

THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

*EVERYTHING YOU NEED TO
KNOW ABOUT GIT*

Scott Chacon and Ben Straub

Apress®

Pro Git

Scott Chacon, Ben Straub

版本 2.1.0, 2018-02-24

目錄

授權條款	1
Scott Chacon 的作者序	2
Ben Straub 的作者序	3
銘謝	4
簡介	5
開始	7
關於版本控制	7
Git 的簡史	10
Git 基礎要點	11
命令列	14
Git 安裝教學	14
初次設定 Git	16
取得說明文件	18
摘要	19
Git 基礎	20
取得一個 Git 倉儲	20
紀錄變更到版本庫中	21
檢視提交的歷史記錄	33
復原	39
與遠端協同工作	42
標籤	47
Git Aliases	51
總結	52
使用 Git 分支	53
簡述分支	53
分支和合併的基本用法	59
分支管理	68
分支工作流程	69
遠端分支	72
衍合	82
總結	90
伺服器上的 Git	91
通訊協定	91
在伺服器上佈署 Git	95
產生你的 SSH 公鑰	97
設定伺服器	99
Git 常駐程式	101
Smart HTTP	102
GitWeb	104
GitLab	105
第3方 Git 託管方案	109
總結	110
分散式的 Git	111
分散式工作流程	111
對專案進行貢獻	114
維護一個專案	135
Summary	149
GitHub	150
建立帳戶及設定	150

參與一個專案	154
維護專案	172
Managing an organization	184
Scripting GitHub	188
總結	197
Git Tools	198
Revision Selection	198
Interactive Staging	205
Stashing and Cleaning	209
Signing Your Work	215
Searching	219
Rewriting History	223
Reset Demystified	229
Advanced Merging	249
Rerere	266
Debugging with Git	272
Submodules	275
Bundling	294
Replace	298
Credential Storage	306
Summary	311
Customizing Git	312
Git Configuration	312
Git Attributes	322
Git Hooks	330
An Example Git-Enforced Policy	333
Summary	342
Git and Other Systems	343
Git as a Client	343
Migrating to Git	387
Summary	402
Git Internals	403
Plumbing and Porcelain	403
Git Objects	404
Git References	413
Packfiles	418
The Refspec	421
Transfer Protocols	423
Maintenance and Data Recovery	428
Environment Variables	435
Summary	441
附錄 A: Git in Other Environments	442
Graphical Interfaces	442
Git in Visual Studio	447
Git in Eclipse	448
Git in Bash	449
Git in Zsh	450
Git in Powershell	452
Summary	453
附錄 B: Embedding Git in your Applications	454
Command-line Git	454

Libgit2	454
JGit	459
附錄 C: Git Commands	464
Setup and Config	464
Getting and Creating Projects	465
Basic Snapshotting	465
Branching and Merging	468
Sharing and Updating Projects	470
Inspection and Comparison	472
Debugging	472
Patching	473
Email	474
External Systems	475
Administration	475
Plumbing Commands	476
Index	477

授權條款

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Scott Chacon 的作者序

歡迎來到 Pro Git 的第二版。第一版發行至今已經超過 4 年了。從那個時候到現在，有很多東西已經改變了，但也有很多重要的事情沒有改變。獲益於 Git 核心團隊令人吃驚的向下相容程度，所以大多數核心命令和概念至今仍然有效。與此同時，在 Git 的相關社群中，也有一些重大的新增和改變。這本書第二版是為了處理這些改變並更新這本書，使得這本書可以對新使用者更有幫助。

當我寫第一版時，Git 是個相當難用而且只適合重度使用者使用的工具。它在當時的社群中使用率開始蒸蒸日上，但還沒像今天一樣無所不在。至今，幾乎所有的開源社群都已經採用它。Git 在 Windows、各種平台上的圖形化介面、對 IDE 的支援、和商業使用上都已經有了難以置信的進展。四年前的 Pro Git 這本書根本無法預知到這些事。這個新版本主要的目標之一就是要接觸這些在 Git 社群中的最新的東西。

有在使用 Git 的開放原始碼社群也暴增了。當我在大約五年前坐下來寫這本書時（第一版花了我一些時間寫出來），我開始在一家發展網頁版 Git 託管服務且小有名氣的公司工作，名為 GitHub。在第一版發佈時，大概只有幾千人在使用 GitHub，而且這個公司只有 4 個員工。當我現在在寫這個序的時候，GitHub 第一千萬個被託管的專案已經產生，還有將近 500 萬個註冊的開發者帳戶和 230 個員工。不論是非對錯，GitHub 用了我在第一版時難以想像的方式，劇烈地改變了大片的開源社群。

我在 Pro Git 的第一版中寫了一個用 GitHub 來託管 Git 的範例，這是一個我不是很喜歡的部分。我不是很喜歡在這個我完全是寫給社群的資源中提到我的公司。到了現在第二版，我仍然很不希望這樣，但 GitHub 在 Git 社群中的重要性卻已經是無法避而不談。所以除了這個 Git 託管的範例，我已經決定把書的那部分變成更有深度地去描述何謂 GitHub 和如何有效的使用它。如果你要學習如何使用 Git，那麼知道如何使用 GitHub 將會幫助你融入這個龐大的社群，而且無論你要把程式碼託管在哪裡，都是有足夠參考價值的。

從第一版到現在還有個重要的改變，就是在 Git 傳輸資料的方法新增並大量使用 HTTP 協定。因為 HTTP 的簡便性，在書中大部分的例子已經從 SSH 改用 HTTP。

看到 Git 近年來的發展，從一個相對模糊的版本控制系統變成一個基本上主導了開放原始碼和商業的版本控制系統，這真的很令人驚豔。我很開心 Pro Git 這本書能表現得這麼好，它是世上既成功又完全開放原始碼的資訊書籍之一。

我希望你能享受這個新版的 Pro Git。

Ben Straub 的作者序

這本書的第一版就讓我迷上了 Git，讓我為你介紹一個用更自然且我前所未見的方式去製作軟體。我作為一個軟體開發者已經數年了，但這方式讓我踏入一條比以前更嶄新有趣的路。

經過幾年後的現在，我是一個主要 Git 實作的貢獻者，我已經在最大的 Git 記管服務公司工作，然後我環遊世界去教導人們 Git。當 Scott 詢問我是否對於撰寫第二版有興趣時，我甚至完全不需要思考。

非常高興和榮幸能參與這本書的撰寫。我希望它能幫助你，就如同它對我的幫助一樣多。

銘謝

『給我的妻子，Becky，缺少了她這次的冒險永遠不會開始。』—— Ben。

『這個版本獻給我的女孩們。給這些年一直照顧我的妻子 Jessica，和當我老了，老到無法搞清楚發生什麼事的時候，將會照顧我的女兒 Josephine』—— Scott。

簡介

你將花費你生命中若干個小時的時間來閱讀這本有關 Git 的書，讓我們先用幾分鐘的時間來介紹一下我們為你準備的東西。以下是本書十個主要章節以及三篇附錄的大綱。

在「第一章」我們將介紹版本控制系統（VCSs）和 Git 的基本概念：不涉及技術內容，僅介紹什麼是 Git，為什麼它會成為 VCSs 大家庭中的一員，它與其它 VCSs 的區別，以及為什麼那麼多人都在使用 Git。然後，如果你的系統還沒有安裝 Git，我們將介紹如何下載 Git 以及如何進行設定。

在「第二章」，我們將闡述 Git 的基本使用：如何在您可能遇到的 80% 情況中使用 Git。在閱讀這個章節以後，您應該就會克隆（clone）倉庫、查看專案歷史、修改文件和貢獻一些修改。假設這本書在你看完這個章節後就毀損，那麼到這裡為止的知識也足夠你運用到你重新再去買一本的時候。

「第三章」關注於 Git 的分支模型。分支模型通常被稱為 Git 的殺手鐗。在此你將體會到底是什麼東西讓 Git 如此地與眾不同。當你學習完本章後，你可能會覺得需要一段時間來思考：在沒有 Git 分支的日子裡，到底是如何生活的。

「第四章」將會關注於伺服器端的 Git。這一章是給那些想要在您的組織內或你自己個人的協作伺服器上面設置 Git 的人。如果你希望採用別人維護的伺服器，在此也會提供許多託管的選擇。

「第五章」將帶你看過各種分散式工作流程的完整細節，以及如何使用 Git 來完成這些流程。當你學完這個章節後，你應該能夠熟練地使用多個遠端版本庫、透過電子郵件使用 Git、巧妙地兼顧眾多遠端分支並貢獻補綴。

「第六章」將談到 GitHub 記管服務以及更進階的工具。我們將談到註冊與帳號管理，創建和使用 Git 版本庫，貢獻到專案的共通工作流程以及接受他人貢獻到你的專案，GitHub 的程式設計介面（programmatic interface）和那些能夠讓您的生活變得更輕鬆的小技巧。

「第七章」關於 Git 的進階命令。您將學習到一些進階主題，諸如掌握可怕的 `reset` 命令，使用二分搜尋來找出錯誤，修改 歷史，更細微的版本選擇…等等。本章的介紹將豐富您的 Git 知識，讓您成為一個真正的大師。

「第八章」是關於如何設定你自己的 Git 環境。這個章節包括設定掛鉤程式來執行或貫徹自訂的策略；以及如何使用環境設定來讓你可以用你喜歡的方式做事。我們還會提到如何建立自訂的程式來執行一個自訂的提交策略。

「第九章」是關於 Git 如何應對其他 VCSs 的能力。這章節將會提到如何在 Subversion（SVN）的世界使用 Git 以及如何從其他 VCSs 轉換為 Git。很多組織仍然在使用 SVN，而且並不想要改用 Git，在此你將學到 Git 不可思議的魔力——如果你不得不使用 SVN 伺服器，這個章節將告訴你怎麼做。這個章節還會提到如何從不同系統匯入專案，以便你能夠全心全力投入 Git 的懷抱。

「第十章」將深入 Git 神秘、漂亮的實現細節。現在，您已經知道所有有關 Git 的知識，並且能夠把 Git 用的強大、優雅。接下來，您可以繼續學習 Git 如何存儲對象、Git 的對象模型是怎樣的、打包文件的細節、服務器協議…等更多知識。在這整本書中，我們會在適當的地方引用本章節的內容，以便您可以深入理解某部分的實作細節。如果你和我們一樣想要深入理解 Git 全部的實作細節，您也可以先閱讀第十章。我們將選擇權交給您。

在「附錄 A」我們將會看到在各種特定環境中使用 Git 的範例。我們將會涵蓋一些您可能會想用 Git 的不同 GUIs 和 IDE 程式環境。如果你想在 shell、Visual Studio 或 Eclipse 中使用 Git，那就看看這章的內容吧。

在「附錄 B」我們將探索透過類似 libgit2 和 JGit 等工具來對 Git 編寫腳本或擴充。如果你對寫複雜且快速的自訂工具感興趣，而且想要了解 Git 的底層存取，那麼這章就是你需要的。

最後在「附錄 C」，我們會一次看過所有 Git 的主要命令，複習在本書中介紹的內容，回憶我們能夠使用這些命令做什麼。如果您需要知道本書中我們使用了哪些 Git 命令，您可以在這裡查閱。

現在，讓我們開始吧。

開始

本章將介紹 Git 的相關知識。首先將從版本控制工具的背景知識開始，接著是如何在你的系統上運作 Git，而最後則是如何設定它。閱讀完本章後你應該可以了解為什麼 Git 如此流行、為何你應該使用它，並完成準備工作。

關於版本控制

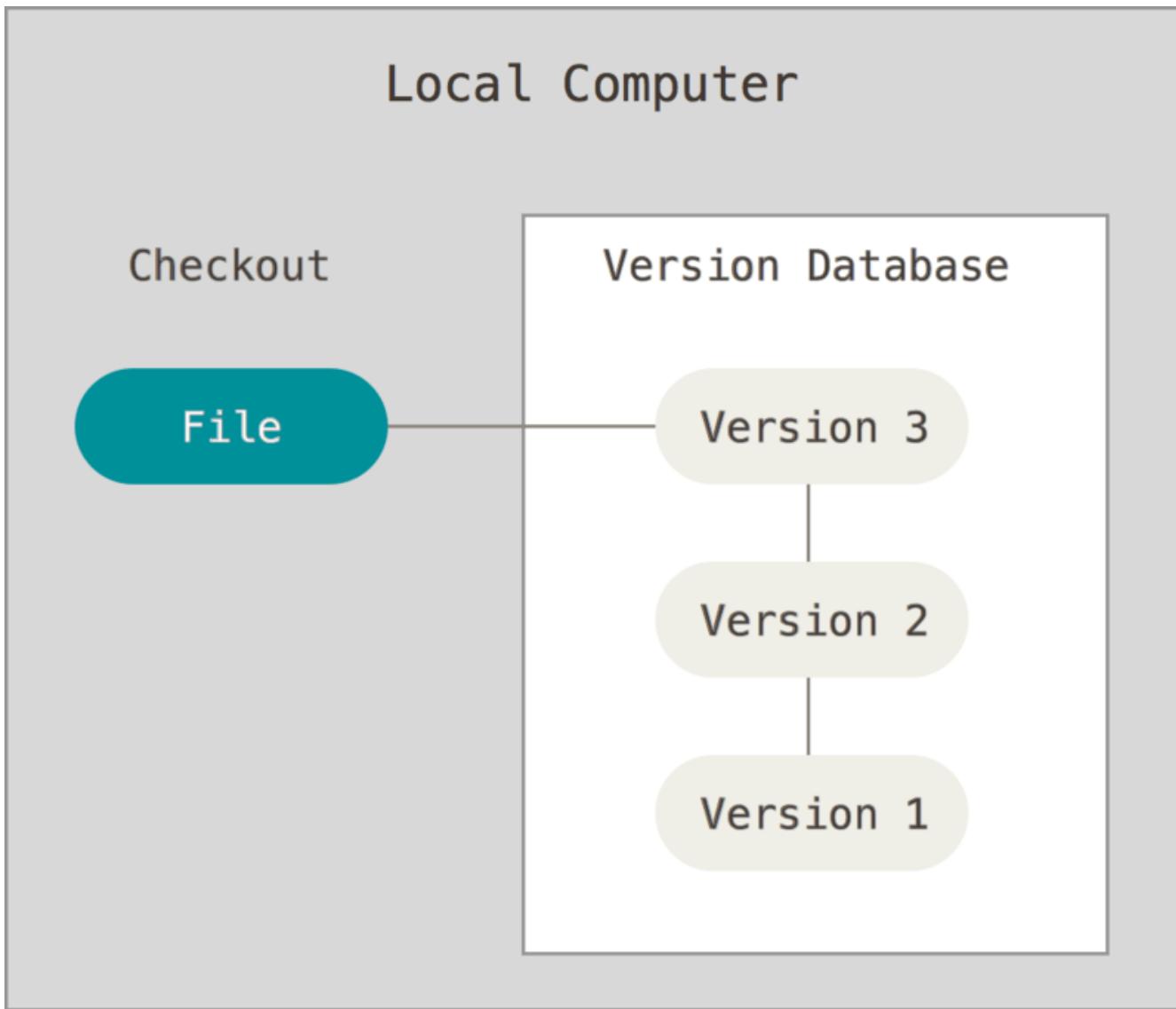
什麼是「版本控制」？我為什麼要關心它呢？版本控制是一種記錄一個或若干文件內容變化，以便將來查閱特定版本修訂情況的系統。在本書的範例中，你將會學到如何對軟體的原始碼做版本控制。當然，你實際上可以對電腦上任意型態的檔案做版本控制。

如果您是美術設計或是網頁設計師，你可能會想要記錄每一次對影像或版面配置的修改（這也通常是你最想要的功能），採用版本控制系統（VCS）就是明智之選。它允許你將檔案復原到之前的狀態、將整個專案復原到先前的狀態、比對某一段時間的修改、查看最後是誰在哪個時間點做了錯誤的修改導致問題發生，誰在何時提出了某個功能缺陷……等。使用版本控制系統一般也意味著如果你做了一些傻事或者遺失檔案，你能很容易地恢復到原先的樣子，但額外增加的工作量卻微乎其微。

本地端版本控制

很多人作版本控制的方法是把檔案複製到另一個目錄（如果他們夠聰明的話，他們還會幫資料夾加上時間）。這種做法很常見，因為這樣做很簡單，但是卻也非常容易產生離譜的錯誤。這種做法非常容易搞混資料夾，意外寫錯檔案或複製覆蓋到不想要的檔案。

為了解決這個問題，程式設計師很久以前就開發了很多種本機版本控制系統，大多都是採用某種簡單的資料庫來紀錄檔案的所有變更記錄。

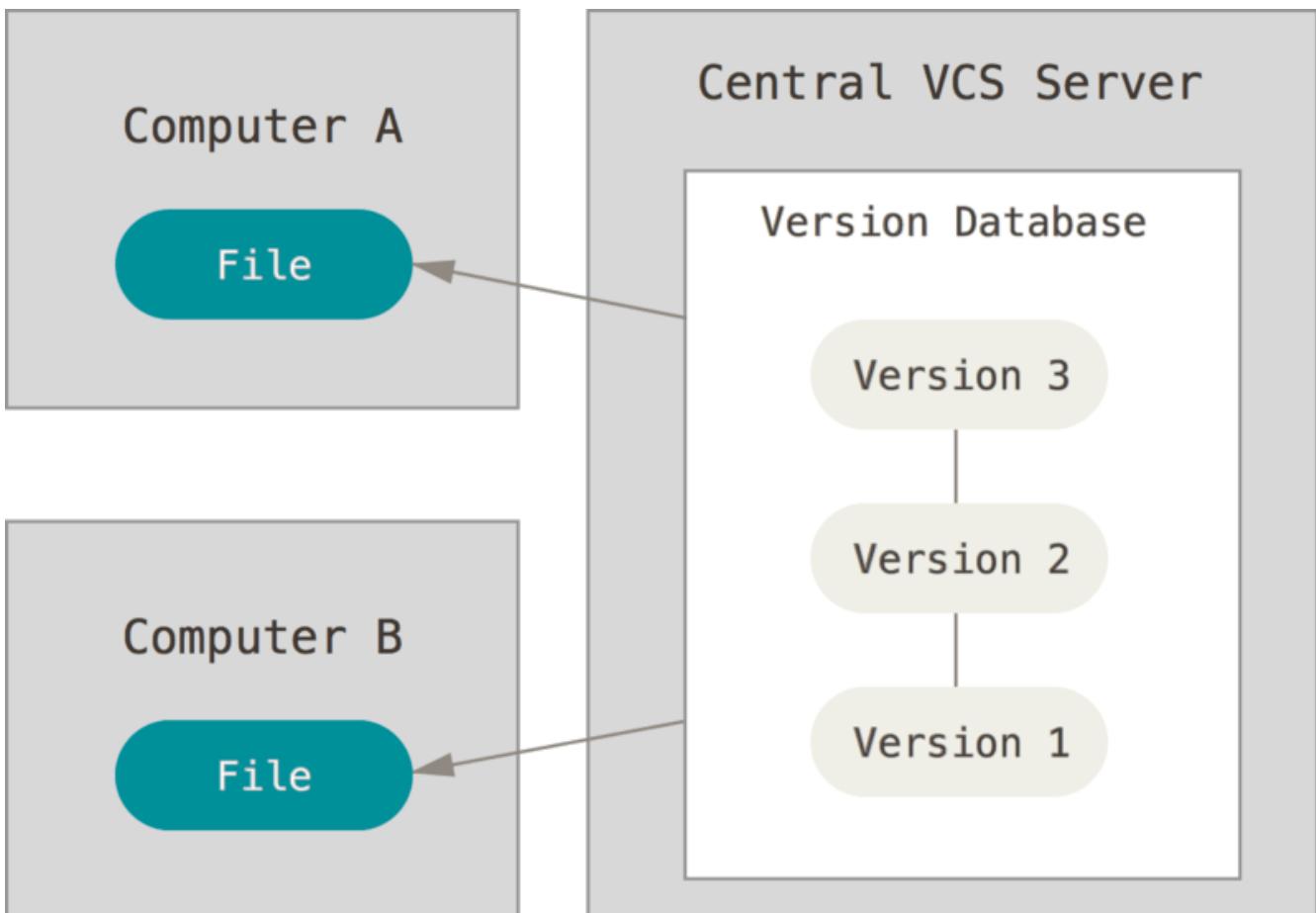


圖表 1. 本機版本控制

其中最流行的一種叫做 RCS，至今許多電腦上都還可以找到他的蹤影。甚至在流行的 Mac OS X 系統中，只要安裝了開發者工具包以後，你就會有 `rcc` 的指令可以使用。RCS 的工作原理是在硬碟上保存一堆特殊格式的補丁集合（patch set，即檔案從一個版本變更到另一個版本所需資訊）；通過套用任意的補丁，便可以重新產生出每個版本的檔案內容。

集中化的版本控制系統

接下來人們又遇到了重大問題，就是如何和其他電腦上的開發者協同合作？為了解決這個問題，於是集中化的版本控制系統應運而生。這類系統（如：CVS、Subversion 和 Perforce），都有一個伺服器來管理所有版本的檔案，而許多用戶端會連到這台伺服器取出檔案來使用。多年以來，這儼然成為版本控制系統的標準做法。



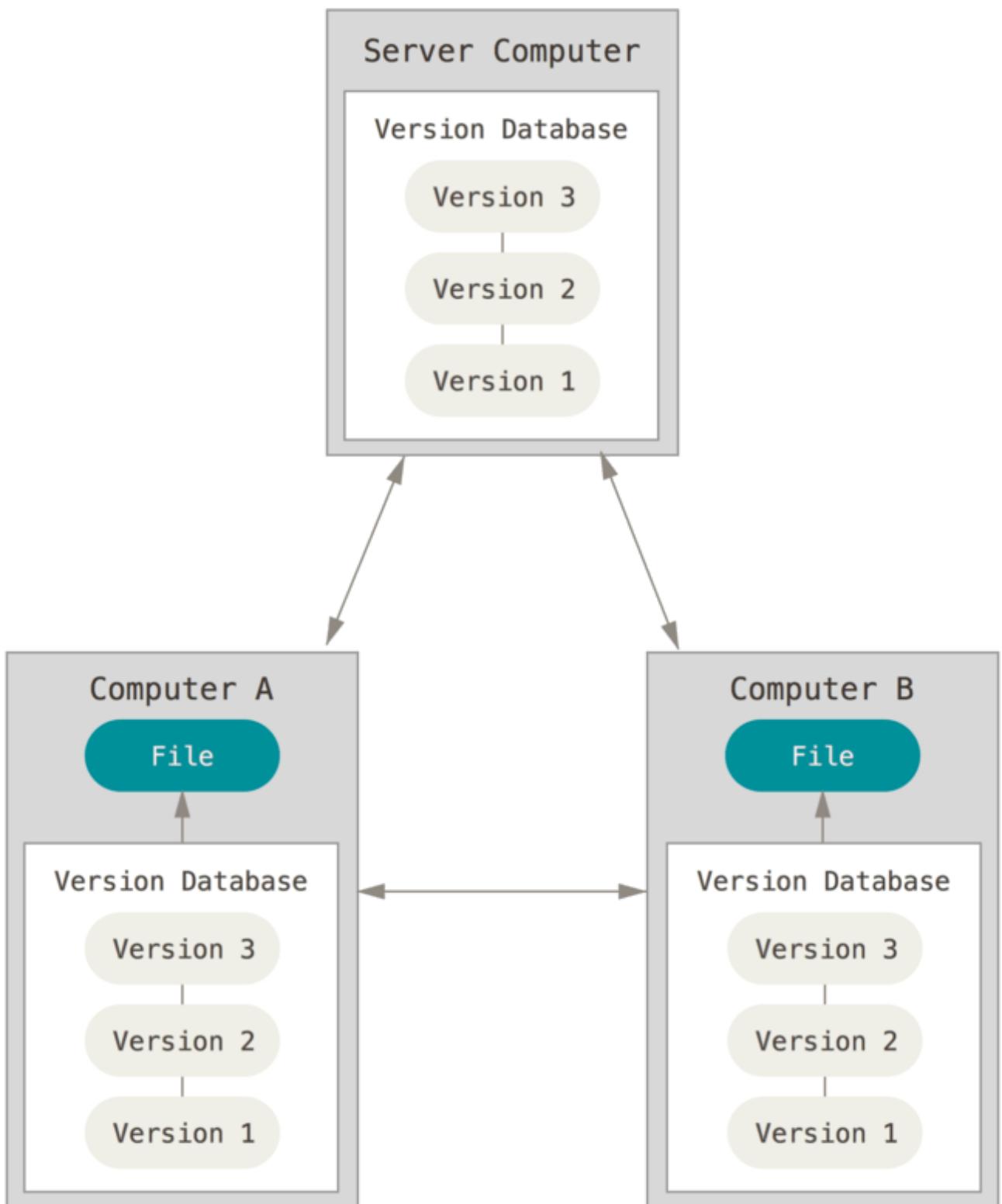
圖表 2. 集中化的版本控制系統。

相對於本機版本控制系統，這種做法帶來了許多好處。舉例來說，每個人都可以一定程度的知道專案中的其他人正在做些什麼。管理員也可以輕鬆掌控每個開發者的權限；而且比每個用戶端只用本機的版本控制系統好管理很多。

然而，集中化的版本控制系統也有一些嚴重的缺點。最嚴重的當然是中央伺服器如果發生故障的時候。如果當機一小時，那麼這個小時之中，沒有人可以提交更新，也就無法協同合作。如果中心版本庫的硬碟發生損壞，又沒有做適當的備份，那麼你就絕對會遺失所有資料——包括專案的全部變更歷史，只會剩下用戶端各自機器上保留的單獨快照。本機版本控制系統也存在類似的問題——只要你的專案歷史都放在同一個地方，就有遺失所有資料的風險。

