員協同工作，當他編輯一個檔案的時候還要運行特定的腳本。

檔案歷史

因為 Git 記錄的是項目範圍的變更，重造單一檔案的變更歷史比其他跟蹤單一檔案的版本控制系統要稍微麻煩些。

好在麻煩還不大，也是值得的，因為 Git 其他的操作難以置信地高效。例如，‘git checkout‘比‘cp -a‘都快，而且項目範圍的 delta 壓縮也比基于檔案的 delta 集合的做法好多了。

初始克隆

The initial cost is worth paying in the long run, as most future operations will then be fast and offline. However, in some situations, it may be preferable to create a shallow clone with the `--depth` option. This is much faster, but the resulting clone has reduced functionality.

當一個項目歷史很長後，與在其他版本系統裡的檢出代碼相比，創建一個克隆的開銷會大的多。

長遠來看，開始付出的代價還是值得付出的，因為大多將來的操作將由此變得很快，並可以離線完成。然而，在一些情況下，使用 ‘--depth‘創建一個淺克隆比較划算些。這種克隆初始化的更快，但得到克隆的功能有所削減。

不穩定的項目

變更的大小決定寫入的速度快慢是 Git 的設計。一般人做了小的改動就會提交新版本。這裡一行臭蟲修改，那裡一個新功能，修改掉的註釋等等。但如果你的檔案在相鄰版本之間存在極大的差異，那每次提交時，你的歷史記錄會以整個項目的大小增長。

任何版本控制系統對此都束手無策，但標準的 Git 用戶將遭受更多，因為一般來說，歷史記錄也會被克隆。

應該檢查一下變更巨大的原因。或許檔案格式需要改變一下。小修改應該僅僅導致幾個檔案的細小改動。

或許，資料庫或備份/打包方案才是正選，而不是版本控制系統。例如，版本控制就不適宜用來管理網絡攝像頭周期性拍下的照片。

如果這些檔案實在需要不斷更改，他們實在需要版本控制，一個可能的辦法是以中心的方式使用 Git。可以創建淺克隆，這樣檢出的較少，也沒有項目的歷史記錄。當然，很多 Git 工具就不能用了，並且修復必須以補丁的形式提交。這也許還不錯，因為似乎沒人需要大幅度變化的不穩定檔案歷史。

另一個例子是基于韌體的項目，使用巨大的二進制檔案形式。用戶對韌體檔案的變化歷史沒有興趣，更新的壓縮比很低，因此韌體修訂將使倉庫無謂的變大。

這種情況，源碼應該保存在一個 Git 倉庫裡，二進制檔案應該單獨保存。為了簡化問題，應該發佈一個腳本，使用 Git 克隆源碼，對韌體只做同步或 Git 淺克隆。

全局計數器

一些中心版本控制系統維護一個正整數，當一個新提交被接受的時候這個整數就增長。Git 則是通過哈希值來記錄所有變更，這在大多數情況下都工作的不錯。

但一些人喜歡使用整數的方法。幸運的是，很容易就可以寫個腳本，這樣每次更新，中心 Git 倉庫就增大這個整數，或使用 tag 的方式，把最新提交的哈希值與這個整數關聯起來。

每個克隆都可以維護這麼個計數器，但這或許沒什麼用，因為只有中心倉庫以及它的計數器對每個人才有意義。

空子目錄

空子目錄不可加入管理。可以通過創建一個空檔案以繞過這個問題。

Git 的當前實現，而不是它的設計，是造成這個缺陷的原因。如果運氣好，一旦 Git 得到更多關注，更多用戶要求這個功能，這個功能就會被實現。

初始提交

傳統的計算機系統從 0 計數，而不是 1。不幸的是，關於提交，Git 並不遵從這一約定。很多命令在初始提交之前都不友好。另外，一些極少數的情況必須作特別地處理。例如重訂一個使用不同初始提交的分支。

Git 將從定義零提交中受益：一旦一個倉庫被創建起來，HEAD 將被設為包含 20 個零位元組的字元串。這個特別的提交代表一棵空的樹，沒有父節點，早于所有 Git 倉庫。

然後運行 git log，比如，通知用戶至今還沒有提交過變更，而不是報告致命錯誤並退出。這與其他工具類似。

每個初始提交都隱式地成為這個零提交的後代。

不幸的是還有更糟糕的情況。如果把幾個具有不同初始提交的分支合併到一起，之後的重新修訂不可避免的需要人員的介入。

介面怪癖

對提交 A 和提交 B，表達式 “A..B” 和 “A...B” 的含義，取決於命令期望兩個終點還是一個範圍。參見 **git help diff** 和 **git help rev-parse** 。

附錄 B: 本指南的翻譯

我推薦如下步驟翻譯本指南，這樣我的腳本就可以快速生成 HTML 和 PDF 版本，並且所有翻譯也可以共存於同一個倉庫。

克隆源碼，然後針對不同目標 語言的 IETF tag創建一個目錄。參見 W3C 在國際化方面的文章。例如，英語是 “en”，日語是 “ja”，正體中文是 “zh-Hant”。然後在新建目錄，翻譯這些來自 “en” 目錄的 `txt` 檔案。

例如，將本指南譯為 克林貢語，你也許鍵入：

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" 是克林貢語的IETF語言碼。
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # 翻譯這個檔案
```

對每個檔案都一樣。

打開 Makefile 檔案，把語言碼加入 ‘TRANSLATIONS‘變數，現在你可以時不時查看你的工作：

```
$ make tlh
$ firefox book-tlh/index.html
```

經常提交你的變更，然後然我知道他們什麼時候完成。GitHub.com 提供一個便于 fork “gitmatic” 項目的界面，提交你的變更，然後告訴我去合併。

但請按照最適合你的方式做：例如，中文譯者就使用 Google Docs。只要你的工作使更多人看到我的工作，我就高興。