分散式版本控制系統

於是分散式版本控制系統（Distributed Version Control Systems，簡稱 DVCSs）就此登上舞台。在 DVCS 系統（如 Git、Mercurial、Bazaar 和 Darcs）中，用戶端並不只取出最新的檔案快照；還把整個倉儲做個鏡像。假設有任何一個協同合作的伺服器故障，事後都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。



圖表 3. 分散式版本控制

除此之外，許多這類的系統都可以很好的和許多遠端倉儲互動，所以你可以和不同群組的人使用不同的方式，在同一個專案內協同合作。你可以根據需要設定許多工作流程（如：階層式模型），這是在集中式的版本控制系統中是無法實現的。

Git 的簡史

如同許多生命中偉大的事物一樣，Git 伴隨著一點點創造性破壞和熱烈的討論而生。

Linux kernel 是規模相當大的開放原始碼軟體專案。Linux kernel 在 1991 年到 2002 年間的維護工作，幾乎都是透過補丁和壓縮檔來完成的。在 2002 年時，Linux kernel 開始採用名為 BitKeeper 的商業分散式版本控制系統。

在 2005 年時，開發 Linux kernel 的社群與開發 BitKeeper 的商業公司的合作關係結束，也就無法再免費使用該工具。這就迫使了 Linux 社群（特別是 Linux 之父 Linus Torvalds）基於使用 BitKeeper 所學到的經驗，來開發自有的工具。這個系統必須達成下列目標：

- 快速
- 簡潔的設計
- 完整支援非線性的開發（上千個同時進行的分支）
- 完全的分散式系統
- 能夠有效地處理像 Linux kernel 規模的專案（速度及資料大小）

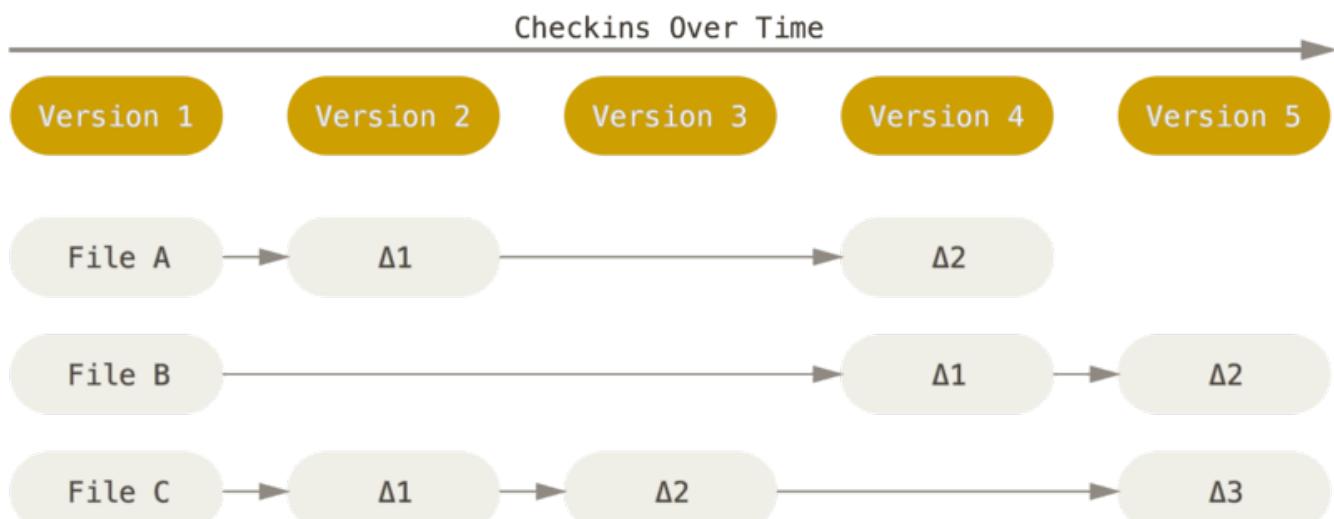
從 Git 在 2005 年誕生後，現在的 Git 已相當成熟，也能很容易上手，並保持著最一开始的要求的品質。它不可思議的快速、處理大型專案非常有效率、也具備相當優秀足以應付非線性開發的分支系統。（參考 [使用 Git 分支](#)）

Git 基礎要點

你要如何用幾句話形容 Git？請仔細閱讀這個重要的章節，如果你瞭解 Git 的本質以及運作的基礎，那麼你將能夠輕鬆且有效率的使用 Git。在學習之前，試著忘記以前所知道的其它版本控制系統，如：Subversion 及 Perforce。這將會幫助你使用此工具時發生不必要的誤解。Git 儲存資料及對待資料的方式遠異於其它系統，即使它們的使用者介面是很相似的。瞭解這些差異會幫助你更準確的使用此工具。

記錄檔案快照，而不是差異

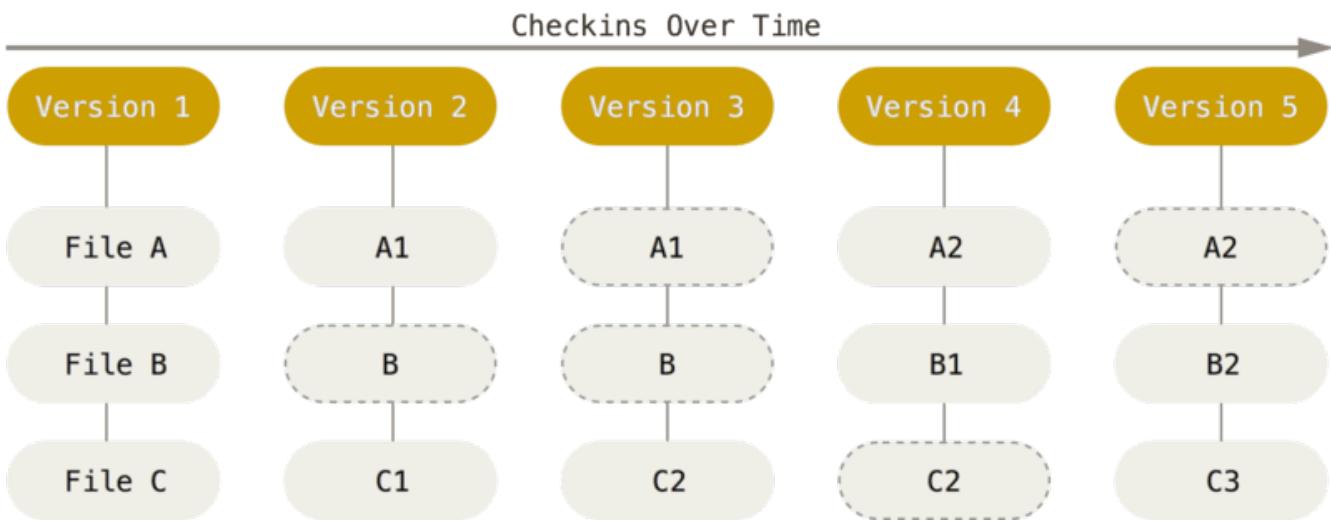
Git 與其它版本控制系統（包含 Subversion 以及與它相關的）最主要的差別是如何處理資料的方式。一般來說，其他大部分的系統是紀錄一連串檔案更改的資訊。這些系統（CVS、Subversion、Perforce、Bazaar…等等）儲存一組基本的檔案以及這些檔案隨時間遞增的更動資料。



圖表 4. 將檔案存成版本與版本之間每個檔案的差異。

但是 Git 不是用這種方式儲存及看待這些資料，而是將其視為小型檔案系統的一組快照（Snapshot）。每當你提交（commit）（註：在 Git 儲存目前專案的狀態）時，Git 會紀錄下你所有目前檔案的樣子，並且參照到這次快照中。為了講求效率，只要檔案沒有變更，Git 不會再度儲存該檔案，而是直接將上一次相同的

檔案參照到這次快照中。Git 把它的資料視為一連串的快照。



圖表 5. 將檔案存成許多次的快照。

這是 Git 和其他相似的版本管理系統之間一個重要的差異。它使得 Git 從各方面重新考量被其他老一輩版本管理系統所採用的作法，並使得 Git 更像是一個上面有一些強大工具的小型檔案系統，而不僅只是版本管理系統。本書將會在 [使用 Git 分支](#) 裡面介紹 Git 分支時，帶領你探索採用這種做法所獲得的好處。

大部份的操作皆可在本地端完成

大部份 Git 的操作皆只需要本地端的檔案及資源即可完成 — 通常並不需要網路上其它電腦的資訊。如果你以前使用過每項操作都需要網路延遲的集中式版本控制系統，在這方面 Git 將會讓你覺得速度快到有如神助。因為專案所有的歷史資料都存在你的本機磁碟中，大多數的操作看起來都像是瞬間完成的。

例如：想要瀏覽專案的歷史時，Git 不需要到伺服器下載歷史再顯示 — 就只需要從本機的資料庫讀取。這意味著你幾乎馬上就可以看到專案的歷史。若讀者想瞭解某個檔案一個月前的版本與現在版本的差別，Git 可以找出一個月前的檔案並在本機比對差異，而不是要求遠端的伺服器執行這項工作，或者從伺服器取回舊版本的檔案之後才在本機比對。

這也代表你只有一點點操作沒辦法在你斷線或是中斷 VPN 後執行。如果你在飛機或火車上想要做些小工作，你可以愉快的提交並且等到連上網路後再上傳。如果你回家後沒辦法使 VPN 正常運作，你仍然可以進行你的工作。在很多其他的系統上，這麼做通常是不可能或是非常痛苦的事。以 Perforce 為例，當你連不上伺服器時，你幾乎沒事可做。在 Subversion 和 CVS 中，你可以修改檔案，但是你沒辦法提交版本（只因為你連不上資料庫）。這看起來可能不是什麼大問題，但是你可能會驚訝於 Git 能做到的事情有這麼大的差異。

Git 能檢查完整性

在 Git 中所有的物件在儲存前都會被計算校驗碼 (checksum) 並以校驗碼參照物件。這意味著你不可能瞞著 Git 對任何檔案或目錄進行修改。此功能內建在 Git 底層並整合到它的設計哲學。Git 更能夠馬上察覺傳輸時的遺失或是檔案的毀損。

Git 用來計算校驗碼的機制稱為 SHA-1 雜湊演算法。一個校驗碼是由 40 個 16 進位的字母 (0-9 和 a-f) 所組成，Git 會根據檔案的內容和資料夾的結構來計算。一個 SHA-1 校驗碼看起來如下所示：

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

你會在 Git 中到處都看到校驗碼，因為校驗碼被 Git 到處使用。事實上在 Git 的資料庫內，每個檔案都是用其

內容的校驗碼來儲存，而不是使用檔名。

Git 通常只增加資料

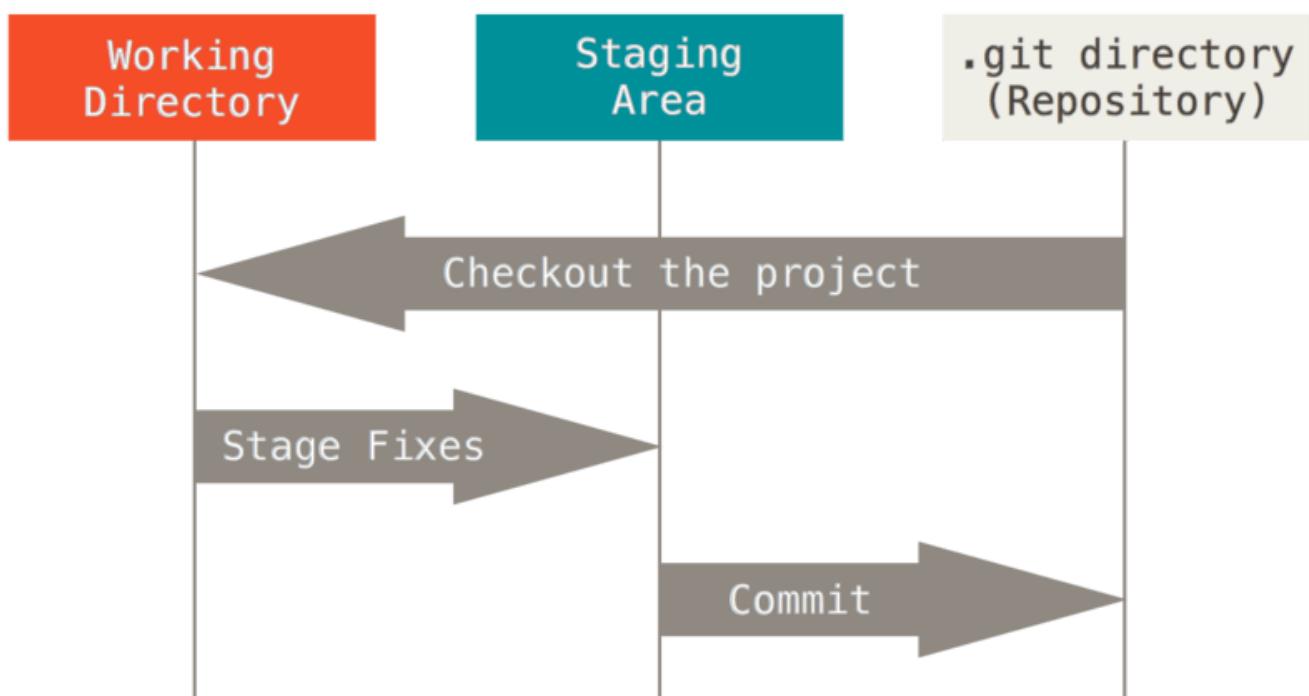
當你使用 Git，幾乎所有的動作都只是增加資料到 Git 的資料庫。你很難藉此讓做出讓系統無法復原或者清除資料的動作。在任何版本控制系統中，你尚未提交的修改都有可能會遺失或者搞亂。但是只要你提交快照到 Git 後，很難會發生遺失的情況，特別是你定期將資料庫推送（push）到其它儲存庫時，就更不可能會弄丟資料。

這讓我們在使用 Git 的時候時可以像在玩玩具一樣，因為我們知道我們可以隨意操作而不會弄壞任何東西。在 [復原](#) 中，我們會進一步討論 Git 如何儲存資料，以及你如何復原看似遺失的資料。

三種狀態

現在，請特別注意。若你希望接下來的學習過程順利些，請務必記住以下這些關於 Git 的知識。Git 會把你的檔案標記為三種主要的狀態：已提交（committed）、已修改（modified）及已預存（staged）。已提交代表這檔案已安全地存在你的本地端資料庫。己修改代表這檔案已被修改但尚未提交到本地端資料庫。已預存代表這檔案將會被存到下次你提交的快照中。

這帶領我們到 Git 專案的三個主要區域：Git 資料夾、工作目錄（working directory）以及預存區（staging area）。



圖表 6. 工作目錄，預存區及 Git 資料夾。

Git 資料夾是 Git 用來儲存你專案的後設資料及物件資料庫的地方。這是 Git 最重要的部份，而且當你克隆一個其他電腦的儲存庫時，這個資料夾也會被同時複製。

工作目錄是專案被檢出的某一個版本。這些檔案從 Git 目錄內被壓縮過的資料庫中拉出來並放在硬碟供你使用或修改。

預存區是一個單一檔案，一般來說放在 Git 目錄下，儲存關於下次提交的資訊。有時它會稱為索引「index」，但現在更常被稱呼為預存區。

基本 Git 工作流程大致如下：

1. 你在你工作目錄修改檔案。
2. 預存檔案，將檔案的快照新增到預存區。
3. 做提交的動作，這會讓存在預存區的檔案快照永久地儲存在 Git 目錄中。

若檔案已被存於 Git 資料夾內，則稱為已提交。若檔案先被修改，接著被增加到預存區域，則稱為已預存。若檔案被檢出後有被修改，但未被預存，則稱為已修改。在 [Git 基礎](#) 內你將會學到更多關於這些狀態的知識以及如何利用它們的優點或者直接略過預存步驟。

命令列

Git 的使用方式有很多。有原始的命令列工具，也有許多不同功能的圖形用戶界面。在這本書，我們將以命令列使用 Git。原因之一是，命令列是可以使用 Git *所有*命令的唯一地方 -- 為簡單起見，大多數的圖形用戶界面只實作了 Git 的部分功能。當你學會使用命令列版本，你也會知道如何使用 GUI 版本；反過來則不一定。而且，選擇哪個圖形客戶端是個人喜好，但是所有使用者都會有安裝好的命令列工具可以使用。

因此，我們希望你知道如何在 Mac 打開終端機或在 Windows 打開命令提示字元或 PowerShell。如果你不清楚我們在說什麼，可能需要先暫停、快速去研究一下，才能跟得上這本書的例子和說明。

Git 安裝教學

在你開始使用 Git 以前，你必須先在你的電腦設定到讓 Git 可以使用。如果你之前已經安裝過，那麼你應該確認 Git 已經升級到最新版。你可以使用套件（package）進行安裝、透過安裝程式或是自行下載原始碼自己編譯。

筆記 本書在撰寫時，Git 的版本為 **2.0.0**。雖然本書使用到的指令在比較舊版的 Git 中通常都可以使用，但是仍然會有一些指令的行為相差極大或是根本無法使用。Git 提供了相當出色的向下相容性，所以如果你目前的 Git 版本大於 2.0，那麼應該是不會有什麼太大的問題。

在 Linux 安裝

如果你想要透過二進位安裝程式安裝基本的 Git 工具集，你通常可以直接透過你所用的發行版（distribution）內建的基礎套件管理工具。舉例來說，如果你使用的是 Fedora，你可以使用 yum：

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

如果你是使用 Debian 系列的發行版，如 Ubuntu，你可以使用 apt-get：

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

如果需要更多選擇，Git 官方網站上有更多其他的發行版中安裝 Git 的安裝步驟，網址為 <http://git-scm.com/download/linux>。

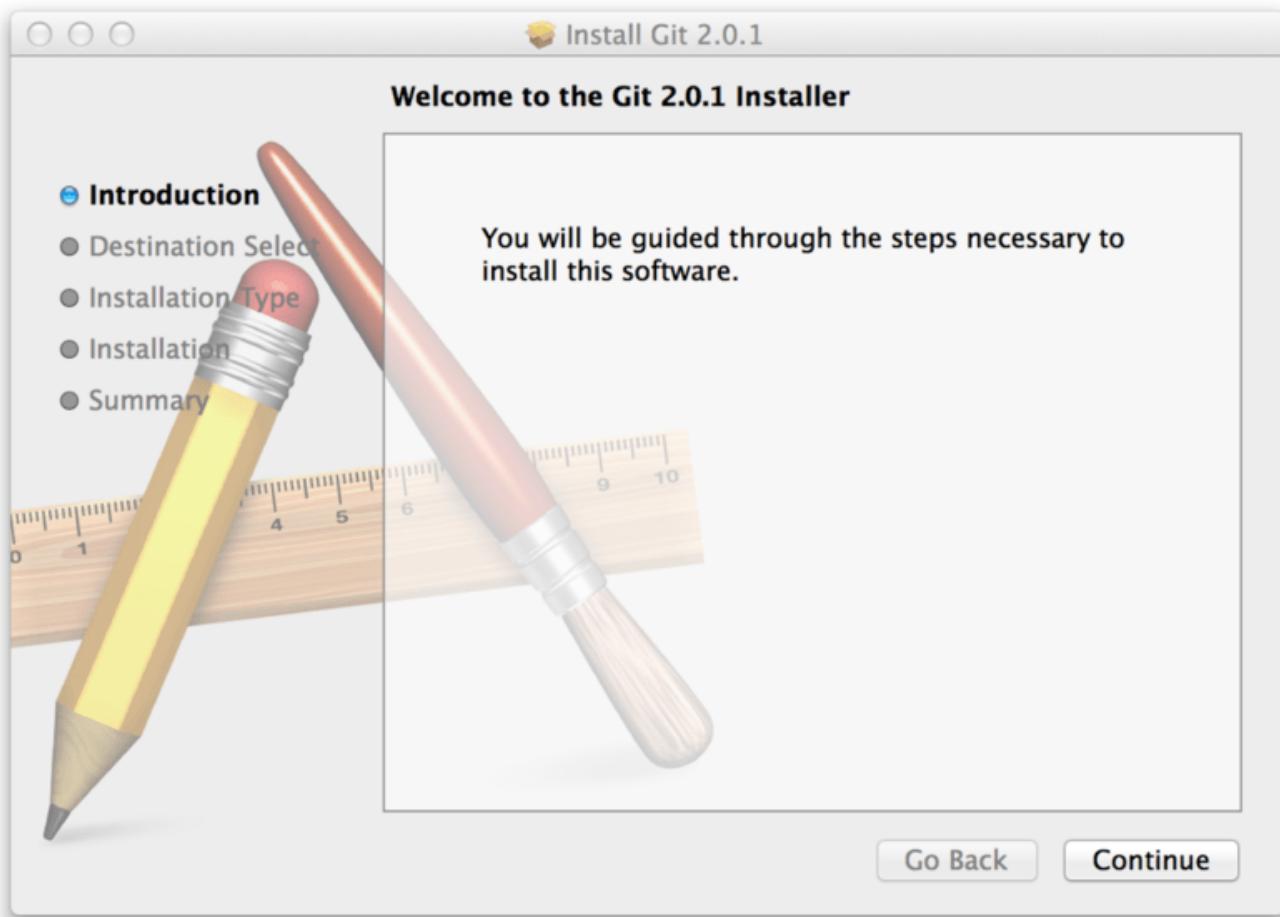
在 Mac 中安裝

在 Mac 中安裝 Git 有很多種方法。最簡單的方法應該就是直接安裝 Xcode 命令列開發者工具（Xcode

Command Line Tools)。在 Mavericks (10.9)

或更新版的系統中，你甚至可以直接在終端機中直接打入「git」指令。如果系統發現你還沒安裝過，便會自動提示你進行安裝。

如果你希望安裝更新的版本，你也可以透過二進位安裝程式安裝。在 Git 官方網站上有維護最新版的安裝程式可供下載，網址在 <http://git-scm.com/download/mac>。



圖表 7. Git OS X 安裝程式

你還可以透過 GitHub 的 Mac 安裝程式來安裝。GitHub 的 Git 圖形化工具提供相對應的選項讓你安裝 Git 命令列工具。你可以從 GitHub for Mac 官方網站下載，網址在 <http://mac.github.com>。

在 Windows 中安裝

在 Windows 中安裝 Git 也有很多種方式。最正式的安裝程式在 Git 官方網站可供下載。你只需要連到 <http://git-scm.com/download/win> 然後下載就會自動開始。請注意這是一個名為 Git for Windows 的專案，與 Git 本身是互相獨立的。如果你需要更多資料，請查閱 <http://git-for-windows.github.io/>。

另一個安裝 Git 的簡單方法就是直接安裝 GitHub for Windows。這個安裝程式內已經預設提供 Git 的命令列版本和圖形化工具。而且它也能夠完美搭配 Powershell，設定實體憑證快取和完整的 CRLF 設定。我們將會在本書的其他章節學到這些事情，但我只想強調，這就是你需要的東西。你可以直接從 GitHub for Windows 下載，網址在 <http://windows.github.com>。

從原始碼安裝

某些人可能會發現從原始碼安裝 Git 反而比較好用，因為你可以拿到最新的 Git 版本。通常二進位安裝程式都會落後於 Git 原始碼的版本，雖然 Git 近幾年已經逐漸成熟，兩者的版本差異可能不大。

如果你希望從原始碼安裝 Git，你需要擁有以下 Git 所需的函式庫：curl, zlib, openssl, expat 和 libiconv。舉例來說，如果你的系統有 yum（例如 Fedora）或 apt-get（例如 Debian 系列的發行版），你可以使用其中一個指令來安裝這些最小相依關係（the minimal dependencies），這樣才有辦法安裝編譯並安裝 Git 可執行檔。

```
$ sudo yum install curl-devel expat-devel gettext-devel \
openssl-devel perl-devel zlib-devel
$ sudo apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
libbz-dev libssl-dev
```

為了能夠建立這些格式（doc、html、info）的文件，你還需要安裝這些額外的相依關係（注意：RHEL 系列（像是 CentOS、Scientific Linux）的使用者必需 [啟用 EPEL 版本庫](#)，才能下載 docbook2X 套件）：

```
$ sudo yum install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

另外，如果你使用 Fedora 或 RHEL 系列，你還需要做這個：

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

這是因為所使用的檔名不同。（譯注：此行命令是對 docbook2x-texi 做一個捷徑，將其「連結」到實際上所使用的 db2x_docbook2texi。）

當你成功的安裝所有必備的相依關係，你就可以繼續下一步：從以下其中一個地方抓回最新的 Git 原始碼 tarball 壓縮檔。你可以從 Kernel.org 網站取得，網址在 <https://www.kernel.org/pub/software/scm/git>；或是在 GitHub 上面的鏡像，網址在 <https://github.com/git/git/releases>。通常在 GitHub 網站上你會比較容易知道哪個原始碼是最新的；但是在 kernel.org 網站上會同時提供該檔案的數位簽章，以便你下載後對檔案進行驗證。

再來，編譯並安裝 Git：

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

當你安裝完以後，你就可以透過 Git 來取得 Git 最新的原始碼如下：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

初次設定 Git

你已經在你的系統上安裝 Git，你或許會想要自訂你的 Git 環境。你在每台電腦上只需設定一次，這些設定

在 Git 更新時會被保留下來。你也可以藉由再次執行隨時變更這些設定。

Git 附帶一個名為 `git config` 的工具，讓你能夠取得和設定組態參數。這些設定允許你控制 Git 各方面的外觀和行為。這些參數被存放在下列三個地方：

1. 檔案 `/etc/gitconfig`：裡面包含該系統所有使用者和使用者倉儲的預設設定。如果你傳遞 `--system` 參數給 `git config`，它就會明確地從這個檔案讀取或寫入設定。
2. 檔案 `~/.gitconfig`、`~/.config/git/config`：你的帳號專用的設定。只要你傳遞 `--global`，就會明確地讓 Git 從這個檔案讀取或寫入設定
3. 任何倉儲中 Git 資料夾的 `config` 檔案（位於 `.git/config`）：這個倉儲的專用設定。

每個層級的設定皆覆蓋先前的設定，所以在 `.git/config` 的設定優先權高於在 `/etc/gitconfig` 裡的設定。

在 Windows 系統，Git 會在 `$HOME` 目錄（對大部份使用者來說是 `C:\Users\$USER`）內尋找 `.gitconfig`。它也會尋找 `/etc/gitconfig`，只不過它是相對於 MSys 根目錄，取決於讀者當初在 Windows 系統執行 Git 的安裝程式時安裝的目的地。如果你使用的 Git for Windows 版本是 2.x 或之後的版本，有個系統層級的組態檔，位於 Windows XP 系統的 `C:\Documents and Settings\All Users\Application Data\Git\config`；而 Vista 及其之後的系統，則位於 `C:\ProgramData\Git\config`。要修改這個組態檔只能透過管理者權限執行 `git config -f <file>`。

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

再次提醒，若你有傳遞 `--global` 參數，只需要做這工作一次，因為在此系統，不論 Git 做任何事都會採用此資訊。若你想指定不同的名字或電子郵件給特定的專案，只需要在該專案目錄內執行此命令，並確定未加上 `--global` 參數。

許多圖形使用者介面的工具會在你第一次使用它們時幫你做這工作。

指定編輯器

現在你的識別資料已設定完畢，讀者可設定預設的文書編輯器，當 Git 需要你輸入訊息時會使用它。預設情況下，Git 會使用系統預設的編輯器。

若你想指定不同的編輯器，例如：Emacs，可以執行下列指令：

```
$ git config --global core.editor emacs
```

如果你想要在 Windows 系統上使用 Notepad++ 做為編輯器，你可以執行下列步驟：

在 x86 系統上

```
$ git config --global core.editor "'C:/Program  
Files/Notepad++/notepad++.exe' -multiInst -nosession"
```

在 x64 系統上

```
$ git config --global core.editor "'C:/Program Files  
(x86)/Notepad++/notepad++.exe' -multiInst -nosession"
```

筆記 在 Unix 類的系統（如 Linux 和 Mac）或者 Windows 系統，Vim、Emacs 和 Notepad++ 是開發者最常用的純文字編輯器。如果你不熟悉使用這些編輯器，你可能需要針對自己想要使用的編輯器，去找出實際的 Git 設定方法。

警告 你或許會發現，如果你沒有設定編輯器，當系統預設編輯器被執行時你很有可能會不知所措。而在 Windows 系統下，使用系統預設編輯器編輯訊息的過程中，有可能因不熟悉而誤用，進而導致某些 Git 操作過早結束，

檢查讀者的設定

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：

```
$ git config --list  
user.name=John Doe  
user.email=johndoe@example.com  
color.status=auto  
color.branch=auto  
color.interactive=auto  
color.diff=auto  
...
```

你可能會看到同一個設定名稱出現多次，因為 Git 從不同的檔案讀到同一個設定名稱（例如：`/etc/gitconfig` 及 `~/.gitconfig`）。在這情況下，Git 會使用最後一個設定名稱的設定值。

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

```
$ git config user.name  
John Doe
```

取得說明文件

若讀者在使用 Git 時需要幫助，有三種取得 Git 命令說明文件的方法：

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

例如：讀者可以下列命令取得 config 命令的說明文件

```
$ git help config
```

這些指令的優點是你可以隨時使用它們，甚至是沒有連上網路的時候。若說明文件及這本書不足以幫助讀者，且讀者需要更進一步的協助，可試著進入 Freenode IRC 伺服器 ([irc.freenode.net](irc://irc.freenode.net)) 的 [#git](#) 或 [#github](#) 頻道。這些頻道平時都有上百位對 Git 非常瞭解的高手，而且通常樂意協助。

摘要

你應該要對 Git 有基本的認識，並了解 Git 與你先前使用的集中式版本管理系統的差異。你的系統上現在應該要有設定過你個人身份的 Git。是時候該學一些基本的 Git 了！

Git 基礎

假如您只能閱讀一章來學習 Git，那麼這個章節就是您的不二選擇。本章節涵蓋你以後使用 Git 來完成絕大多數工作時，最常被使用的基本指令。在讀完本章節後，你應該有能力設定及初始化一個倉儲（repository）、開始及停止追蹤檔案（track）、預存（stage）及提交（commit）更新。本章還會提到如何讓 Git 忽略某些檔案和檔案匹配（patterns）、如何迅速而簡單地撤銷錯誤操作、如何瀏覽你專案的歷史版本及觀看不同提交（commits）之間的變更、以及如何將更新推送（push）至遠端倉儲或從遠端倉儲拉取（pull）提交。

取得一個 Git 倉儲

你有兩種主要方法來取得一個 Git 倉儲。第一種是將現有的專案或者資料夾匯入 Git；第二種是從其它伺服器克隆（clone）一份現有的 Git 倉儲。

在現有資料夾中初始化倉儲

若你打算使用 Git 來追蹤（track）現有的專案，只需要進入該專案的資料夾並執行：

```
$ git init
```

這個命令將會建立一個名為 `.git` 的子資料夾，其中包含 Git 所有必需的倉儲檔案，也就是 Git 倉儲的骨架。到現在這步驟為止，倉儲預設沒有追蹤任何檔案。（想知道你剛建立的 `.git` 資料夾內有些什麼檔案，請參考[Git Internals](#)）

如果你的專案資料夾原本已經有檔案（不是空的），那麼建議你應該馬上追蹤這些原本就有的檔案，然後進行第一次提交。你可以通過多次 `git add` 指令來追蹤完所有你想要追蹤的檔案，然後執行 `git commit` 提交：

```
$ git add *.c  
$ git add LICENSE  
$ git commit -m 'initial project version'
```

進行這些動作應該不會花你太多時間。到現在這步驟為止，你已經得到了一個追蹤若干檔案及第一次提交內容的 Git 倉儲。

克隆現有的倉儲

若你想要取得現有 Git 倉儲的複本（例如：你想要開始協作的倉儲），那你需要使用的命令是 `git clone`。若你熟悉其它像是 Subversion 的版本控制系統，你應該注意現在這個命令是克隆（clone），而非取出（checkout）。這是 Git 和其他版本控制系統的重要差異：Git 並不僅只是取得專案最新的內容，而是把遠端倉儲內幾乎所有的資料都抓回來了。專案歷史紀錄中，每個檔案的每個版本預設都會在你執行 `git clone` 時被拉取（pull）回來。實際上，如果伺服器的硬碟損壞，你通常可以使用任何客戶端克隆的倉儲來將伺服器重建回原本克隆的狀態。（你可能遺失一些伺服器的掛勾程式 `hooks`，但你所有的版本資料都還會健在，請查看 [在伺服器上佈署 Git](#) 獲得更多資訊）

克隆倉庫的命令格式是 `git clone [url]`。例如：若你想克隆名為 libgit2 的 Git linkable library，可以執行下列命令：

```
$ git clone https://github.com/libgit2/libgit2
```

這指令將會建立名為「libgit2」的資料夾，並在這個資料夾下初始化一個 `.git` 資料夾，從遠端倉儲拉取所有資料，並且取出（checkout）專案中最新的版本。若你進入新建立的 `libgit2` 資料夾，你將會看到專案的檔案都在裡面了，並且準備就緒等你進行後續的開發或使用。若你想要將倉儲克隆到「libgit2」以外名字的資料夾，只需要再多指定一個參數即可：

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

這個命令做的事與上一個命令大致相同，只不過在本地創建的倉庫名字變為 `mylibgit`。

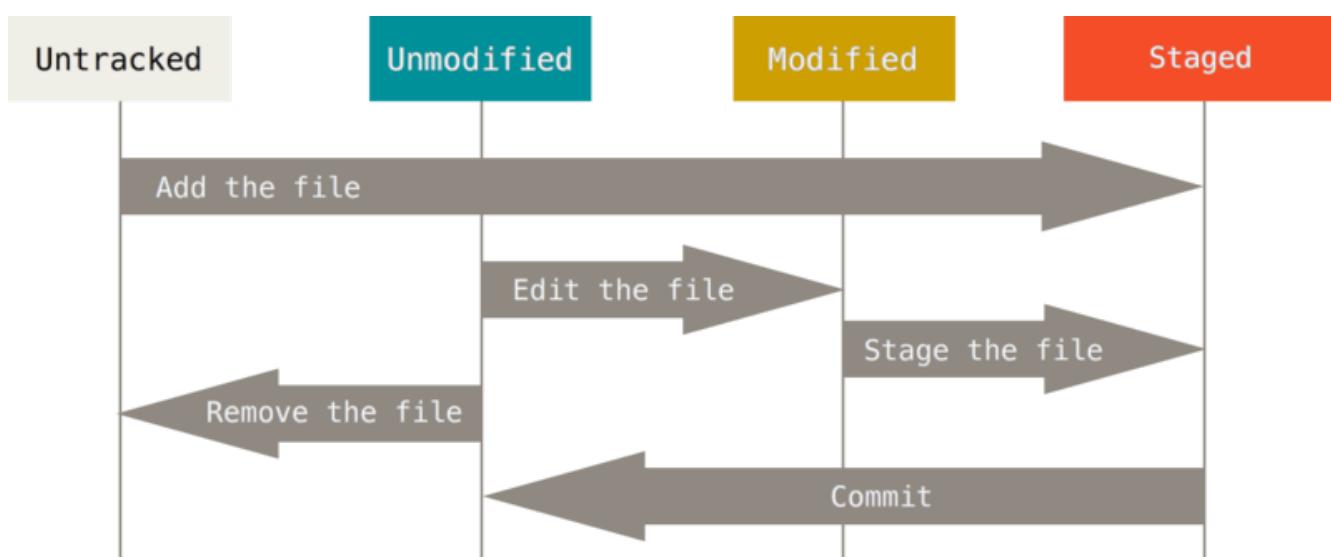
Git 支援多種數據傳輸協定。上一個範例採用 `https://` 協定，但你可能會看過 `git://` 或 `user@server:path/to/repo.git` 等使用 SSH 傳輸的協定。在 [在伺服器上佈署 Git](#) 章節將會介紹這些協定在伺服器端如何配置及使用，以及各種方式的優缺點。

紀錄變更到版本庫中

現在你的手上有了一个貨真價實的 Git 版本庫和這個專案中所有檔案的檢出（checkout）或工作複本（working copy），每當你修改檔案到一個你想記錄它的階段時，你就需要提交（commit）這些變更的快照到版本庫中。

請記住，你工作目錄下的每個檔案不外乎兩種狀態：已追蹤、未追蹤。「已追蹤」檔案是指那些在上次快照中的檔案：它們的狀態可能是「未修改」、「已修改」、「已預存（staged）」；「未追蹤」則是其它以外的檔案——在工作目錄中，卻不包含在上次的快照中，也不在預存區（staging area）中的任何檔案；當你第一次克隆（clone）一個版本庫時，所有檔案都是「已追蹤」且「未修改」，因為 Git 剛剛檢出它們並且你尚未編輯過任何檔案。

隨著你編輯某些檔案，Git 會視它們為「已修改」，因為自從上次提交以來你已經更動過它們；你預存（stage）這些已修改檔案，然後提交所有已預存的修改內容，接著重覆這個循環。



圖表 8. 檔案狀態的生命週期。

檢查你的檔案狀態

`git status` 命令是用來偵測哪些檔案處在什麼樣的狀態下的主要工具；如果你在克隆之後直接執行該命令，應該會看到類似以下內容：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

這意味著你有一個乾淨的工作目錄——換句話說，已追蹤的檔案沒有被修改；Git 也沒有看到任何未追蹤檔案，否則它們會在這裡被列出來；最後，這個命令告訴你目前在哪一個分支上，也告訴你它和伺服器上的同名分支是同步的。到目前為止，該分支一直都是預設的「master」，在這裡你先不用擔心它，[使用 Git 分支](#)會詳細地介紹「分支（branch）」和「參照（reference）」。

假設你在專案中新增一個檔案，例如：一個簡單的 `README` 檔案；如果該檔案先前並不存在，執行 `git status` 命令後，你會看到未追蹤檔案：

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

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

你可以看到新增檔案 `README` 尚未被追蹤，因為它被列在輸出訊息的「Untracked files」欄位下方；基本上「未追蹤」表示 Git 發現這個檔案在上次的快照（提交）中並不存在；Git 並不會將此檔案納入你的提交快照，除非你明確地告訴 Git 要這麼做；它會這樣做是為了避免你意外地將一些二進位暫存檔或其它你並不想的檔案納入版本控制。讓我們開始追蹤 `README` 檔案，因為你確實想要將它開始納入版本控制。

追蹤新的檔案

要開始追蹤一個新的檔案，可以使用 `git add` 命令；要開始追蹤 `README` 檔案，你可以執行：

```
$ git add README
```

如果再次執行檢查狀態命令，可以看到 `README` 檔案現在是準備好被提交的「已追蹤」和「已預存」狀態：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

由於它放在「Changes to be committed」欄位下方，你可以得知它已經被預存，如果你在此時提交，在執行 `git add` 的當下所加進來的檔案版本就會被記錄在歷史快照中；你或許會想到之前執行 `git init` 後也有執行過 `git add (files)`——那就是開始追蹤目錄內的檔案。`git add` 命令接受「檔案」或「目錄」做為路徑名稱；如果是目錄，該命令會用遞迴的方式加入那個目錄下所有的檔案。

預存修改過的檔案

讓我們修改一個已追蹤檔案；假設你修改了一個先前已追蹤的檔案 `CONTRIBUTING.md`，接著再次執行 `git status`，你會看到類似以下文字：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

    modified:   CONTRIBUTING.md
```

`CONTRIBUTING.md` 檔案出現在「Changes not staged for commit」欄位下方——代表著位於工作目錄的已追蹤檔案已經被修改，但尚未預存；要預存該檔案，你可執行 `git add` 命令；`git add` 是一個多重用途的指令——用來「開始追蹤」檔案、「預存」檔案以及做一些其它的事，像是「標記合併衝突（merge-conflicted）檔案為已解決」。比起「把這個檔案加進專案」，把它想成「把檔案內容加入下一個提交中」會比較容易理解。現在，讓我們執行 `git add` 將 `CONTRIBUTING.md` 檔案預存起來，並再度執行 `git status`：

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:  CONTRIBUTING.md
```

這兩個檔案目前都被預存，而且將會成為你下次提交的內容；此時，假設在提交前你想起要對 **CONTRIBUTING.md** 再做一個小修改，你再次開啟檔案並修改它，然後準備提交；然而，當我們再次執行 **git status**：

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:  CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

    modified:  CONTRIBUTING.md
```

見鬼了？現在 **CONTRIBUTING.md** 同時被列在已預存「及」未預存。這怎麼可能？原來 Git 在你執行 **git add** 命令時，的確將當時的檔案內容預存起來；如果你現在提交，最後一次執行 **git add** 命令時，那個當下的 **CONTRIBUTING.md** 的版本會被提交，而不是在提交時你在工作目錄所看到的檔案版本被提交；如果你在 **git add** 後修改檔案，你必需再次執行 **git add** 預存最新版的檔案：

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:  CONTRIBUTING.md
```

簡潔的狀態輸出

雖然 `git status` 輸出內容相當全面，但也相當囉嗦；Git 另外提供一個簡潔輸出的選項，因此你可以以一種較精簡的方式來檢視你的修改；如果你執行 `git status -s` 或 `git status --short`，你可以從該命令得到一個相當簡單的輸出內容：

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

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` 等等。標記有二個欄位——左邊欄位用來指示「預存區」狀態，右邊欄位則是「工作目錄」狀態。所以在這個範例中，在工作目錄中的檔案 `README` 是已修改的，但尚未被預存；而 `lib/simplegit.rb` 檔案則是已修改且已預存的；`Rakefile` 則是曾經修改過也預存過，但之後又再次修改，所以總共有二次修改，一個有預存一個沒有。

忽略不需要的檔案

通常你會有一類檔案不想讓 Git 自動加入，也不希望它們被顯示為未追蹤，這些通常是自動產生的檔案，例如：日誌檔案或者編譯系統產生的檔案；在這情況下，你可以新建一個名為 `.gitignore` 的檔案，在該檔中列舉符合這些檔名的模式（pattern）。以下是一個 `.gitignore` 範例檔內容：

```
$ cat .gitignore
*[oa]
*~
```

第一列告訴 Git 忽略任何副檔名為「.o」或「.a」的檔案，它們可能是編譯系統建置程式碼時所產生的目的檔及連結檔；第二列告訴 Git 忽略所有檔名以波浪號（`~`）結尾的檔案，這種檔案通常被用在很多文字編輯器中，例如：Emacs 把它用在暫存檔；你可能也想忽略 `log`、`tmp`、`pid` 目錄、自動產生的文件等等；在你要開始做事之前將 `.gitignore` 設定好通常是一個不錯的主意，這樣你就不會意外地將實際上並不想追蹤的檔案提交到你的 Git 版本庫。

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。
- 以斜線（`/`）開頭以避免路徑遞迴。（譯注：只忽略特定路徑；如果不以斜線開頭，則不管同名檔案或同名資料夾在哪一層都會被忽略。）
- 以斜線（`/`）結尾代表是目錄。
- 以驚嘆號（`!`）開頭表示將模式規則反向。

Glob 模式就像是 Shell 所使用的簡化版正規運算式（regular expressions）；一個星號（`*`）匹配零個或多個字元、`[abc]` 匹配中括弧內的其中一個字元（此例為 `a`、`b`、`c`）、問號（`?`）匹配單一個字元、中括弧內

的字以連字號連接（如：`[0-9]`）用來匹配任何在該範圍內的字元（此例為 0 到 9）；
你也可以使用二個星號用來匹配巢狀目錄；`a/**/z` 將會匹配到 `a/z`、`a/b/z`、`a/b/c/z` 等等。

以下是另一個 `.gitignore` 範例檔案：

```
# 不要追蹤檔名為 .a 結尾的檔案
*.a

# 但是要追蹤 lib.a，即使上面已指定忽略所有的 .a 檔案
!lib.a

# 只忽略根目錄下的 TODO 檔案，不包含子目錄下的 TODO
/TODO

# 忽略 build/ 目錄下所有檔案
build/

# 忽略 doc/notes.txt，但不包含 doc/server/arch.txt
doc/*.txt

# 忽略所有在 doc/ 目錄底下的 .pdf 檔案
doc/**/*.pdf
```

提示

如果你的專案想要有個好開頭，GitHub 在 <https://github.com/github/gitignore> 中針對幾十種專案和程式語言維護了一個相當完整、好用的 `.gitignore` 範例檔案列表。

檢視已預存及未預存的檔案

如果 `git status` 命令提供的資訊對你來說太過簡略——你要想精確地知道你修改了什麼，而不只是那些檔案被修改——你可以使用 `git diff` 命令；稍後我們會更詳盡講解 `git diff` 命令，然而大部分你在使用它的時候只是為了瞭解兩個問題：已修改但尚未預存的內容是哪些？已預存而準備被提交的內容又有哪些？儘管 `git status` 命令透過列出檔名的方式大略回答了這些問題，但 `git diff` 可顯示檔案裡的哪些列被加入或刪除——如同以往地以補綴（patch）格式呈現。

假設你再次編輯並預存 `README` 檔案，接著修改 `CONTRIBUTING.md` 檔案卻未預存它，如果你執行 `git status` 命令，你會再次看到類似以下資訊：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   CONTRIBUTING.md
```

想瞭解尚未預存的修改，輸入不帶其它參數的 `git diff`：

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your
PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your
change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your
patch is
+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a
PR
that highlights your work in progress (and note in the PR title that
it's
```

這命令會比對「工作目錄」和「預存區」之間的版本，然後顯示尚未被存入預存區的修改內容。

如果你想檢視你已經預存而接下來將會被提交的內容，可以使用 `git diff --staged`；這個命令比對的對象是「預存區」和「最後一次提交」。

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

很重要且需要注意的一點是 `git diff` 不會顯示最後一次提交後的所有變更——只會顯示未預存的變更；這會讓人困惑，因為如果你預存了所有的變更，`git diff` 不會輸出任何內容。

舉其它例子，如果你預存 `CONTRIBUTING.md` 檔案後又編輯它，你可以使用 `git diff` 檢視檔案中哪些變更是已預存的、哪些是尚未預存的。如果它看起來像這樣：

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   CONTRIBUTING.md
```

現在你可以使用 `git diff` 來檢視哪些部分是仍然未預存的：

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
## Starter Projects

See our [projects
list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

以及使用 `git diff --cached` 檢視哪些部分是已預存的（`--staged` 和 `--cached` 是同義選項）：

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your
PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your
change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your
patch is
+longer than a dozen lines.
```

If you are starting to work on a particular area, feel free to submit a PR

that highlights your work in progress (and note in the PR title that it's

Git 外部差異比對工具

接下來我們還會在書中的其它地方以各種不同的用法來使用 `git diff` 命令；如果你傾向於使用圖形或外部差異比對檢視工具，有另一種方法可以查看這些差異內容；執行 `git difftool` 取代 `git diff`，你可以用軟體工具檢視任何這類型的差異，像是 `emerge`、`vimdiff` 或其它更多的工具（包括商業化的產品）；執行 `git difftool --tool -help` 以查看在你系統上有什麼可用的。

提交你的修改

現在你的預存區已被建構成你想要的，你可以開始提交你的變更；記住：任何未預暫存的檔案——新增的、已修改的，自從你編輯它們卻尚未用 `git add` 預存的——將不會納入本次的提交中；它們仍以「已修改」的身份存在磁碟中。在目前情況下，假設你上次執行 `git status` 時，你看到所有檔案都已經被預存，因此你準備提交你的變更。最簡單的提交方式是輸入 `git commit`：

```
$ git commit
```

這麼做會啟動你選定的編輯器（由你的 Shell 的 `$EDITOR` 環境變數所指定——通常是 vim 或 emacs；你也可以如同 [開始](#) 所介紹的，使用 `git config --global core.editor` 命令指定任何一個你想使用的）。

編輯器會顯示如下文字（此範例為 Vim 的畫面）：

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   new file: README
#   modified: CONTRIBUTING.md
#
~
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

你可以看到預設的提交訊息中包含最近一次 `git status` 的輸出並以註解方式呈現，以及最上方有一列空白列；你可以移除這些註解後再輸入提交訊息，或者保留它們以提醒你現在正在提交什麼樣的內容。（如果想對你已經修改的內容得到更明確的提示，可以在 `git commit` 上加上 `-v` 選項；這麼做連修改的差異內容也會被放到編輯器中，如此你便可以精確地看到你正在提交的修改內容。）當你關閉編輯器，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。

另一種方式則是在 `commit` 命令的 `-m` 選項後方直接輸入提交訊息，如下：

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

現在你已經建立了你的第一個提交！你可從輸出訊息看到此提交相關資訊：提交到哪個分支（`master`） 、提交的 SHA-1 校驗碼（`463dc4f`） 、有多少檔案被更動，以及統計此提交有多少列被新增和被移除。

記住：那個提交記錄了你放在預存區的快照。任何你尚未預存的已修改檔案仍然安好地在那裡，你可以做另一次提交來把它加入到你的歷史中；每一次提交時，你都正在對專案記錄一個快照，可以在之後用來「復原」或「比對」。

略過預存區

雖然「預存區」的用法讓你能夠很有技巧地且精確地提交你想記錄的內容而意外地好用，但有時候它也比你實際需要的工作流程要繁瑣得多；如果你想跳過預存區，Git 提供了一個簡易的捷徑，在 `git commit` 命令加上 `-a` 選項，使 Git 在提交前自動預存所有已追蹤的檔案，讓你略過 `git add` 步驟：

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working
     directory)

      modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

請留意這種使用情況：在提交之前，你並不需要執行 `git add` 來預存 `CONTRIBUTING.md` 檔案；那是因為 `-a` 選項會納入所有已變更的檔案；很方便，但請小心，有時候它會納入你並不想變更。

移除檔案

要從 Git 中刪除一個檔案，你需要將它從已追蹤檔案中移除（更準確地說，是從預存區中移除），然後再提交；`git rm` 命令可完成此工作，它同時也會將該檔案從工作目錄中移除，如此它之後也不會身為未追蹤檔案而被你看到。

如果你僅僅是將檔案從工作目錄中移除，那麼它會被列在 `git status` 輸出內容的「Changed but not updated」（也就是「未預存」）欄位下面：

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working
     directory)

      deleted:   PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

如果你接著執行 `git rm`，它會預存該檔案的移除動作：

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

下一次提交時，該檔案將會消失而且不再被追蹤；如果你修改了檔案且已經把修改內容加入索引中（譯注：「加入索引」和「預存」是同義詞），你必需使用 **-f** 選項才能強制將它移除；這是一種為了避免已記錄的快照意外被移除後再也無法使用 Git 復原的保護機制。

另一個有用的技巧是保留工作目錄的檔案，但將它從預存區中移除；換句話說，你或許想保留在磁碟機上的檔案但不希望 Git 再繼續追蹤它；當你忘記將某些檔案加到 **.gitignore** 中而且不小心預存它的時候會特別用有，像是不小心預存了一個大的日誌檔案或者一堆 **.a** 已編譯檔案。加上 **--cached** 選項可做到這件事：

```
$ git rm --cached README
```

你可將「檔案」、「目錄」、「file-glob 模式」做為參數傳給 **git rm** 命令，那意味著你可以做類似下面的事：

```
$ git rm log/*.*log
```

注意：星號 ***** 前面有反斜線 (****)；這是必須的，因為 Git 在你的 Shell 檔名擴展 (filename expansion) 之上另外有自己的檔名擴展；此命令會移除在 **log/** 所有副檔名為 **.log** 的檔案。或者你也可以做像下面的事：

```
$ git rm \*~
```

此命令會移除所有以 **~** 結尾的檔案。

移動檔案

Git 不像其它 VCS 系統，它並不會明確地追蹤檔案的移動；如果你在 Git 中重新命名一個檔案，並不會有任何 Git 後設資料記錄這個動作以辨別你曾經重新命名過檔案；然而 Git 可以在檔案移動後很聰明地將它們找出來——我們稍後會對偵測檔案的移動再多做一點說明。

因此 Git 有一個 **mv** 命令反而有點令人困惑；如果你想要在 Git 中重新命名一個檔案，你可以執行以下命令：

```
$ git mv file_from file_to
```

並且它運作地想當好；事實上，如果你執行類似以下的動作然後檢視一下狀態，你將看到 Git 將該檔案視為一個重新命名過的檔案：

```
$ git mv README.md README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

其實，它相當於執行下列命令：

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git 會在背後判斷檔案被重新命名，因此不管是用上述方法還是用 `mv` 命令並沒有差別；實際上唯一不同的是 `mv` 是一個命令，而不是三個——它只是個方便的功能。更重要的是你可以使用任何你喜歡的工具來重新命名一個檔案，然後在提交前才使用 `add/rm`。

檢視提交的歷史記錄

在產生數筆提交 (commit) 或者克隆 (clone) 一個已有歷史記錄的版本庫之後，你或許會想要檢視之前發生過什麼事；最基本也最具威力的工具就是 `git log` 命令。

以下範例使用一個非常簡單的「simplegit」專案做展示；欲取得此專案，執行：

```
$ git clone https://github.com/schacon/simplegit-progit
```

在此專案目錄內執行 `git log`，你應該會看到類似以下訊息：

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

預設情況（未加任何選項）`git log` 以反向的時間順序列出版本庫的提交歷史記錄——也就是說最新的提交會先被列出來；如你所見，它也會列出每筆提交的 SHA-1 校驗碼、作者名字及電子郵件、寫入日期以及提交訊息。

`git log` 命令有大量且多樣的選項，能精確地找出你想搜尋的結果；在這裡，我們會展示一些最受歡迎的選項。

最有用的選項之一是 `-p`，用來顯示每筆提交所做的修改內容；你還可以加上 `-2` 選項，限制只輸出最後兩筆提交內容。

```

$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.platform = Gem::Platform::RUBY
  s.name      = "simplegit"
-  s.version   = "0.1.0"
+  s.version   = "0.1.1"
  s.author    = "Scott Chacon"
  s.email     = "schacon@gee-mail.com"
  s.summary   = "A simple gem for using Git in Ruby code."

```

```

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
  end

end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file

```

這個選項除了顯示相同的資訊以外，還會在每筆提交資訊後面附加每個修改檔案的差異內容（譯註：使用 `-` `+` 來表示差異，`-` 是刪除行，`+` 是新增行；未修改的上下文資訊預設是三行，用來定位有修改的地方）；對於「程式碼審核」或「快速瀏覽」協同工作者所新增的一系列提交內容，這是非常有幫助的。你也可以使用 `git log` 提供的一系列「摘要」選項；例如：若想檢視每筆提交簡略的統計資訊，你可以使用 `--stat` 選項：

```

$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

Rakefile | 2 +-  

1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

lib/simplegit.rb | 5 -----  

1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

README          |  6 ++++++
Rakefile        | 23 ++++++++++++++++++++++++
lib/simplegit.rb | 25 ++++++++++++++++++++++++
3 files changed, 54 insertions(+)

```

如你所見，`--stat` 選項在每筆提交訊息的下方列出「被更動的檔案」、「總共有多少檔案被更動」、「這些檔案中有多少行被加入或移除」；它也會在最後印出總結訊息。

另一個實用的選項是 `--pretty`，用來改變原本預設輸出的格式；有數個內建的選項供你選用，其中 `oneline` 選項將每一筆提交顯示成單獨一行，對於檢視大量的提交時很有用；更進一步，`short`、`full`、`fuller` 選項輸出的格式大致相同，但分別會少一些或者多一些資訊。

```

$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

```

最有趣的選項是 `format`，讓你可以指定自訂的輸出格式；當需要輸出給機器分析時特別有用——因為明確地指定了格式，即可確定它不會因為更新 Git 而被更動：

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

`git log --pretty=format` 實用選項 列出 `format` 一些更實用的選項。

表格 1. `git log --pretty=format` 實用選項

選項	輸出說明
%H	該提交 SHA-1 雜湊值
%h	該提交簡短的 SHA-1 雜湊值
%T	「樹 (tree)」物件的 SHA-1 雜湊值
%t	「樹」物件簡短的 SHA-1 雜湊值
%P	親代 (parent) 提交的 SHA-1 雜湊值
%p	親代提交簡短的 SHA-1 雜湊值
%an	作者名字
%ae	作者電子郵件
%ad	作者日期 (依據 --date 選項值而有不同的格式)
%ar	作者日期，相對時間格式。
%cn	提交者名字
%ce	提交者電子郵件
%cd	提交者日期
%cr	提交者日期，相對時間格式。
%s	標題

你可能會好奇「作者 (author)」與「提交者 (committer)」之間的差別，作者是最初修改的人，而提交者則是最後套用該工作成果的人；因此，如果你送出某個專案的補綴，而該專案其中一個核心成員套用該補綴，則你與該成員都有功勞——你是作者，而該成員則是提交者。我們會在 [分散式的 Git](#) 提到更多它們之間的差別。

當 `oneline`、`format` 和另一個 `log` 選項 `--graph` 結合在一起使用時將特別有用，該選項會附加一個還不錯的 ASCII 圖形用來顯示分支及合併的歷史。

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
| \
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
| /
* d6016bc require time for xschema
* 11d191e Merge branch 'defunkt' into local
```

當下一個章節談到分支 (branching) 和合併 (merging) 時，這種輸出型式將會變得更為有趣。

這些只是一些簡單的 `git log` 格式化輸出選項——還有更多其它的；[git log 的常用選項](#) 列出我們目前涵蓋的以及一些你可能常常會用到的格式化選項，以及它們會如何改變 `log` 命令的輸出格式。

表格 2. `git log` 的常用選項

選項	說明
<code>-p</code>	顯示每筆提交的補綴。
<code>--stat</code>	顯示每筆提交中更動檔案的統計及摘要資訊。
<code>--shortstat</code>	只顯示 <code>--stat</code> 提供的訊息中關於更動、插入、刪除的文字。
<code>--name-only</code>	在提交訊息後方顯示更動的檔案列表。
<code>--name-status</code>	在檔案列表顯示「新增」、「更動」、「刪除」等資訊。
<code>--abbrev-commit</code>	只顯示 SHA-1 校驗碼的前幾位數，而不是顯示全部 40 位數。
<code>--relative-date</code>	以相對時間格式顯示日期（例如：「2 weeks ago」），而不是使用完整的日期格式。
<code>--graph</code>	在輸出的日誌旁邊顯示分支及合併歷史的 ASCII 圖形。
<code>--pretty</code>	以其它格式顯示提交。選項包括 <code>oneline</code> 、 <code>short</code> 、 <code>full</code> 、 <code>fuller</code> 及可自訂格式的 <code>format</code> 。

限制日誌的輸出

除了輸出格式的選項以外，`git log` 還有一些有用的輸出限制選項——也就是讓你能夠只顯示一個子集合的提交；你先前已看過其中一個——用 `-2` 選項只顯示最後二筆提交，事實上，你可以用 `-<n>`，其中 `n` 是任意整數，用來顯示最後 `n` 筆提交；實際上，你可能不太會那麼常用到它，因為 Git 預設把輸出導向分頁器，所以你一次只能看到一頁的日誌輸出內容。

然而，像 `--since` 和 `--until` 這些限制時間的選項就很有用；例如，以下命令列出最近兩週以來的提交：

```
$ git log --since=2.weeks
```

這個命令支援各種格式——你可以指定特定的日期格式（例如："2008-01-15"），或者相對日期格式（例如："2 years 1 day 3 minutes ago"）。

你也可以過濾列表中符合某些搜尋條件的提交；`--author` 選項允許你過濾特定作者，而`--grep` 選項允許你以關鍵字搜尋提交訊息。（注意：如果你想要同時比對作者及提交訊息，你必需加上`--all-match`，否則只要滿足其中一個條件的提交都會被列出來。）

另一個實用的選項是`-S`，用來尋找所修改的內容中被加入或移除某字串的提交；舉例，如果你想要找出最後一個有加入或移除某個特定函數參照的提交，你可以使用：

```
$ git log -Sfunction_name
```

最後一個實用的`git log` 過濾選項是路徑，如果你指定一個目錄或檔名，你可以列出只對這些檔案有修改記錄的提交；這個選項永遠放在最後一個，並且通常使用二個連接號（`--`）將路徑與其它選項隔開。

我們在 [Options to limit the output of git log](#) 中列出這些選項和一些其它常用選項供你參考。

表格 3. *Options to limit the output of git log*

選項	說明
<code>-(n)</code>	只顯示最後 n 筆提交。
<code>--since, --after</code>	列出特定日期後的提交。
<code>--until, --before</code>	列出特定日期前的提交。
<code>--author</code>	列出作者名字符串符合指定字串的提交。
<code>--committer</code>	列出提交者名字符串符合指定字串的提交。
<code>--grep</code>	列出提交訊息中符合指定字串的提交。
<code>-S</code>	列出修改檔案中有加入或移除指定字串的提交。

例如：如果你想檢視 Git 原始碼的測試檔案中（譯註：它們都放在資料夾`t/`），由 Junio Hamano 在 2008 年 10 月份所提交，但不包含「合併提交」的提交。可執行以下的命令：

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
--before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link
HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into
an unborn branch
```

在 Git 原始碼接近 40,000 筆提交歷史記錄中，這個命令列出其中符合條件的 6 筆。

復原

在任何一個過程中，你都可能想要復原某些內容，在這裡我們會回顧一些基本的工具用來復原你做過的修改內容；小心！因為復原操作不是永遠都可逆的，這是少數在使用 Git 時，執行錯誤的動作會遺失資料的情況。

一個常見的復原操作發生在當你太早提交（commit），接著才發現忘了加入某些檔案，或者寫錯了提交訊息；如果你想要重新提交，你可以在提交命令上使用 `--amend` 選項：

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

這個命令會再次把預存區（staging area）拿來提交，如果自從上次提交以來你沒有做過任何檔案修改（例如：在上一次提交後，馬上執行此命令），那麼整個快照看起來會與上次提交的一模一樣，唯一有可能更動的是提交訊息。

同樣用來提交訊息的文字編輯器會先啟動，並且已填好上一次提交的訊息內容；你可以像往常一樣編輯這些訊息，接著它會覆蓋掉上一次的提交。

例如：如果你提交後才意識到你想要把某些忘記預存（stage）的修改也一併加入到上一個提交中，你可以這樣做：

```
$ git commit -m 'initial commit'  
$ git add forgotten_file  
$ git commit --amend
```

最終只會得到一個提交——第二次的提交取代了第一次提交的結果。

將已預存的檔案移出預存區

接下來的兩節會展示如何操作預存區和工作目錄中已修改的檔案；用來顯示這二個區域狀態的命令也會好心地提示你如何做復原操作，例如：假設你已經修改了二個檔案，並且想要分別提交它們，但是你卻意外地使用 `git add *` 把它們二個都預存了，要如何將其中一個「移出預存區（unstage）」呢？`git status` 命令提示你：

```
$ git add *  
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
    renamed: README.md -> README  
    modified: CONTRIBUTING.md
```

在「Changes to be committed」文字正下方，說明了使用 `git reset HEAD <file>...` 將檔案移出預存區；因此，讓我們依循該建議將 `CONTRIBUTING.md` 檔案移出預存區：

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M  CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

modified:   CONTRIBUTING.md
```

這個命令有一點奇怪，不過它的確可行；`CONTRIBUTING.md` 檔案現在又回到了「修改」但「未預存 (unstaged)」的狀態。

筆記 雖然 `git reset` 命令加上 `--hard` 選項會讓它成為一個危險的命令，但在本例中工作目錄內的檔案卻不會被修改到；這是因為執行沒有附加選項的 `git reset` 命令並不危險——它只會修改預存區。

關於 `git reset` 命令，到目前為止所有你需要知道的就只有這個神奇用法；我們將在 [Reset Demystified](#) 中深入了解 `reset` 更多的細節，包括「它可以做什麼」以及「如何操控它做一些真正有趣的事情」。

復原被修改的檔案

當你不要保留 `CONTRIBUTING.md` 檔案的修改時該怎麼辦？你如何才能輕易地復原它——將它還原到上次提交時的樣子（或最初克隆時、或當初放到工作目錄時的版本）？很幸運的，`git status` 也告訴你該如何做；在上一個範例的輸出中，有修改而未預存的內容長得像這樣：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

modified:   CONTRIBUTING.md
```

它相當明確地提示你如何捨棄工作目錄所做的修改，讓我們跟著提示做：

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed: README.md -> README
```

你可以看到那些修改已經被還原了。

重要 你必需明瞭一件很重要的事：`git checkout -- <file>` 是一個危險的命令，你對那個檔案所做的任何修改都會消失——Git 只是複製了另一個檔案來覆蓋它；除非你很肯定地知道你不需要那個檔案了，否則千萬不要使用這個命令。

如果你仍然想保留那個檔案所做的修改，但是某個當下需要先復原檔案，我們將會在 [使用 Git 分支](#) 中介紹「收藏（stashing）」和「分支（branching）」，一般而言它們是比較好的做法。

切記，在 Git 「已提交」的任何東西幾乎總是能夠被復原的，即使是在被刪除的分支上曾經出現過的提交，或者因為 `--amend` 而被覆蓋掉的提交，它們都是可以被復原的（詳見 [Data Recovery](#) 以了解資料復原）；然而，從來沒被提交過的內容，失去後大概就沒辦法再救回來了。

與遠端協同工作

為了能在任意的 Git 專案上協同工作，你需要知道如何管理你的遠端版本庫。遠端版本庫是指被託管在網際網路或其他網路中的各種專案版本庫。你可以擁有許多遠端版本庫；通常來說，它如果不是唯讀的，就是可讀寫的。與其它人協同工作包括了：「管理」遠端版本庫、以及將分享的資料「推送（push）」到端遠版本庫、或者從遠端版本庫「拉取（pull）」分享的資料；管理遠端版本庫則包括了了解如何：「新增」遠端版本庫、「移除」不再有效的遠端版本庫、管理各式各樣的「遠端分支」、定義遠端分支是否被「追蹤」等等。我們將在這一節介紹這些遠端管理技巧。

顯示你的遠端

使用 `git remote` 命令可以檢視你已經設定好的遠端版本庫，它會列出每個遠端版本庫的「簡稱」。如果你克隆（clone）了一個遠端版本庫，你至少看得到「origin」——它是 Git 給定的預設簡稱，用來代表被克隆的來源。

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

你也可以指定 `-v` 選項來顯示 Git 用來讀寫遠端簡稱時所用的網址。

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

如果遠端版本庫不止一個，這個命令會將它們全部列出來。例如，一個版本庫內連結了許多其它協作者的遠端版本庫，可能看起來就像這樣：

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45      https://github.com/cho45/grit (fetch)
cho45      https://github.com/cho45/grit (push)
defunkt    https://github.com/defunkt/grit (fetch)
defunkt    https://github.com/defunkt/grit (push)
koke       git://github.com/koke/grit.git (fetch)
koke       git://github.com/koke/grit.git (push)
origin     git@github.com:mojombo/grit.git (fetch)
origin     git@github.com:mojombo/grit.git (push)
```

這意味著我們可以很輕鬆地拉取任何協作者的貢獻；我們可能對其中某些遠端版本庫還擁有推送權限，不過我們不會在這裡詳述這個部分。

注意：這些遠端版本庫使用了不同的通訊協定；我們將會在 [在伺服器上佈署 Git](#) 對它有更多的說明。

新增遠端版本庫

我們在之前的章節中已經提過並給了一些範例來說明 `clone` 命令如何隱式地為你加入 `origin` 遠端；而在這裡將說明如何「明確地」新增一個遠端。選一個你可以輕鬆引用的簡稱，用來代表要新增的遠端 Git 版本庫，然後執行 `git remote add <簡稱> <url>` 來新增它：

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb    https://github.com/paulboone/ticgit (fetch)
pb    https://github.com/paulboone/ticgit (push)
```

現在你可以在命令列中使用 `pb` 這個字串來代表整個網址；例如，如果你想從 Paul 的版本庫中取得所有資訊，而這些資訊並不存在於你的版本庫中，你可以執行 `git fetch pb`：

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master      -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

現在 Paul 的 master 分支可以在本地端透過 `pb/master` 存取到（譯註：把 `pb/master` 想成 Paul 的 master 在你本地端的「分身」——實際上它被稱為「遠端追蹤分支（remote-tracking branch）」）——你可以把它合併（merge）到你的其中一個分支；或者，如果你想要檢視它的話，可以在它身上檢出（checkout）一個本地分支。（我們將會在 [使用 Git 分支](#) 中詳細介紹什麼是分支以及如何使用它們。）

從你的遠端獲取或拉取

如同你剛剛所看到的，要從你的遠端專案中取得資料，你可以執行：

```
$ git fetch [remote-name]
```

這個命令會連到遠端專案，然後從遠端專案中將你還沒有的資料全部拉下來；執行完成後，你應該會有那個遠端版本庫中所有分支的參照（reference）（譯註：再次強調，遠端的分支在本地端的分身——遠端追蹤分支），可以隨時用來合併或檢視。

如果你克隆了一個版本庫，`clone` 命令會自動增加一個「origin」來代表遠端版本庫；所以，`git fetch origin` 會獲取（fetch）在你克隆（或者最後一次獲取）之後任何被推送到伺服器上的新的工作內容。很重要的一點是：`git fetch` 命令只會下載資料到你的版本庫——它並不會自動合併你的任何工作內容，也不會自動修改你正在修改的東西；當你準備好合併你的工作內容時，你必需用手動的方式進行合併。

如果你的目前分支被設定為「追蹤」遠端上的分支（閱讀下一節以及 [使用 Git 分支](#) 以了解更多資訊），你便可使用 `git pull` 命令來自動「獲取」並「合併」那個遠端分支到你目前的分支裡去；由於 `git clone` 命令會「自動地」將本地分支 `master` 設定為「追蹤」遠端上的 `master`（無論預設分支叫什麼名稱。譯註：只要是預設分支都會自動設定追蹤行為，而 `master` 常常是預設分支），這可能會讓你有比較輕鬆自在的工作流程：只要執行 `git pull` 通常就會從你最初克隆的伺服器上獲取資料，然後試著自動合併到目前的分支。

推送到你的遠端

當你想分享你的專案成果時，你必需將它推送到上游。推送的命令很簡單：`git push [remote-name] [branch-name]`。如果你想要將 `master` 分支推送到 `origin` 伺服器上時（再次說明，克隆時通常會自動地幫你設定好 `master` 和 `origin` 這二個名稱），那麼你可以執行這個命令將所有你完成的提交（commit）推�回伺服器上。

```
$ git push origin master
```

只有在你對克隆來源的伺服器有寫入權限，並且在這個當下還沒有其它人推送過，這個命令才會成功；如果你和其它人同時做了克隆，然後他們先推送到上游，接著換你推送到上游，毫無疑問地你的推送會被拒絕；你必需先獲取他們的工作內容，將其整併到你之前的工作內容，如此你才會被允許推送。閱讀 [使用 Git 分支](#)

以了解更多關於「如何推送到遠端伺服器」的詳細資訊。

檢視遠端

如果你想要對一個特定遠端檢視更多資訊，你可以使用 `git remote show [remote-name]` 命令。如果你在執行這個命令中使用特定的簡稱，例如 `origin`，你會得到下面這個類似的訊息：

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                  tracked
    dev-branch              tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

它同時列出了遠端版本庫的網址和「追蹤分支（tracking branch）」資訊。這個命令很有用地告訴你：目前分支是 `master`（譯註：`HEAD` 意味著目前的），如果你執行 `git pull`，它會在獲取所有遠端參照之後，自動將遠端的 `master` 合併到你的 `master` 分支。它也會列出所有已抓下來的遠端參照（譯註：此例中指「`master tracked`」和「`dev-branch tracked`」）。

這是一個你很可能會遇到的簡單例子；然而，當你更重度地使用 Git 後，你將會從 `git remote show` 中看到更多的資訊：

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                  tracked
    dev-branch              tracked
    markdown-strip          tracked
    issue-43                new (next fetch will store in
remotes/origin)
    issue-45                new (next fetch will store in
remotes/origin)
    refs/remotes/origin/issue-11   stale (use 'git remote prune' to
remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master      merges with remote master
  Local refs configured for 'git push':
    dev-branch           pushes to dev-branch
  (up to date)
    markdown-strip        pushes to markdown-strip
  (up to date)
    master                pushes to master
  (up to date)
```

這個命令顯示了當你在特定的分支上執行 `git push` 時，它將自動推送到哪一個遠端分支；它也顯示了：哪些遠端分支是在你的本地端還沒有的（譯註：new 屬性）、哪些你曾獲取過的遠端分支已經在遠端上被移除了（譯註：stale 屬性）、哪些本地分支是有能力在執行 `git pull` 後自動和它們的遠端追蹤分支合併。

移除或重新命名遠端

你可以執行 `git remote rename` 來重新命名遠端的簡稱。例如：如果你想要將 `pb` 重新命名為 `paul`，你可以這樣使用 `git remote rename`：

```
$ git remote rename pb paul
$ git remote
origin
paul
```

值得一提的是：它會更新所有遠端追蹤分支的名稱；曾經用來被參考的遠端追蹤分支 `pb/master` 現在改名為 `paul/master`。

如果你因為某些原因想要移除一個遠端——你搬動了伺服器、或者不再使用某個特定的鏡像、或者某個貢獻者不再貢獻了——你可以執行 `git remote rm`：

```
$ git remote rm paul
$ git remote
origin
```

標籤

跟大多數的版本管理系統一樣，Git 有能力對專案歷史中比較特別的時間點貼標籤，來表示其重要性。通常大家都會用這個功能來標出發行版本，如 `v1.0`…等等。在這個章節中，你將會學到如何列出所有的標籤，如何建立新的標籤和各種不同的標籤類型。

列出你的標籤

想要列出 Git 中所有標籤的方法非常直覺。只要輸入 `git tag` 如下：

```
$ git tag
v0.1
v1.3
```

這個指令將依字母序列出所有標籤；雖然說標籤用什麼方式列出不是很重要。

你也可以使用特定的 pattern 來搜尋標籤。舉例來說，在 Git 原始碼的版本庫中，已經包含了超過 500 個標籤。如果你只想看到 1.8.5 系列的標籤，你可以執行以下指令：

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

建立新的標籤

Git 主要使用兩種類型的標籤：輕量級標籤和有註解的標籤。

一個輕量級的標籤就像是一個不會移動的分支——這個標籤只會指向一個特定的提交。

然而，有註解的標籤，會在 Git 的資料庫中儲存成完整的物件。它們將被計算校驗碼；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。通常建議你可以建立一個有註解的標籤，以便你可以保留跟這個標籤有關的所有資訊；但是你如果只想要一個暫時的標籤，或是因為某些原因不想保留額外的資訊，你也可以只用輕量級標籤。

有註解的標籤

建立一個有註解的標籤很簡單。最簡單的方法是在你建立標籤時，同時指定 `-a` 的選項如下：

```
$ git tag -a v1.4 -m "my version 1.4"  
$ git tag  
v0.1  
v1.3  
v1.4
```

指令中的 `-m` 選項後面同時指定了一个標籤訊息，這個訊息會和這個標籤一起保存。如果你沒有為標籤指定一個訊息，那麼 Git 會開啟你的編輯器以便你輸入。

當你使用 `git show` 指令時，你可以查看標籤的資訊，還有這個標籤所標記的提交資訊如下：

```
$ git show v1.4  
tag v1.4  
Tagger: Ben Straub <ben@straub.cc>  
Date: Sat May 3 20:19:12 2014 -0700  
  
my version 1.4  
  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon <schacon@gee-mail.com>  
Date: Mon Mar 17 21:52:11 2008 -0700  
  
changed the version number
```

這樣就可以在提交資訊前顯示出標籤的資訊、標籤被建立的時間以及標籤的訊息。

輕量級標籤

另外一種能標記提交的標籤是輕量級標籤。這基本上是把該提交的校驗碼存在一個檔案中，不包含其他資訊。如果想要建立一個輕量級的標籤，不要指定 `-a`、`-s` 或 `-m` 的選項如下：

```
$ git tag v1.4-lw  
$ git tag  
v0.1  
v1.3  
v1.4  
v1.4-lw  
v1.5
```

此時如果對該標籤使用 `git show`，你將不會看到這個標籤的額外資訊。這個指令就只會顯示標籤所在的提交資訊：

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

對以前的提交貼標籤

你也可以對過去的提交貼標籤。假設你的提交歷史看起來如下：

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfe66ff742fcbe added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fce02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

現在，假設你忘記在專案的「updated rakefile」提交貼 v1.2 的標籤。你可以在後來再補貼標籤。要在那個提交上面貼標籤，你需要在指令後面指定那個提交的校驗碼（可以省略後半段）：

```
$ git tag -a v1.2 9fce02
```

你可以看到你已經在那個提交上面貼標籤了：

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fcceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

updated rakefile
...
```

分享標籤

`git push` 指令預設不會傳送標籤到遠端伺服器。在你建立標籤後，你必須明確的要求 Git 將標籤推送到共用的伺服器上面。這個動作就像是在分享遠端分支一樣——你可以執行 `git push origin [tagname]`。

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]           v1.5 -> v1.5
```

如果想要一次推送很多標籤，也可以在使用 `git push` 指令的時候加上 `--tags` 選項。這將會把你所有不在伺服器上面的標籤傳送給遠端伺服器。

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]           v1.4 -> v1.4
 * [new tag]           v1.4-lw -> v1.4-lw
```

現在，當其他人從版本庫克隆或拉取時，他們就能同時拿到你所貼的標籤，

檢出標籤

在 Git 中你不能真的檢出一個標籤，因為它們並不能像分支一樣四處移動。如果你希望工作目錄和版本庫中特定的標籤版本完全一樣，你可以使用 `git checkout -b [branchname] [tagname]` 在該標籤上建立一個新分支：

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

當然，如果在建立新分支以後又進行了一次提交，`version2` 分支將會和 `v2.0.0` 標籤有所差異，因為這個分支已經因為你的提交而改變了，請特別小心。

Git Aliases

在結束「Git 基礎」這個章節以前，在此想和你分享一些使用 Git 的技巧，讓你能夠更簡易且友善的使用 Git——別名（alias）。在本書的後面章節，我們不會再提到，也不會假設你有使用別名的技巧。但是你可能會需要知道如何使用它。

如果你只打了某個指令的一部份，Git 並不會自動推測出你想要的指令。如果你懶得輸入完整的 Git 指令，你可以輕易的使用 `git config` 來替指令設定別名。下面有一些你可能會想要設定別名的範例：

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

舉其中一個例子來說，這樣的設定意味著你可以只打 `git ci` 而不需要打 `git commit`。隨著你深入使用 Git，你將會發現某些指令用的很頻繁，不要猶豫，馬上建立新的指令別名。

這個非常有用的技術還能用來創造一些你覺得應該存在的指令。舉例來說，為了提高 `unstage` 檔案的方便性，你可以加入你自己的 `unstage` 別名：

```
$ git config --global alias.unstage 'reset HEAD --'
```

而且這個 `unstage` 別名會讓以下兩個指令有相同的功用：

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

這樣看起來更加簡單明瞭了。此外，大家通常還會新增一個 `last` 指令如下：

```
$ git config --global alias.last 'log -1 HEAD'
```

如此一來，你可以更簡易的看到最後的提交訊息：

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

Signed-off-by: Scott Chacon <schacon@example.com>
```

如你所見，Git 會將別名直接取代成你別名內設定的指令。然而，你可能會想要執行一個外部指令，而非 git 下的子指令。在這個情況下，你需要在指令的開頭加個 ! 字元。這個技巧在你如果想為 Git 倉儲撰寫自製工具時很有用。我們可以用以下的範例設定 `git visual` 執行 `gitk`：

```
$ git config --global alias.visual '!gitk'
```

總結

現在，您可以完成所有基本的 Git 本地操作——創建或者克隆一個倉儲、修改檔案、預存並提交這些更改、瀏覽倉儲過去的所有更改歷史。下一步，本書將介紹 Git 的殺手級功能：Git 的分支模型。

使用 Git 分支

幾乎每一種版本控制系統（Version Control System，以下簡稱 VCS）都支援某種形式的分支（branch）功能，使用分支意味著你可以從開發主線上分離開來，然後在不影響主線的情況下繼續工作；在很多 VCS 中，這是個昂貴的過程，常常需要對原始程式碼目錄建立一個完整的副本，對大型專案來說會花費很長時間。

有人把 Git 的分支模型視為它的「殺手級功能」，正是因為它而讓 Git 在 VCS 社群中顯得與眾不同。它有何特別之處呢？Git 的分支簡直是難以置信的羽量級，新建分支的操作幾乎可以在瞬間完成，並且一般來說切換不同分支也很快；跟其它的 VCS 不一樣的地方是 Git 鼓勵在工作流程中頻繁地使用分支與合併（merge），即使一天之內進行許多次都沒問題。理解並掌握這個特性後，它會給你一個強大而獨特的工具，從此完全地改變你的開發方式。

簡述分支

為了理解 Git 分支（branch）的使用方式，我們需要回顧一下 Git 是如何保存資料的。

或許你還記得 [開始](#) 的內容，Git 保存的不是變更集或者差異內容，而是一系列快照。

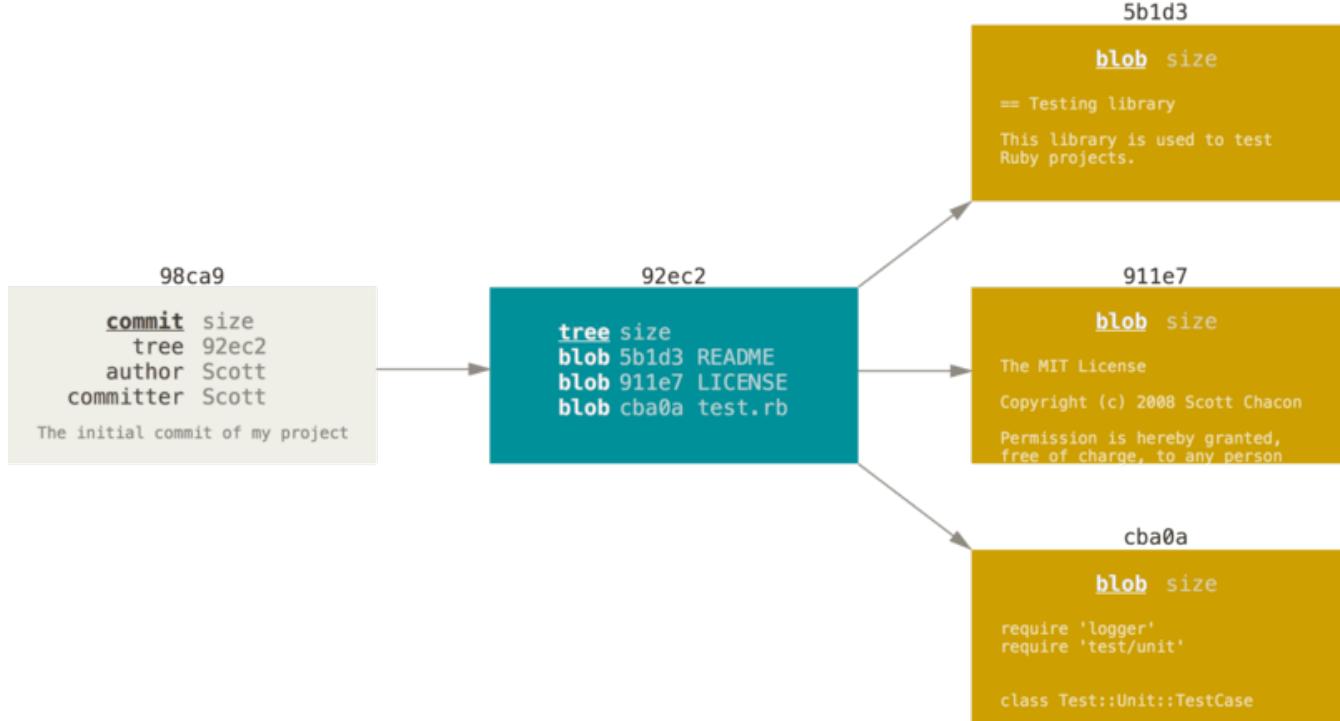
當你製造一個提交（commit）時，Git 會儲存一個提交物件，該物件內容包含一個指標，用來代表已預存的快照內容；這個物件內容還包含「作者名字和電子郵件」、「你輸入的訊息內容」、「指向前一個提交的指標（該提交的親代提交）」：沒有親代（parent）提交表示它是初始的第一個提交，一般情況下只有一個親代提交，超過一個親代提交表示它是從二個以上的分支合併而來的。

為了具體說明，讓我們假設你有一個目錄包含了三個檔案，你預存（stage）並提交了它們；檔案預存操作會對每一個檔案內容（譯註：請注意，只有檔案「內容」）計算雜湊值（即 [開始](#) 中提到的 SHA-1 雜湊值），然後把那個檔案內容版本保存到 Git 版本庫中（Git 把它們視為 blob 類型的物件），再將這個雜湊值寫入預存區（staging area）：

```
$ git add README test.rb LICENSE  
$ git commit -m 'The initial commit of my project'
```

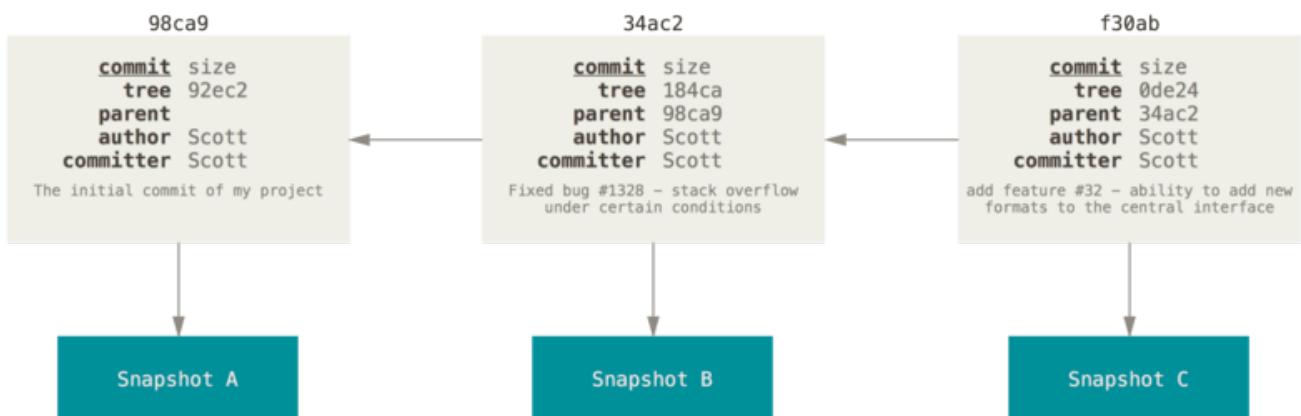
當使用 `git commit` 建立一個提交時，Git 會先計算每一個子目錄（本例中則只有專案根目錄）的雜湊值，然後在 Git 版本庫中將這些目錄記錄為樹（tree）物件；之後 Git 建立提交物件，它除了包含相關提交資訊以外，還包含著指向專案根目錄的樹物件指標，如此它就可以在需要的時候重建此次快照內容。

你的 Git 版本庫現在有五個物件：三個 blob 物件用來儲存檔案內容、一個樹物件用來列出目錄的內容並紀錄各個檔案所對應的 blob 物件、一個提交用來記錄根目錄的樹物件和其他提交資訊。



圖表 9. 單個提交在版本庫中的資料結構

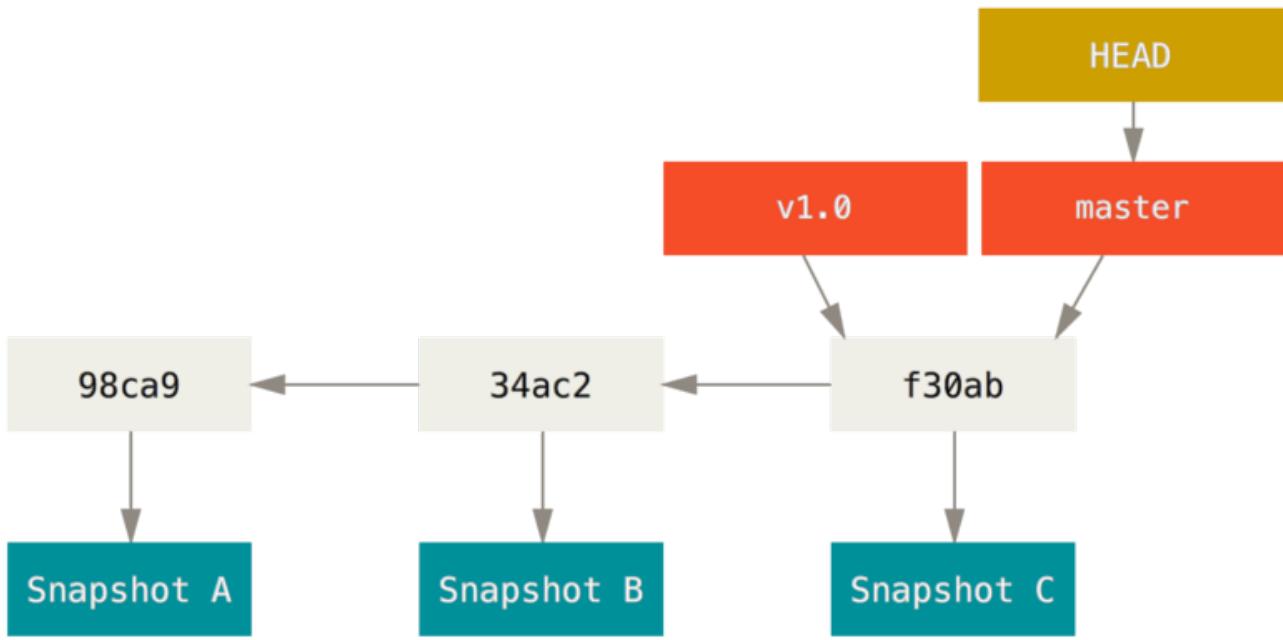
如果你做一些修改並再次提交，這次的提交會再包含一個指向上次提交的指標（譯注：即下圖中的 parent 欄位）。



圖表 10. 提交和它們的親代提交

Git 分支其實只是一個指向某提交的可移動輕量級指標，Git 預設分支名稱是 `master`，隨著不斷地製作提交，`master` 分支會為你一直指向最後一個提交，它在每次提交的時候都會自動向前移動。

筆記 「`master`」在 Git 中並不是一個特殊的分支，它和其它分支並無分別，之所以幾乎每個版本庫裡都會有這個分支的原因是 `git init` 命令的預設行為會產生它，而大部分的人就這麼直接使用它。



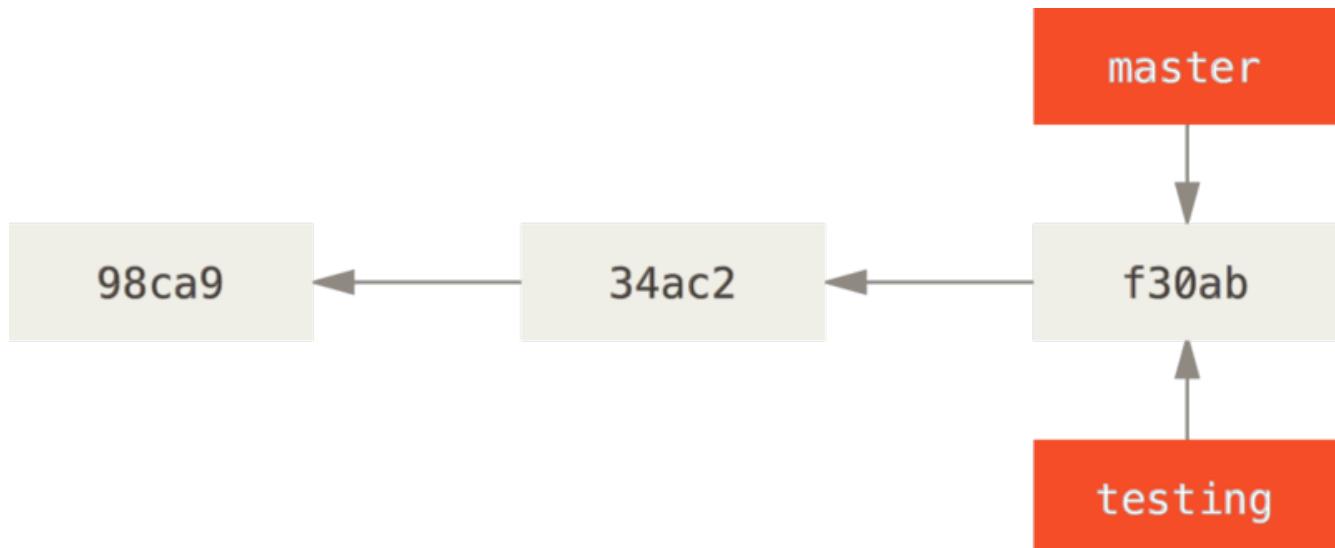
圖表 11. 分支及其提交歷史

建立一個新的分支

建立一個新分支會發生什麼事呢？答案很簡單，建立一個新的、可移動的指標；比如新建一個 testing 分支，可以使用 `git branch` 命令：

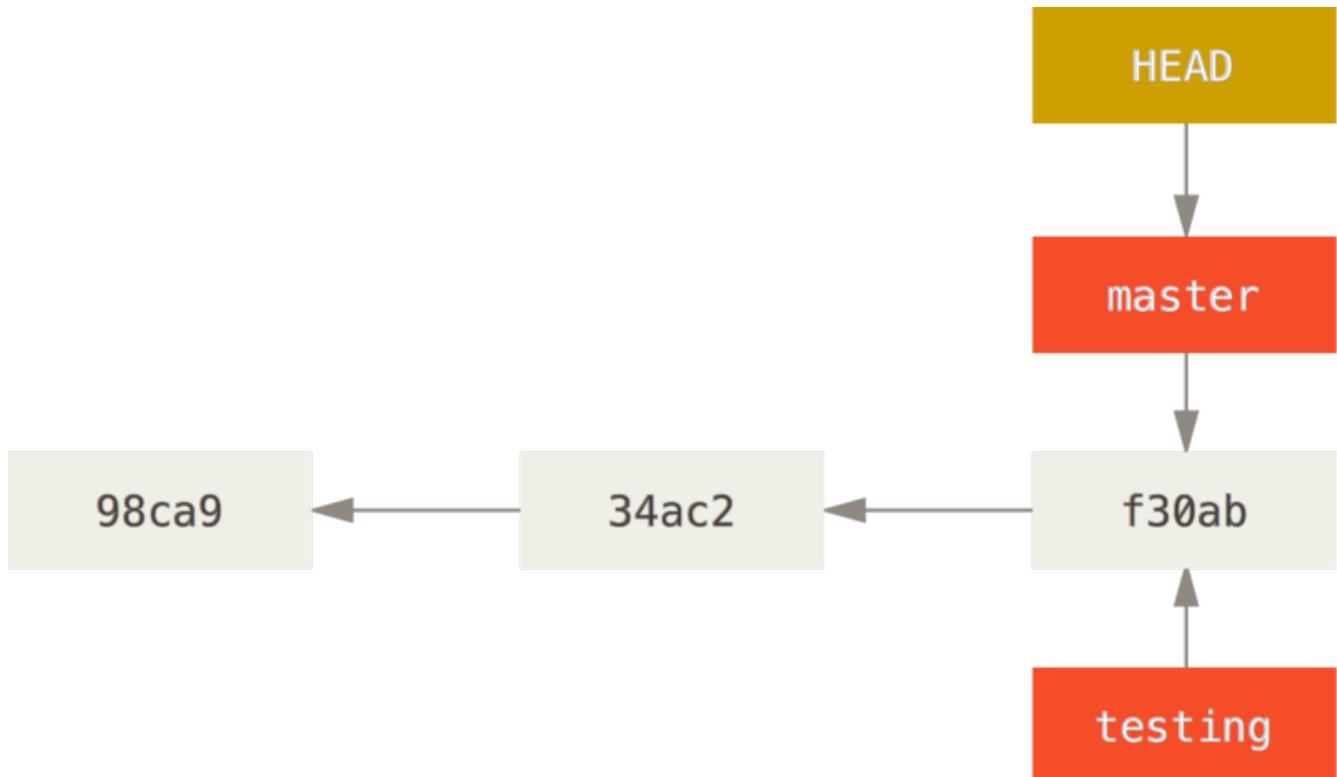
```
$ git branch testing
```

這會在目前提交上新建一個指標。



圖表 12. 二個分支都指向同一系列的提交歷史

Git 如何知道你目前在哪個分支上工作的呢？其實它保存了一個名為 `HEAD` 的特別指標；請注意它和你可能慣用的其他 VCSs 裡的 `HEAD` 概念大不相同，比如 Subversion 或 CVS；在 Git 中，它就是一個指向你正在工作中的本地分支的指標（譯注：`HEAD` 等於「目前的」），所以在這個例子中，你仍然在 `master` 分支上工作；執行 `git branch` 命令，只是「建立」一個新的分支——它並不會切換到這個分支。



圖表 13. HEAD 指向一個分支

你可以很輕鬆地看到分支指標指向何處，只需透過一個簡單的 `git log` 命令，加上 `--decorate` 選項。

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) add feature #32 - ability to add new
formats to the central interface
34ac2 Fixed bug #1328 - stack overflow under certain conditions
98ca9 The initial commit of my project
```

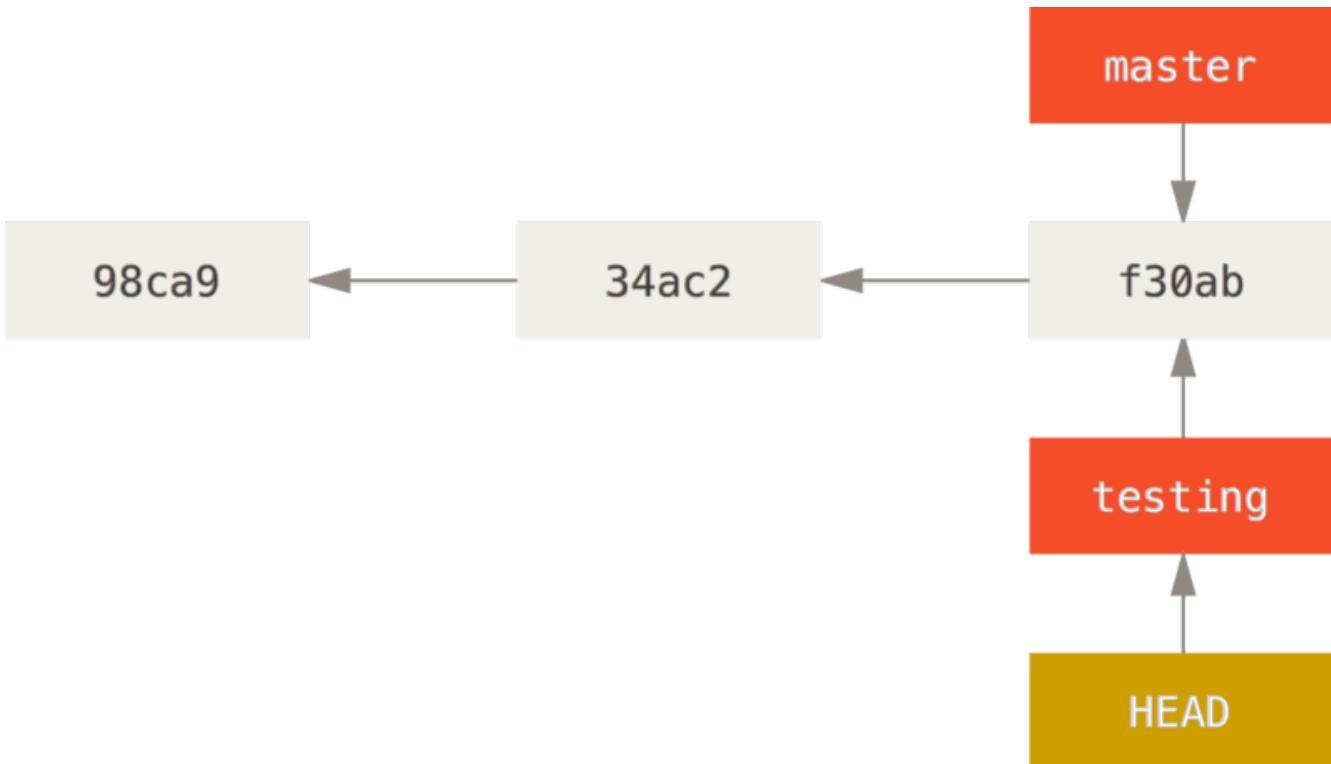
你可以看到「master」和「testing」分支就顯示在 `f30ab` 提交旁邊。

在分支之間切換

要切換到一個已經存在的分支，你可以執行 `git checkout` 命令，讓我們切換到新的 `testing` 分支：

```
$ git checkout testing
```

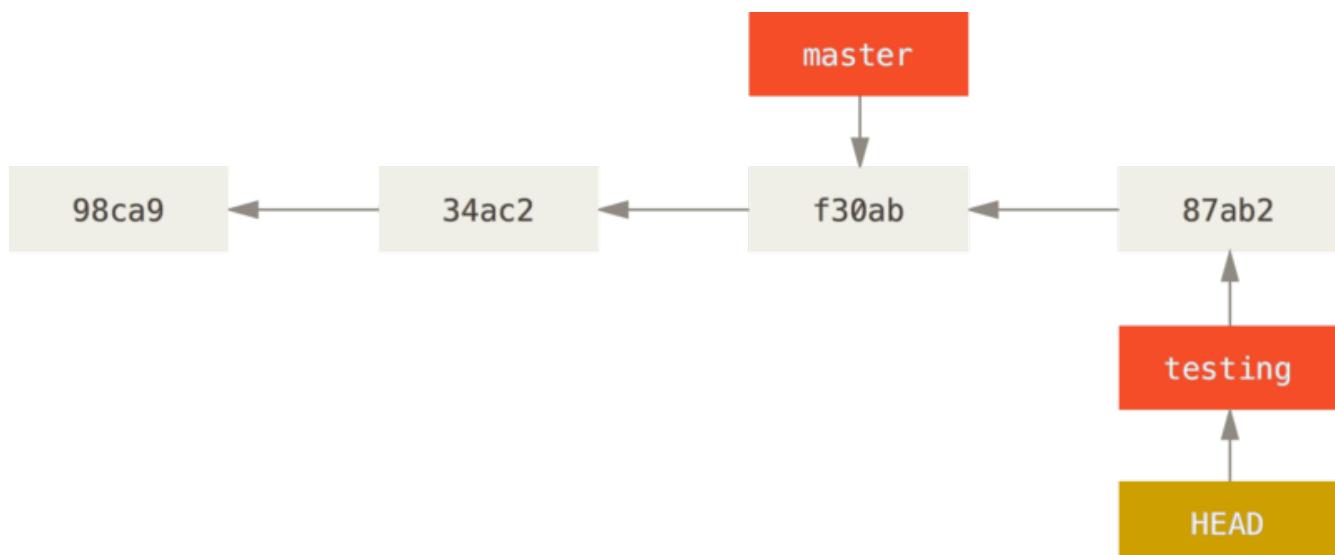
這會移動 `HEAD` 並指向 `testing` 分支。



圖表 14. 被 `HEAD` 指向的分支是目前分支

這樣做有什麼意義呢？好吧！讓我們再提交一次：

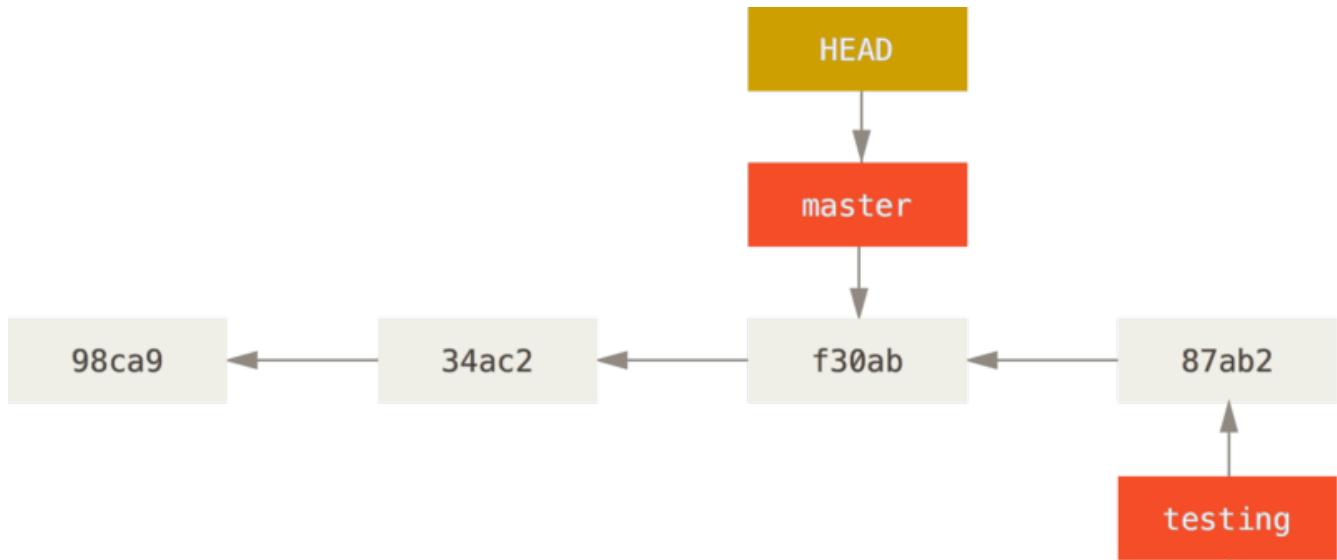
```
$ vim test.rb
$ git commit -a -m 'made a change'
```



圖表 15. 當再次提交時，被 `HEAD` 指向的分支會往前走

非常有趣，現在 `testing` 分支向前移動了，而 `master` 分支仍然指向當初在執行 `git checkout` 時所在的提交，讓我們切回 `master` 分支看看：

```
$ git checkout master
```



圖表 16. 當你檢出時，HEAD 會移動

這條命令做了兩件事，它把 HEAD 指標移回去並指向 `master` 分支，然後把工作目錄中的檔案換成 `master` 分支所指向的快照內容；也就是說，現在開始所做的改動，將基於專案中較舊的版本，然後與其它提交歷史分離開來；它實際上是取消你在 `testing` 分支裡所做的修改，這樣你就可以往不同方向前進。

切換分支會修改工作目錄裡的檔案

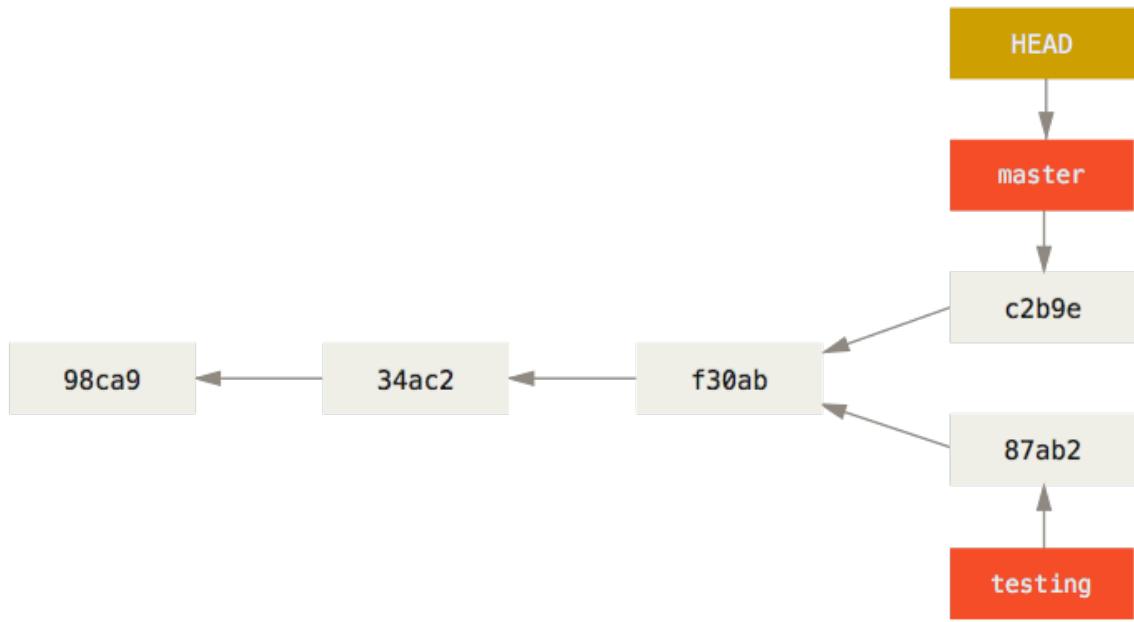
筆記

重要的是要注意：當你在 Git 中切換分支時，工作目錄內的檔案將會被修改；如果切換到舊分支，你的工作目錄會回復到看起來就像當初你最後一次在這個分支提交時的樣子。如果 Git 無法很乾淨地切換過去，它就不會讓你切換過去。

讓我們做一些修改並再次提交：

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

現在你的專案歷史開始分離了（詳見 [分離的歷史](#)）；你建立並切換到新分支，在上面進行了一些工作，然後切換回到主分支進行了另外一些工作，雙方的改變分別隔離在不同的分支裡：你可以在不同分支裡反覆切換，並在時機成熟時把它們合併到一起；而所有這些工作只需要簡單的 `branch`、`checkout`、`commit` 命令。



圖表 17. 分離的歷史

你一樣可以從 `git log` 中輕鬆地看出這件事，執行 `git log --oneline --decorate --graph --all`，它會印出你的提交歷史，顯示你的分支指標在哪裡，以及歷史如何被分離開來。

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

由於 Git 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（40 個字元長度的 SHA-1 字串），所以建立和銷毀一個分支就變得非常廉價；新建一個分支就是向一個檔寫入 41 個位元組（40 個字元外加一個換行符號）那樣地簡單和快速。

這樣的分支功能和大多數舊 VCS 的分支功能形成了鮮明的對比，有些分支功能甚至需要複製專案中全部的檔案到另一個資料夾，而根據專案檔案數量和大小的不同，可能花費的時間快則幾秒，慢則數分鐘；而在 Git 中幾乎都在瞬間完成。還有，因為每次提交時都記錄了親代資訊，將來要合併分支時，它通常會幫我們自動並輕鬆地找到適當的合併基礎；這樣子的特性在無形間鼓勵了開發者頻繁地建立和使用分支。

讓我們來瞧一瞧為什麼你應該要這麼做。

分支和合併的基本用法

讓我們來看一個你在現實生活中，有可能會用到的分支（branch）與合併（merge）工作流程的簡單範例，你做了以下動作：

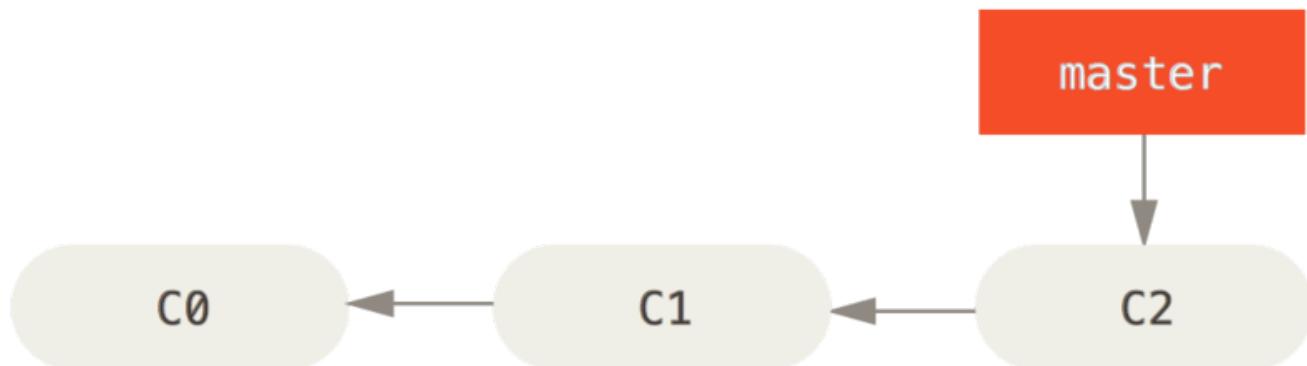
1. 開發一個網站。
2. 建立一個分支以實現一個新故事。
3. 在這個分支上進行開發。

此時你接到一個電話，有個很危急的問題需要緊急修正（hotfix） ，你可以按照下面的方式處理：

1. 切換到發佈產品用的分支。
2. 在同一個提交上建立一個新分支，在這個分支上修正問題。
3. 通過測試後，切回發佈產品用的分支，將修正用的分支合併進來，然後再推送（push）出去以發佈產品。
4. 切換到之前實現新需求的分支以繼續工作。

分支的基本用法

首先，我們假設你正在開發你的專案，並且已經有一些提交（commit）了。



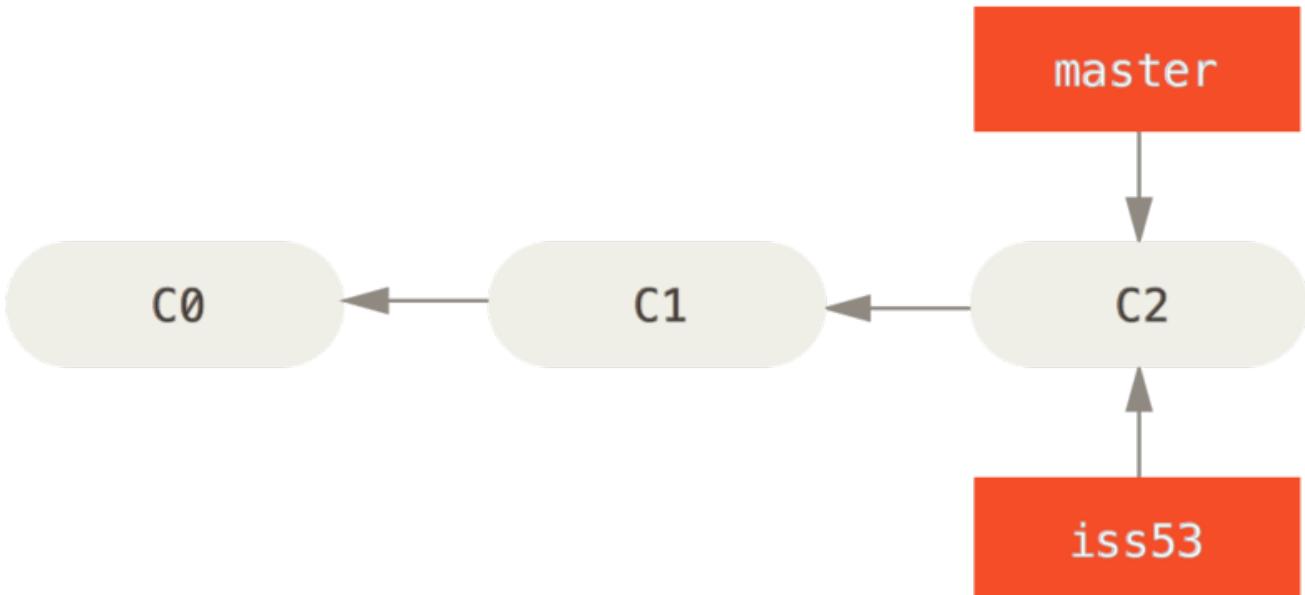
圖表 18. 一個簡單的提交歷史

無論你的公司使用的議題追蹤系統（issue-tracking system）是哪一套，你決定要修正其中的議題 #53；要同時新建並切換到新分支，你可以在執行 `git checkout` 時加上 `-b` 選項：

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

它相當於下面這兩條命令：

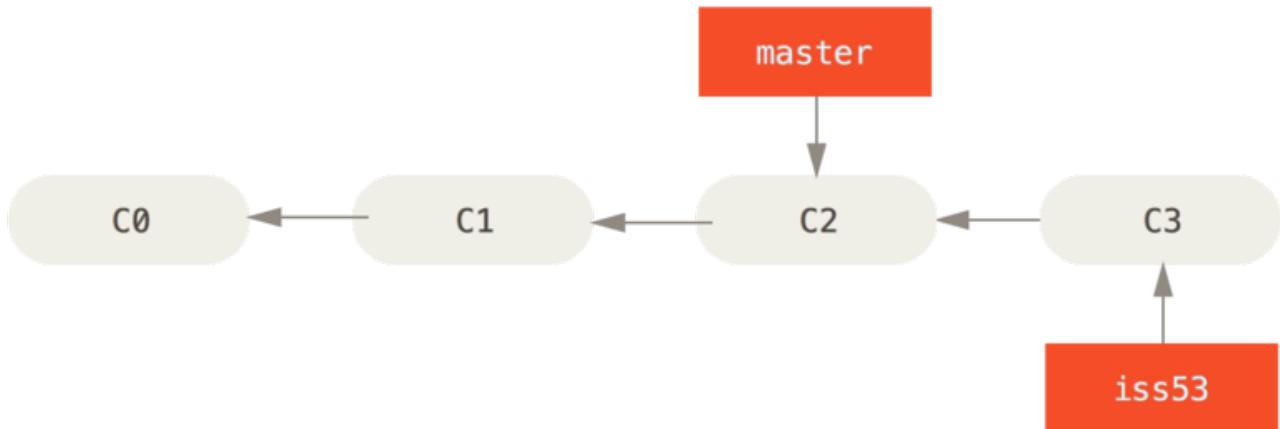
```
$ git branch iss53
$ git checkout iss53
```



圖表 19. 建立一個新分支指標

你開始開發網站，並做了一些提交；因為你檢出（checkout）了這個分支（也就是 HEAD 指標正指向它），`iss53` 分支也隨之向前推進：

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```



圖表 20. 分支 `iss53` 會隨工作進展向前推進

現在你接到電話，那個網站有一個問題需要立即修正；有了 Git，你就不用把你的緊急修正連同 `iss53` 尚未完成的內容一起部署（deploy）到正式環境；你也不用為了正確地套用修正而先花一大堆功夫回復之前 `iss53` 的修改；唯一需要做的只是切換回發佈產品用的 `master` 分支。

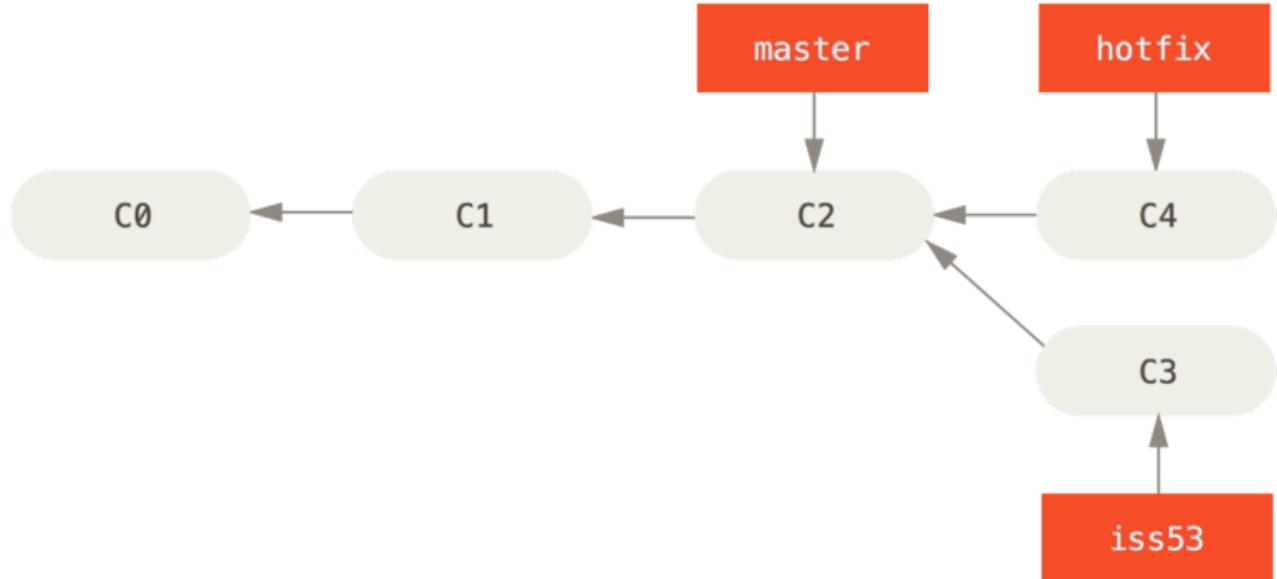
然而，在切換分支之前，留意一下你的工作目錄或預存區（staging area）裡是否有還沒提交的內容，它可能會和你即要檢出的分支產生衝突（conflict），Git 會因此而不讓你切換分支；所以切換分支的時候最好先保持一個乾淨的工作區域。稍後會在 [Stashing and Cleaning](#) 中介紹幾個繞過這種問題的辦法（分別叫做「使用收藏（stashing）」和「提交的修訂方法（commit amending）」）。目前先讓我們假設你已經提交了所有的變更，因此你可以切回 `master` 分支了：

```
$ git checkout master  
Switched to branch 'master'
```

此時工作目錄中的內容和你在解決問題 #53 之前的內容一模一樣，你可以集中精力進行緊急修正了；很重要的一點需要牢記：當你切換分支時，Git 會重置（reset）工作目錄內容，就像回到你在這個分支最後一次提交後的內容，它會自動地增加、刪除和修改檔案以確保工作目錄的內容和當時的內容完全一樣。

接下來開始開始緊急修正；讓我們建立一個緊急修正用的分支來進行工作，直到完成它：

```
$ git checkout -b hotfix  
Switched to a new branch 'hotfix'  
$ vim index.html  
$ git commit -a -m 'fixed the broken email address'  
[hotfix 1fb7853] fixed the broken email address  
1 file changed, 2 insertions(+)
```



圖表 21. 基於 `master` 的緊急修正分支

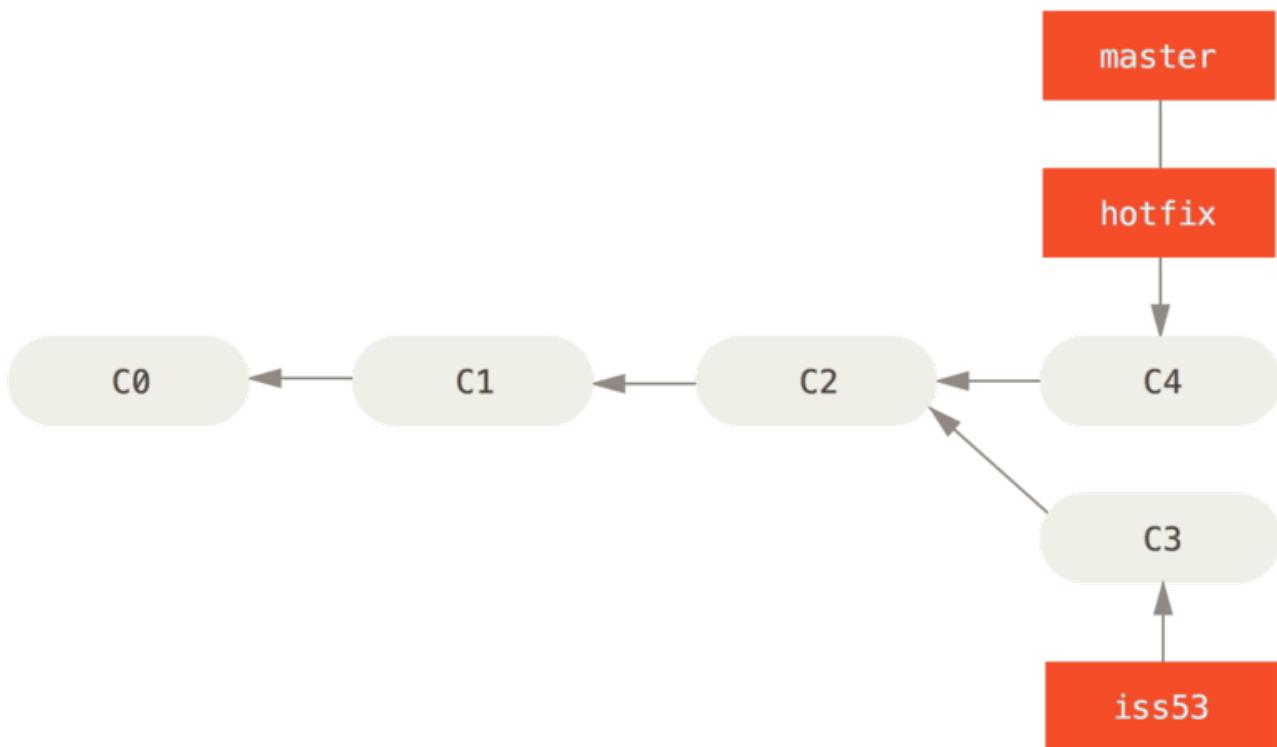
你可以跑一些測試以確保該修正是你想要的，然後切回 `master` 分支並把它合併進來，再部署到產品上；用 `git merge` 命令來進行合併：

```
$ git checkout master  
$ git merge hotfix  
Updating f42c576..3a0874c  
Fast-forward  
 index.html | 2 ++  
 1 file changed, 2 insertions(+)
```

注意合併時有一個「Fast-forward」字眼；由於你要合併的分支 `hotfix` 所指向的提交 `C4` 直接超前了提交 `C2`，Git 於是簡單地把分支指標向前推進；換句話說，如果想要合併的提交可以直接往回追溯歷史到目前所在的提交，Git 會因為沒有需要合併的工作而簡單地把指標向前推進——這就是所謂的「快進（fast-

forward)」。

現在你的修改已經含在 `master` 分支所指向的提交的快照中，你可以部署該修正了。



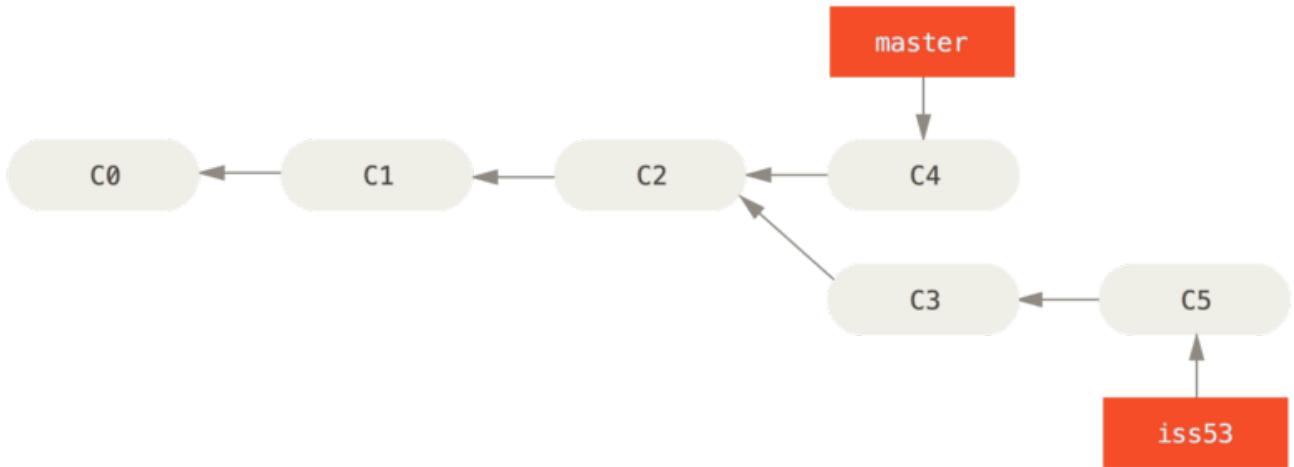
圖表 22. `master` 被快進到 `hotfix`

在那個超級重要的修正被部署以後，你準備要切回到之前被中斷而正在做的工作；然而在那之前，你可以先刪除 `hotfix`，因為你不再需要它了——`master` 也指向相同的提交；使用 `git branch` 的 `-d` 選項執行刪除操作：

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

現在你可以切回到之前用來解決議題 #53 且仍在進展中的分支以繼續工作：

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```



圖表 23. 繼續在分支 `iss53` 上工作

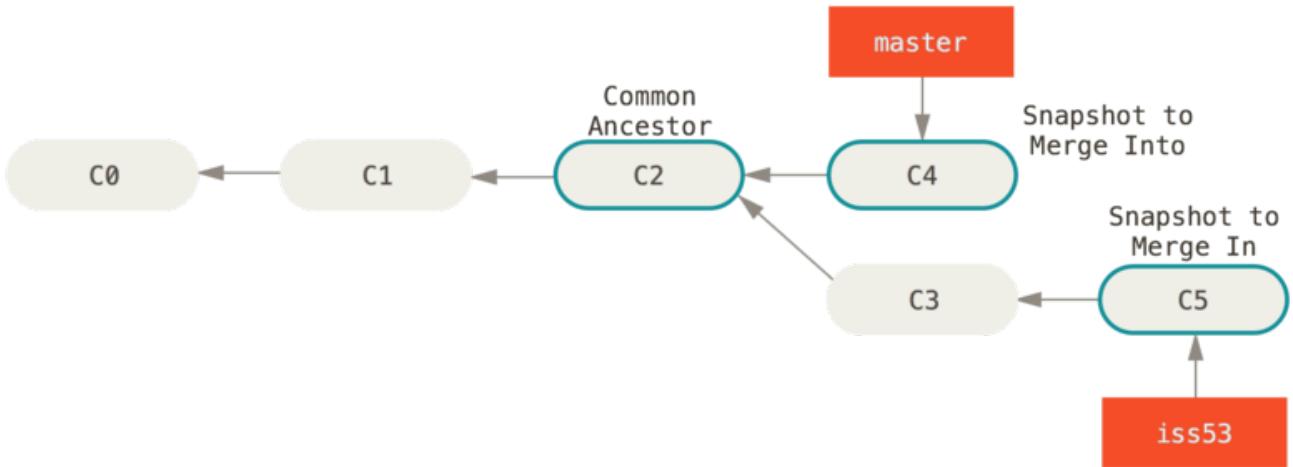
這裡值得注意的是之前 `hotfix` 分支的修改內容尚未包含到 `iss53` 分支的檔案中；如果需要納入那個修正，你可以用 `git merge master` 把 `master` 分支合併到 `iss53` 分支；或者等 `iss53` 分支完成之後，再將它合併到 `master`。

合併的基本用法

你已經完成了議題 #53 的工作，並準備好將它合併到 `master` 分支；要完成這件事，你需要將 `iss53` 分支合併到 `master` 分支，實際操作和之前合併 `hotfix` 分支時差不多，只需切回合併目的地的 `master` 分支，然後執行 `git merge` 命令：

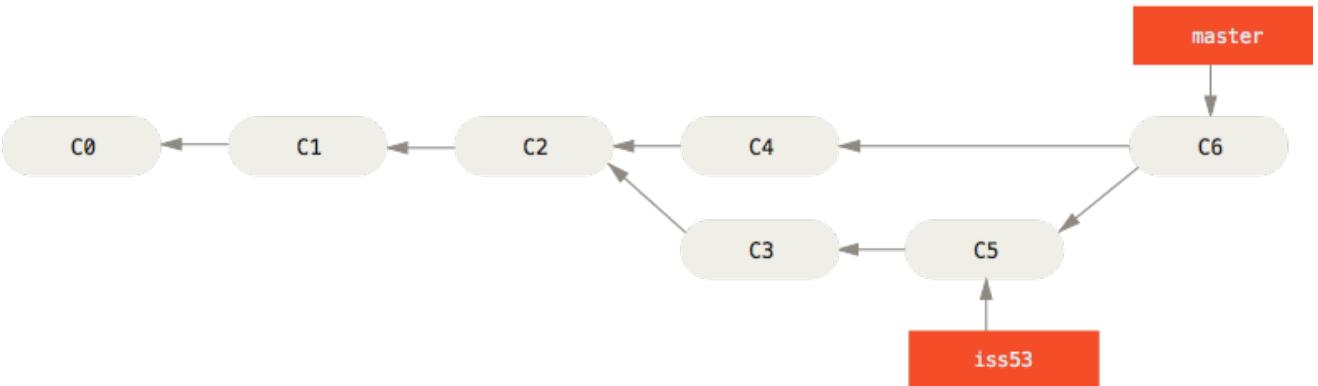
```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

這次的合併和之前合併 `hotfix` 的情況看起來有點不一樣；在這種情況下，你的開發歷史是從一個較早的點便開始分離開來，由於目前所在的提交（譯註：C4）並不是被合併的分支（譯註：`iss53`，它指向 C5）的直接祖先，Git 必需進行一些處理；就此例而言，Git 會用兩個分支末端的快照（譯註：C4、C5）以及它們的共同祖先（譯註：C2）進行一次簡單的三方合併（three-way merge）。



圖表 24. 典型的合併會用到的三個快照

不同於將分支指標向前推進，Git 會對三方合併後的結果產生一個新的快照，並自動建立一個指向這個快照的提交（譯註：C6） 。這個提交被稱為「合併提交（merge commit）」，特別的是它的親代（parent）超過一個（譯註：C4 和 C5）。



圖表 25. 一個合併提交

值得一提的是 Git 會決定哪個共同祖先才是最佳合併基準；這一點和一些較舊的版控工具有所不同，像是 CVS 或 Subversion (1.5 以前的版本) ，它們需要開發者自己手動找出最佳合併基準；這讓 Git 的合併操作比起其他系統都要簡單許多。

既然你的工作成果已經合併了，也就不再需要 iss53 分支了，你可以在議題追蹤系統中關閉該議題，然後刪除這個分支：

```
$ git branch -d iss53
```

合併衝突的基本解法

有時候合併過程並不會如此順利，如果在不同的分支中都修改了同一個檔案的同一部分，Git 就無法乾淨地

合併它們；如果你在解決議題 #53 的過程中修改了 hotfix 中也修改過的部分，將得到類似下面的「合併衝突」結果：

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git 沒有自動產生新的合併提交，它會暫停下來等你解決 (resolve) 衝突；在合併衝突發生後的任何時候，如果你要看看哪些檔案還沒有合併，可以使用 `git status`：

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

它會列出所有有合併衝突且仍未解決的檔案（譯註：列在 `Unmerged paths:` 下面）；Git 會在有衝突的檔案裡加入標準的「衝突解決 (conflict-resolution)」標記，因此你可以手動開啟它們以解決這些衝突；你的檔案會包含類似下面這樣子的區段：

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>> iss53:index.html
```

可以看到 `=====` 隔開的上半部分是 `HEAD` (即 `master` 分支，在執行合併命令前所切換過去的分支) 中的內容，下半部分則是在 `iss53` 分支中的內容；解決衝突的辦法無非是二選一，或者由你自己合併內容；比如你可以把這整段內容替換成以下內容而解決這個衝突：

```
<div id="footer">
  please contact us at email.support@github.com
</div>
```

這個解決方案分別採納了兩個分支中的各一部分內容，並且完整地移除了 `<<<<<`、`=====` 和 `>>>>>` 這些標記行。在解決了每個衝突檔案裡的每個衝突後，對每個檔案執行 `git add` 會將它們標記為已解決狀態，因為預存 (stage) 動作代表了衝突已經解決。

如果你想用圖形介面的工具來解決這些衝突，你可以執行 `git mergetool`，它會呼叫一個適當的視覺化合併工具並引導你解決衝突：

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisediff gvimdiff diffuse
diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
{local}: modified file
{remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

如果不想用預設的合併工具（因為在 Mac 上執行了該命令，Git 預設選擇了 `opendiff`），你可以在「one of the following tools」列表中找到可使用的合併工具，然後只要輸入你想使用的工具名稱即可。

筆記

如果你需要更多進階的工具用來解決刁鑽的合併衝突，我們將在 [Advanced Merging](#) 介紹更多合併操作方法。

退出合併工具以後，Git 會詢問你合併是否成功，如果回答「是」，它會幫你把相關檔案預存起來，將狀態標記為已解決；你可以再次執行 `git status` 來確認所有衝突都已經解決：

```
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:

modified:   index.html
```

如果你滿意這個結果，並且確認了所有衝突都已經解決也預存了，就可以用 `git commit` 來完成這次合併提交；預設的提交訊息看起來像這樣：

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

如果解決衝突的理由不是那麼明顯，或是想要幫助將來的人理解為何你要這樣解決衝突，你可以在訊息中提供更多的細節來說明。

分支管理

到目前為止，你已經建立、合併和刪除過分支（branch）；讓我們再來看一些分支管理工具，這將會在你開始全程使用分支時派上用場。

`git branch` 命令不僅能建立和刪除分支，如果不加任何參數，你將會得到所有分支的簡易清單：

```
$ git branch
  iss53
* master
  testing
```

注意 `master` 分支前面的 `*` 字元，它表示目前所檢出（checkout）的分支（換句話說，`HEAD` 指向這個分支）；這意味著如果你現在提交，`master` 分支將隨之向前移動。若要查看各個分支最後一個提交，執行 `git branch -v`：

```
$ git branch -v
  iss53  93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

`--merged` 和 `--no-merged` 這兩個有用的選項，可以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。使用 `git branch --merged` 來查看哪些分支已被合併到目前分支：

```
$ git branch --merged
  iss53
* master
```

由於之前的 `iss53` 已經被合併了，所以會在列表中看到它；在這個列表中沒有被標記 `*` 的分支通常都可以用 `git branch -d` 刪除；你已經把它們的工作內容整併到其他分支，所以刪掉它們也不會有所損失。

查看所有包含未合併工作的分支，可以運行 `git branch --no-merged`：

```
$ git branch --no-merged
  testing
```

這顯示了你其它的分支；由於它包含了還未合併的工作，嘗試使用 `git branch -d` 刪除該分支將會失敗：

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

如果你確實想要刪除該分支並丟掉那個工作成果，可以用 `-D` 選項來強制執行，就像上面訊息中所提示的。

分支工作流程

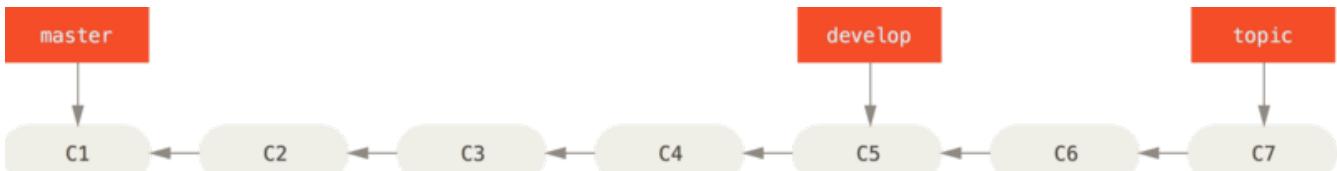
Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate it into your own development cycle.

長期分支

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

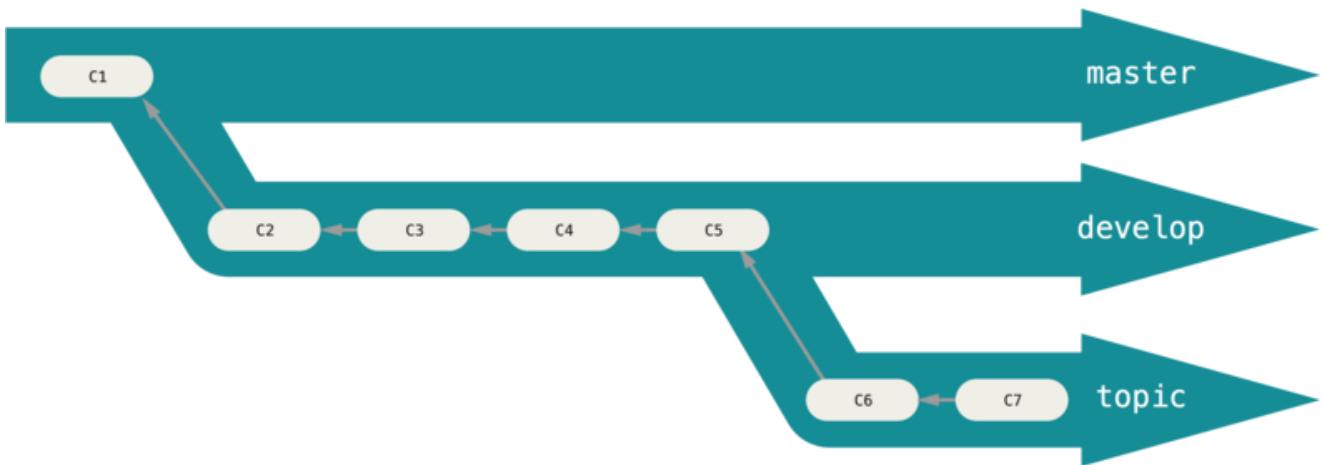
Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their `master` branch – possibly only code that has been or will be released. They have another parallel branch named `develop` or `next` that they work from or use to test stability – it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into `master`. It's used to pull in topic branches (short-lived branches, like your earlier `iss53` branch) when they're ready, to make sure they pass all the tests and don't introduce bugs.

In reality, we're talking about pointers moving up the line of commits you're making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.



圖表 26. A linear view of progressive-stability branching

It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.



圖表 27. A “silo” view of progressive-stability branching

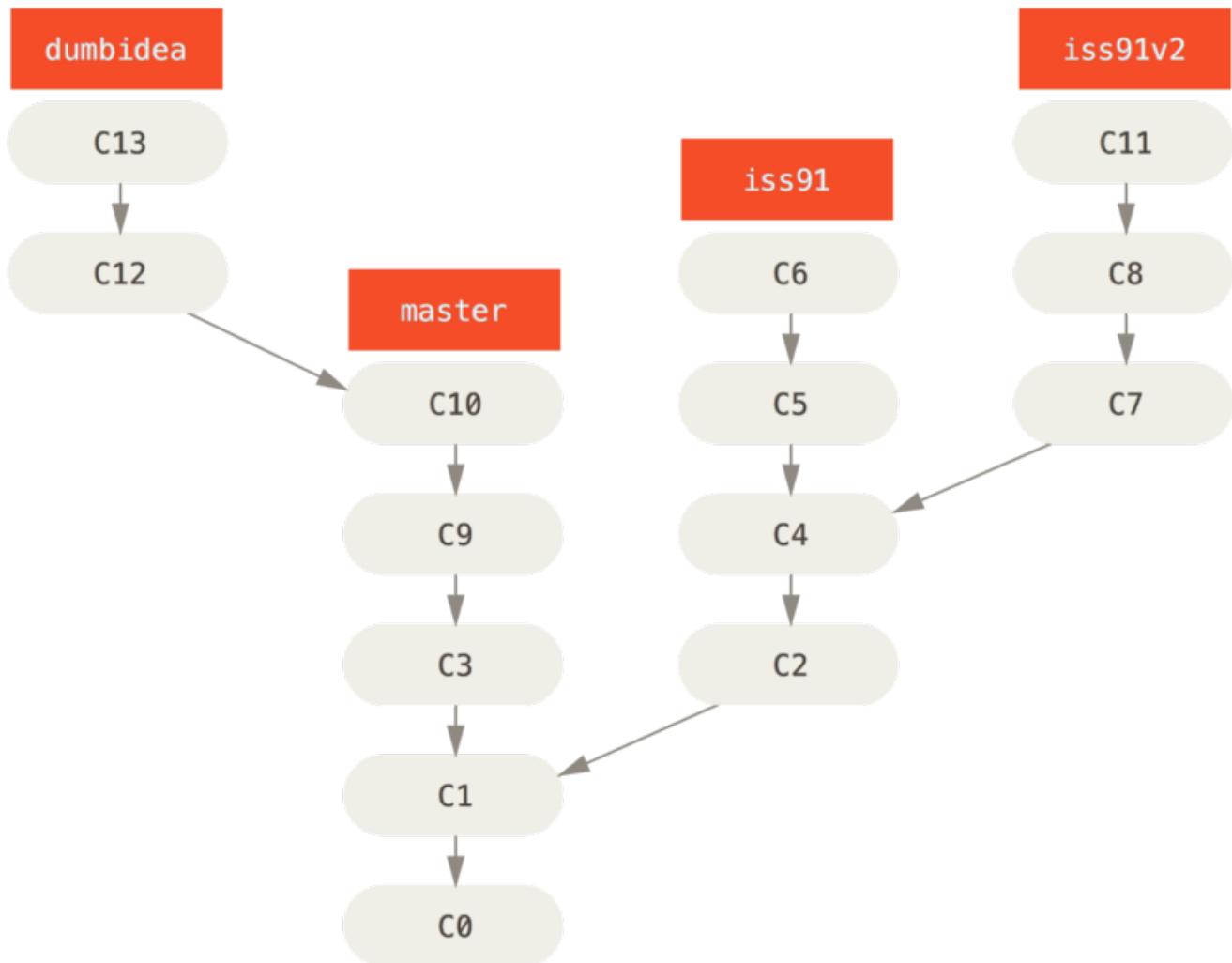
You can keep doing this for several levels of stability. Some larger projects also have a **proposed** or **pu** (proposed updates) branch that has integrated branches that may not be ready to go into the **next** or **master** branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they're merged into the branch above them. Again, having multiple long-running branches isn't necessary, but it's often helpful, especially when you're dealing with very large or complex projects.

主題分支

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. This is something you've likely never done with a VCS before because it's generally too expensive to create and merge branches. But in Git it's common to create, work on, merge, and delete branches several times a day.

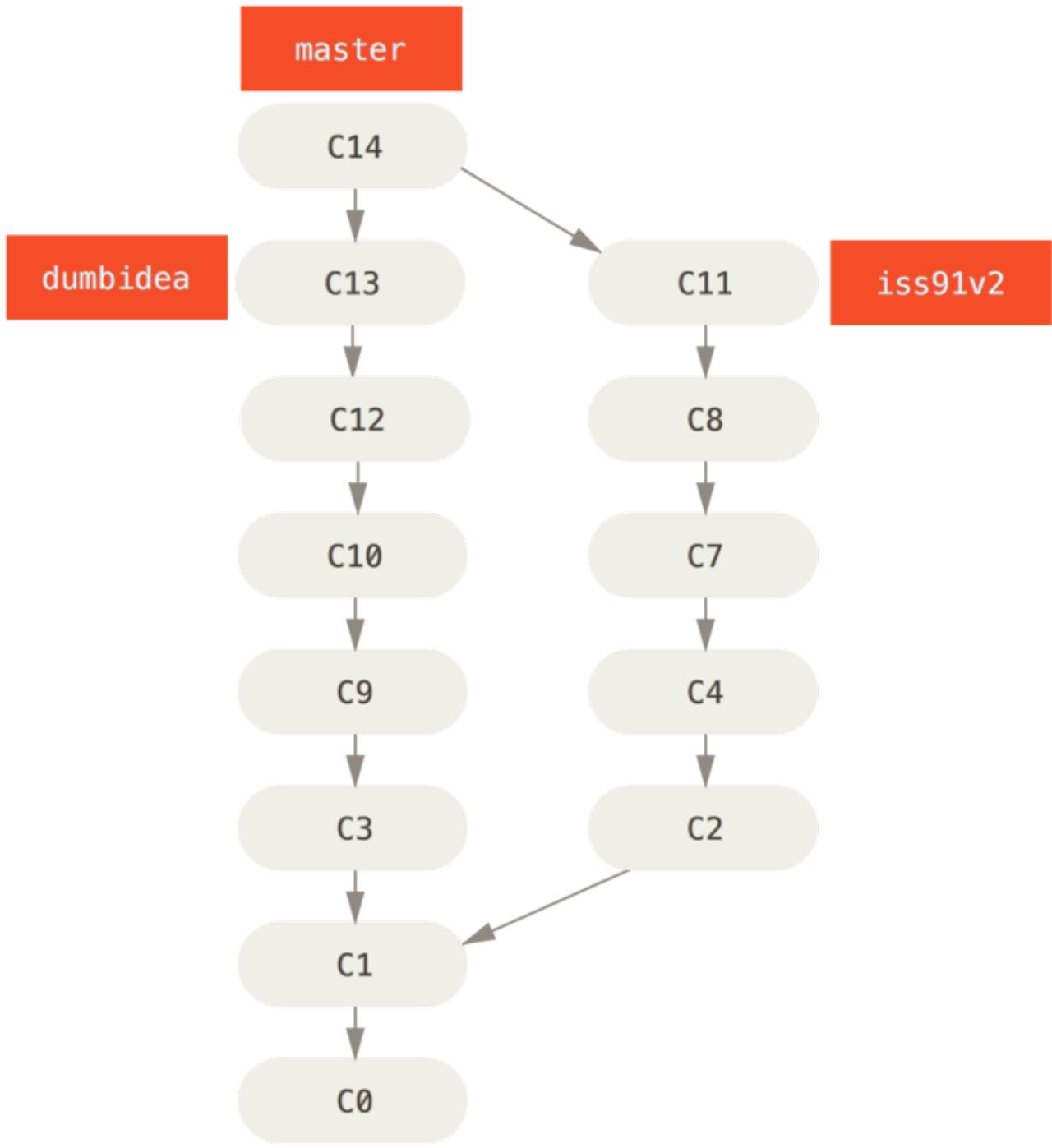
You saw this in the last section with the **iss53** and **hotfix** branches you created. You did a few commits on them and deleted them directly after merging them into your main branch. This technique allows you to context-switch quickly and completely – because your work is separated into silos where all the changes in that branch have to do with that topic, it's easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they're ready, regardless of the order in which they were created or worked on.

Consider an example of doing some work (on **master**), branching off for an issue (**iss91**), working on it for a bit, branching off the second branch to try another way of handling the same thing (**iss91v2**), going back to your **master** branch and working there for a while, and then branching off there to do some work that you're not sure is a good idea (**dumbidea** branch). Your commit history will look something like this:



圖表 28. Multiple topic branches

Now, let's say you decide you like the second solution to your issue best (`iss91v2`); and you showed the `dumbidea` branch to your coworkers, and it turns out to be genius. You can throw away the original `iss91` branch (losing commits `C5` and `C6`) and merge in the other two. Your history then looks like this:



圖表 29. History after merging `dumbidea` and `iss91v2`

We will go into more detail about the various possible workflows for your Git project in [分散式的 Git](#), so before you decide which branching scheme your next project will use, be sure to read that chapter.

It's important to remember when you're doing all this that these branches are completely local. When you're branching and merging, everything is being done only in your Git repository – no server communication is happening.

遠端分支

Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. You can get a full list of remote references explicitly with `git ls-remote [remote]`, or `git remote show [remote]` for remote branches as well as more information. Nevertheless, a more

common way is to take advantage of remote-tracking branches.

Remote-tracking branches are references to the state of remote branches. They’re local references that you can’t move; they’re moved automatically for you whenever you do any network communication. Remote-tracking branches act as bookmarks to remind you where the branches in your remote repositories were the last time you connected to them.

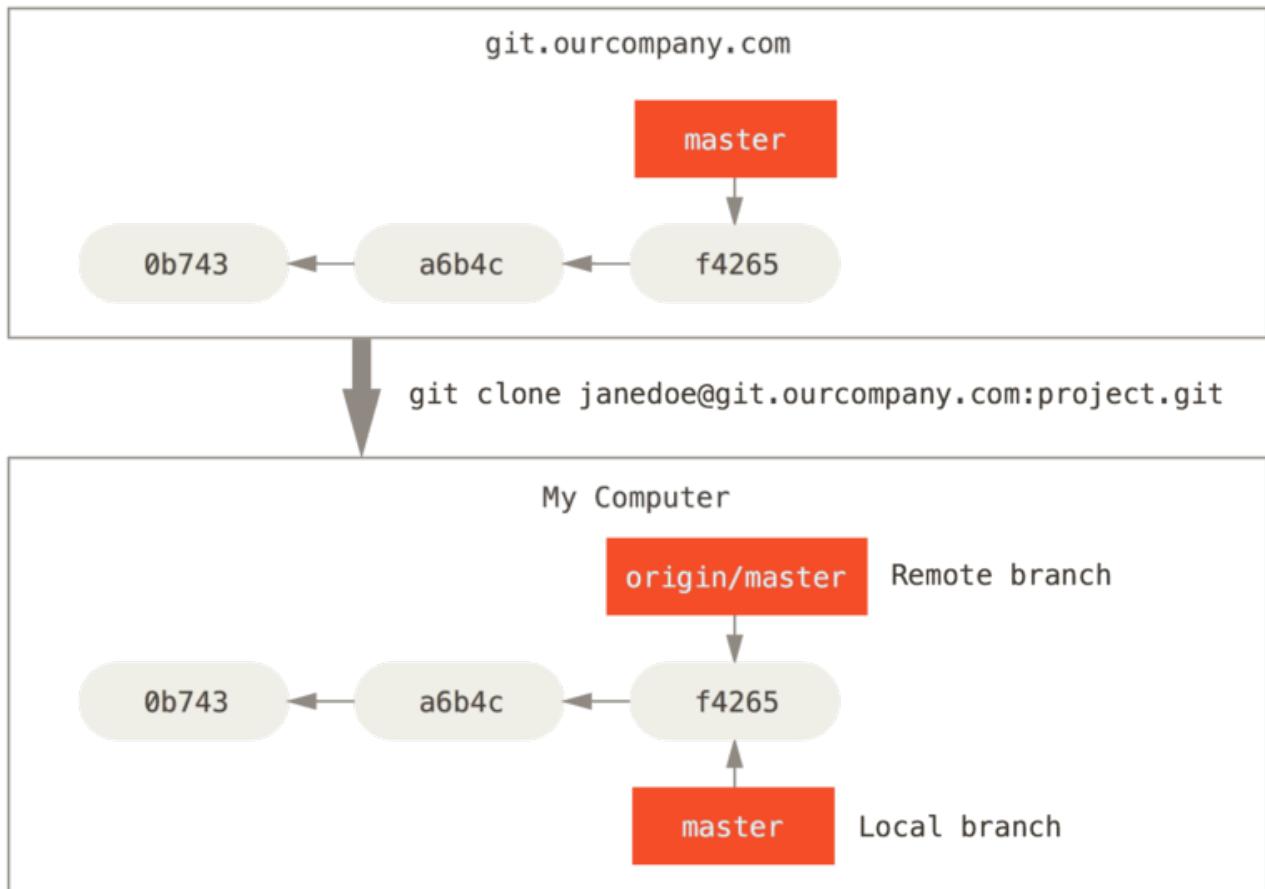
They take the form `(remote)/(branch)`. For instance, if you wanted to see what the `master` branch on your `origin` remote looked like as of the last time you communicated with it, you would check the `origin/master` branch. If you were working on an issue with a partner and they pushed up an `iss53` branch, you might have your own local `iss53` branch; but the branch on the server would point to the commit at `origin/iss53`.

This may be a bit confusing, so let’s look at an example. Let’s say you have a Git server on your network at `git.ourcompany.com`. If you clone from this, Git’s `clone` command automatically names it `origin` for you, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally. Git also gives you your own local `master` branch starting at the same place as origin’s `master` branch, so you have something to work from.

“origin” is not special

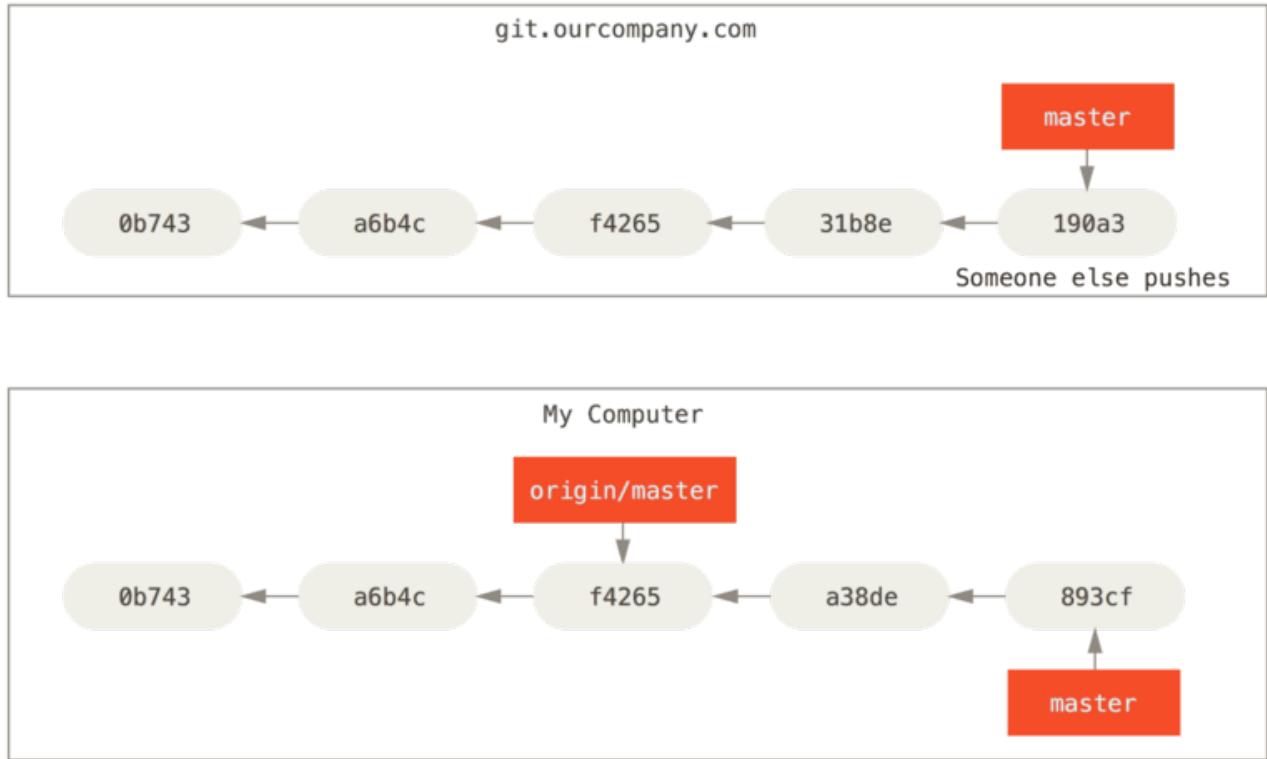
筆記

Just like the branch name “`master`” does not have any special meaning in Git, neither does “`origin`”. While “`master`” is the default name for a starting branch when you run `git init` which is the only reason it’s widely used, “`origin`” is the default name for a remote when you run `git clone`. If you run `git clone -o booyah` instead, then you will have `booyah/master` as your default remote branch.



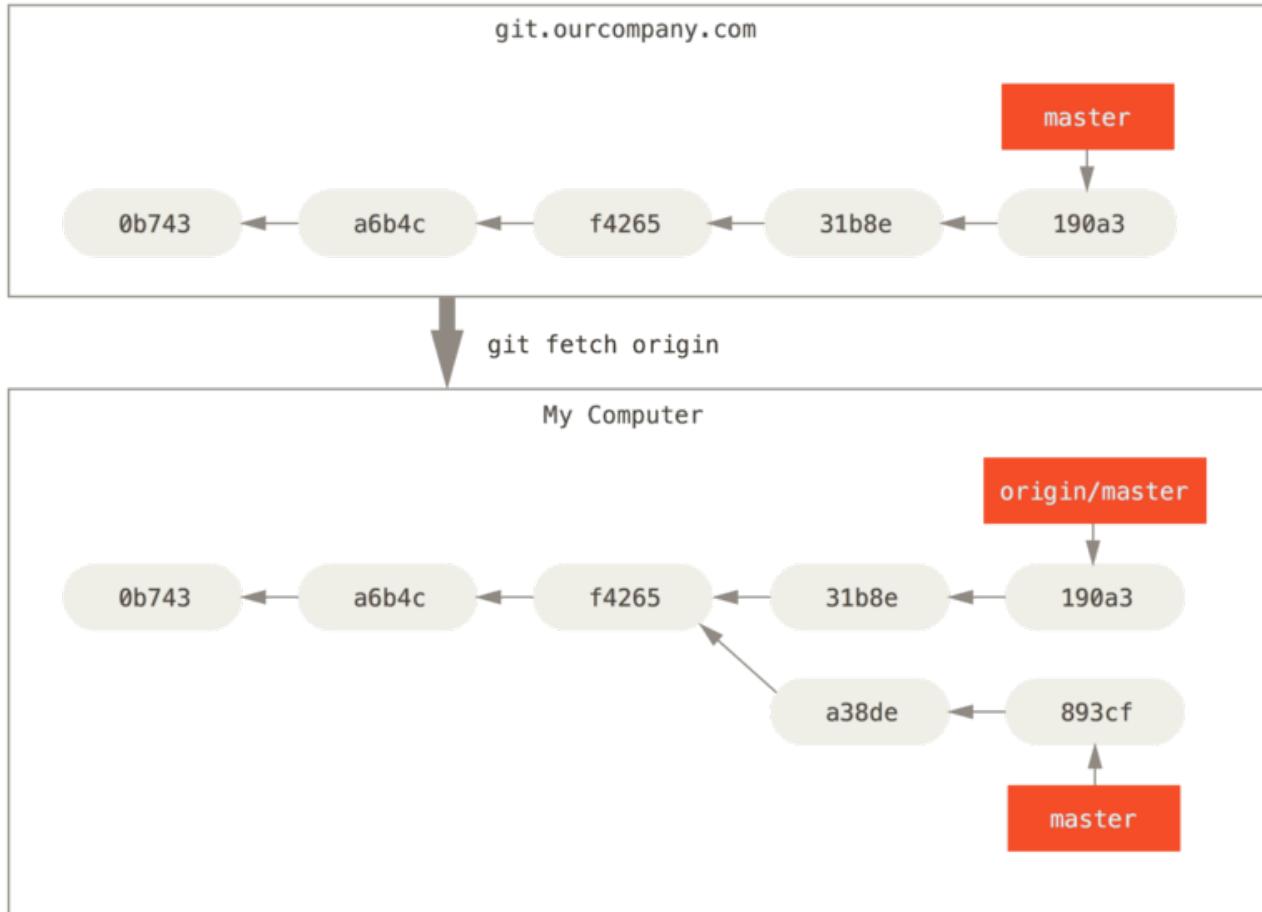
圖表 30. Server and local repositories after cloning

If you do some work on your local `master` branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its `master` branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your `origin/master` pointer doesn't move.



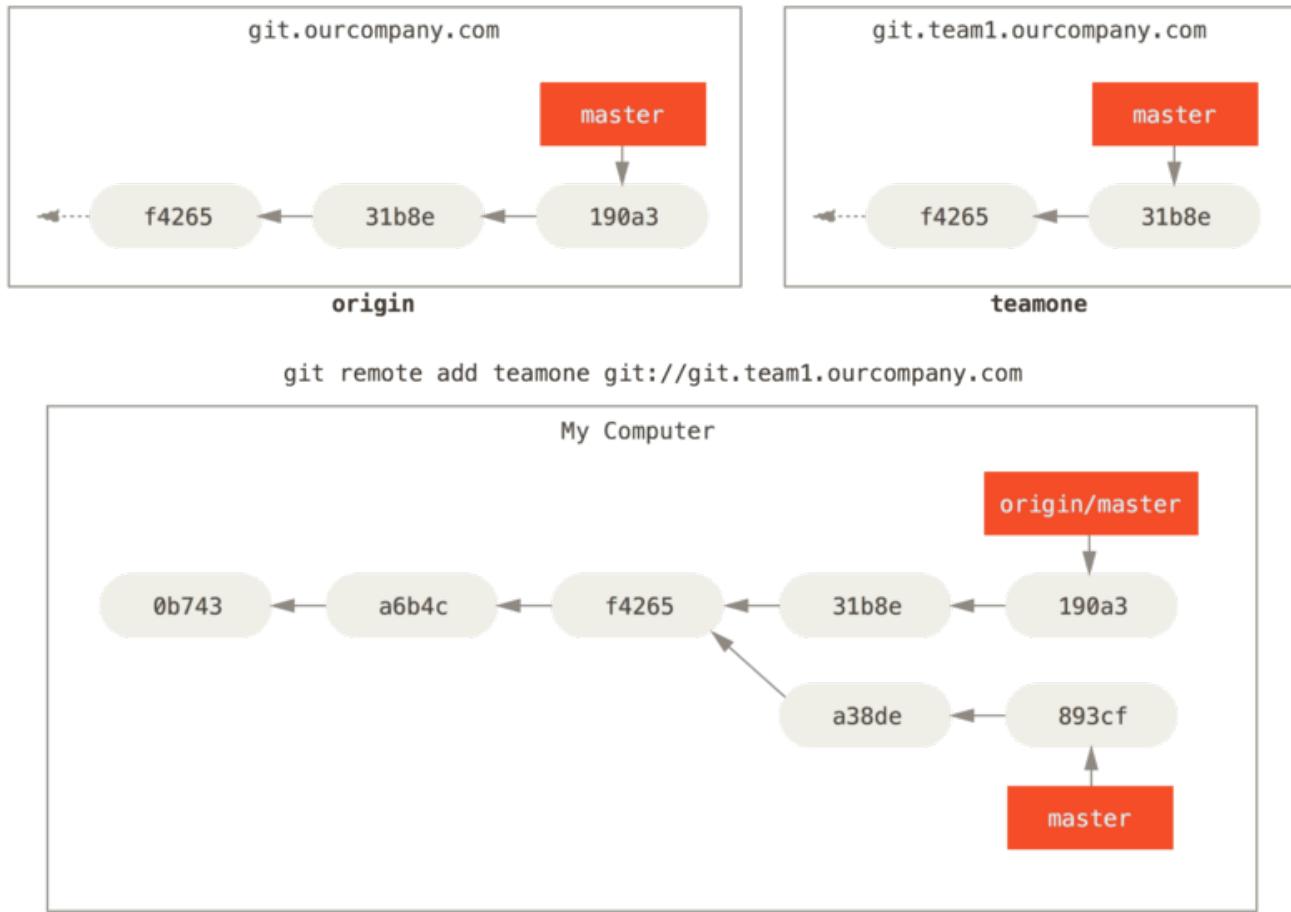
圖表 31. Local and remote work can diverge

To synchronize your work, you run a `git fetch origin` command. This command looks up which server “origin” is (in this case, it’s `git.ourcompany.com`), fetches any data from it that you don’t yet have, and updates your local database, moving your `origin/master` pointer to its new, more up-to-date position.



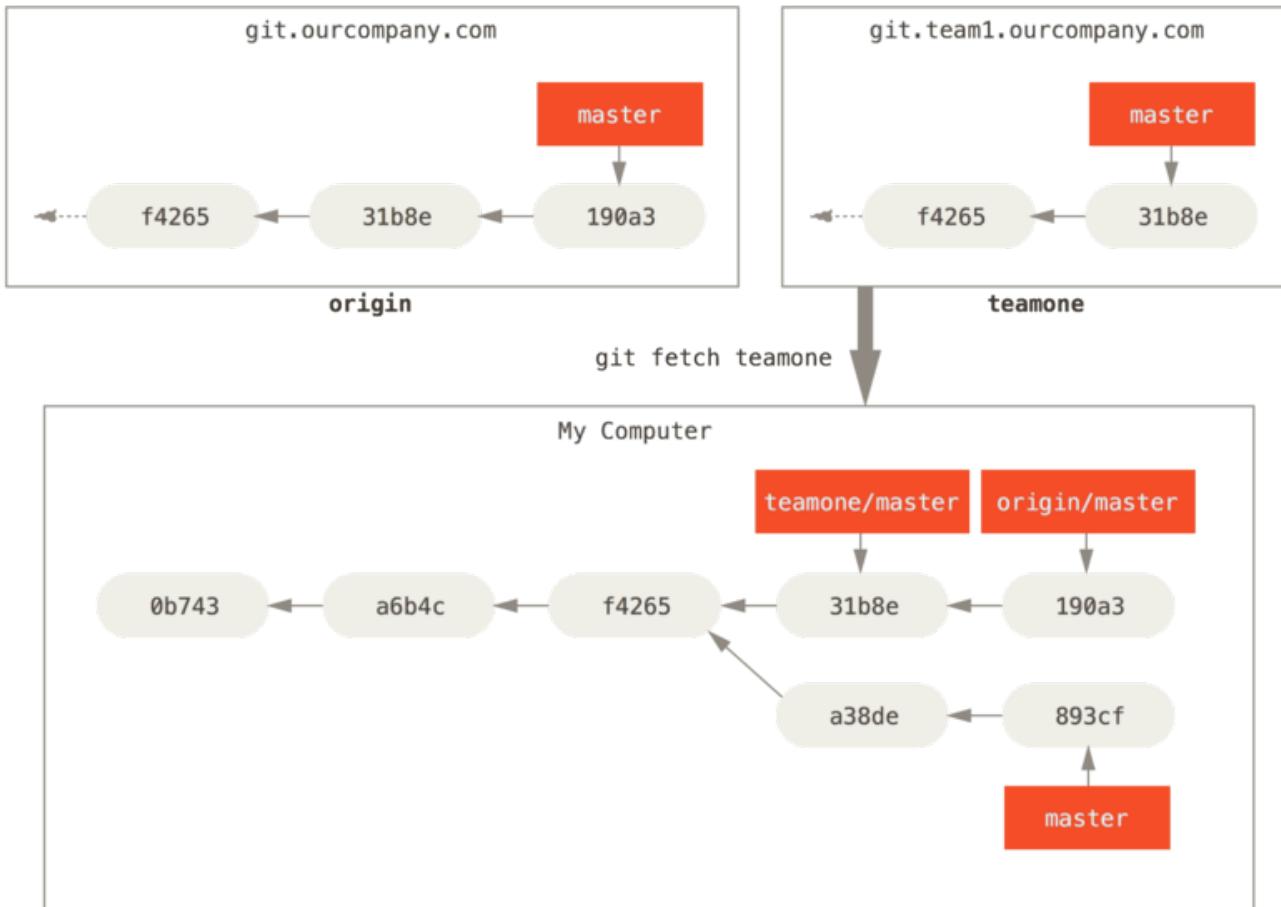
圖表 32. `git fetch` updates your remote references

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let's assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at `git.team1.ourcompany.com`. You can add it as a new remote reference to the project you're currently working on by running the `git remote add` command as we covered in [Git 基礎](#). Name this remote `teamone`, which will be your shortname for that whole URL.



圖表 33. Adding another server as a remote

Now, you can run `git fetch teamone` to fetch everything the remote `teamone` server has that you don't have yet. Because that server has a subset of the data your `origin` server has right now, Git fetches no data but sets a remote-tracking branch called `teamone/master` to point to the commit that `teamone` has as its `master` branch.



圖表 34. Remote tracking branch for `teamone/master`

Pushing

When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches aren't automatically synchronized to the remotes you write to – you have to explicitly push the branches you want to share. That way, you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named `serverfix` that you want to work on with others, you can push it up the same way you pushed your first branch. Run `git push <remote> <branch>`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

This is a bit of a shortcut. Git automatically expands the `serverfix` branchname out to `refs/heads/serverfix:refs/heads/serverfix`, which means, “Take my `serverfix` local branch and push it to update the remote’s `serverfix` branch.” We’ll go over the `refs/heads/` part in detail in [Git Internals](#), but you can generally leave it off. You can also do `git push origin`

`serverfix:serverfix`, which does the same thing – it says, “Take my serverfix and make it the remote’s serverfix.” You can use this format to push a local branch into a remote branch that is named differently. If you didn’t want it to be called `serverfix` on the remote, you could instead run `git push origin serverfix:awesomebranch` to push your local `serverfix` branch to the `awesomebranch` branch on the remote project.

Don’t type your password every time

If you’re using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication. By default it will prompt you on the terminal for this information so the server can tell if you’re allowed to push.

筆記

If you don’t want to type it every single time you push, you can set up a “credential cache”. The simplest is just to keep it in memory for a few minutes, which you can easily set up by running `git config --global credential.helper cache`.

For more information on the various credential caching options available, see [Credential Storage](#).

The next time one of your collaborators fetches from the server, they will get a reference to where the server’s version of `serverfix` is under the remote branch `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

It’s important to note that when you do a fetch that brings down new remote-tracking branches, you don’t automatically have local, editable copies of them. In other words, in this case, you don’t have a new `serverfix` branch – you only have an `origin/serverfix` pointer that you can’t modify.

To merge this work into your current working branch, you can run `git merge origin/serverfix`. If you want your own `serverfix` branch that you can work on, you can base it off your remote-tracking branch:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

This gives you a local branch that you can work on that starts where `origin/serverfix` is.

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a “tracking branch” (and the branch it tracks is called an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and

type `git pull`, Git automatically knows which server to fetch from and branch to merge into.

When you clone a repository, it generally automatically creates a `master` branch that tracks `origin/master`. However, you can set up other tracking branches if you wish – ones that track branches on other remotes, or don’t track the `master` branch. The simple case is the example you just saw, running `git checkout -b [branch] [remotename]/[branch]`. This is a common enough operation that git provides the `--track` shorthand:

```
$ git checkout --track origin/serverfix  
Branch serverfix set up to track remote branch serverfix from origin.  
Switched to a new branch 'serverfix'
```

In fact, this is so common that there’s even a shortcut for that shortcut. If the branch name you’re trying to checkout (a) doesn’t exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you:

```
$ git checkout serverfix  
Branch serverfix set up to track remote branch serverfix from origin.  
Switched to a new branch 'serverfix'
```

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

```
$ git checkout -b sf origin/serverfix  
Branch sf set up to track remote branch serverfix from origin.  
Switched to a new branch 'sf'
```

Now, your local branch `sf` will automatically pull from `origin/serverfix`.

If you already have a local branch and want to set it to a remote branch you just pulled down, or want to change the upstream branch you’re tracking, you can use the `-u` or `--set-upstream-to` option to `git branch` to explicitly set it at any time.

```
$ git branch -u origin/serverfix  
Branch serverfix set up to track remote branch serverfix from origin.
```

Upstream shorthand

筆記 When you have a tracking branch set up, you can reference its upstream branch with the `@{upstream}` or `@{u}` shorthand. So if you’re on the `master` branch and it’s tracking `origin/master`, you can say something like `git merge @{u}` instead of `git merge origin/master` if you wish.

If you want to see what tracking branches you have set up, you can use the `-vv` option to `git branch`. This will list out your local branches with more information including what each branch is tracking and if your local branch is ahead, behind or both.

```
$ git branch -vv
  iss53      7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master     1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this
  should do it
  testing    5ea463a trying something new
```

So here we can see that our `iss53` branch is tracking `origin/iss53` and is “ahead” by two, meaning that we have two commits locally that are not pushed to the server. We can also see that our `master` branch is tracking `origin/master` and is up to date. Next we can see that our `serverfix` branch is tracking the `server-fix-good` branch on our `teamone` server and is ahead by three and behind by one, meaning that there is one commit on the server we haven’t merged in yet and three commits locally that we haven’t pushed. Finally we can see that our `testing` branch is not tracking any remote branch.

It’s important to note that these numbers are only since the last time you fetched from each server. This command does not reach out to the servers, it’s telling you about what it has cached from these servers locally. If you want totally up to date ahead and behind numbers, you’ll need to fetch from all your remotes right before running this. You could do that like this: `git fetch --all; git branch -vv`

Pulling

While the `git fetch` command will fetch down all the changes on the server that you don’t have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself. However, there is a command called `git pull` which is essentially a `git fetch` immediately followed by a `git merge` in most cases. If you have a tracking branch set up as demonstrated in the last section, either by explicitly setting it or by having it created for you by the `clone` or `checkout` commands, `git pull` will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Generally it’s better to simply use the `fetch` and `merge` commands explicitly as the magic of `git pull` can often be confusing.

刪除遠端分支

Suppose you’re done with a remote branch – say you and your collaborators are finished with a feature and have merged it into your remote’s `master` branch (or whatever branch your stable codeline is in). You can delete a remote branch using the `--delete` option to `git push`. If you want to delete your `serverfix` branch from the server, you run the following:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]           serverfix
```

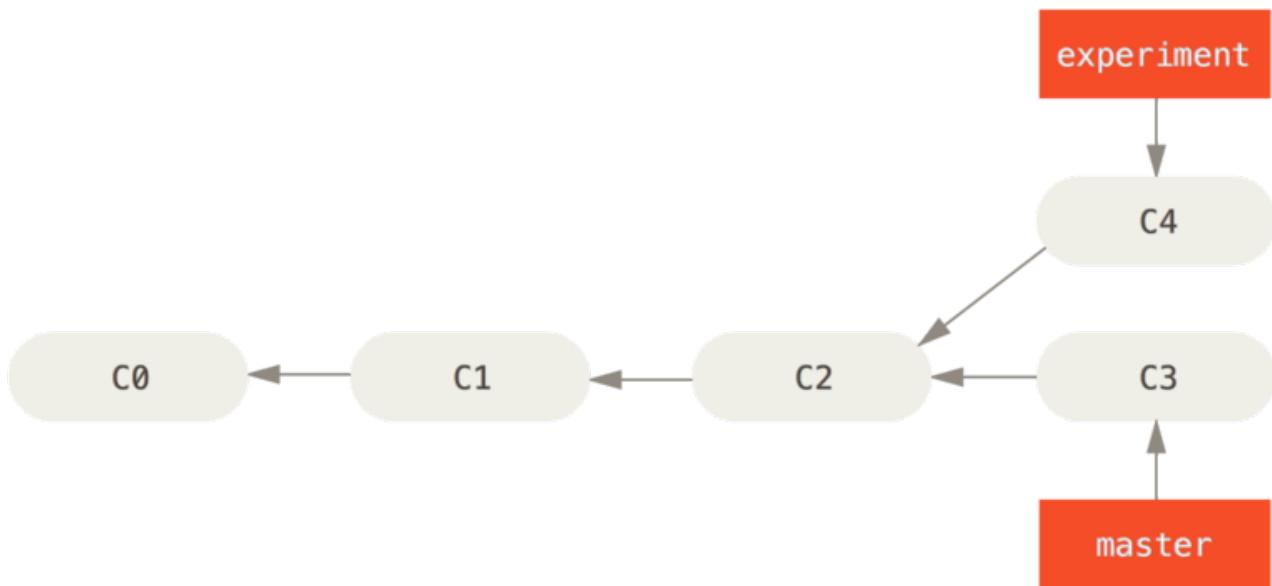
Basically all this does is remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs, so if it was accidentally deleted, it’s often easy to recover.

衍合

In Git, there are two main ways to integrate changes from one branch into another: the **merge** and the **rebase**. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

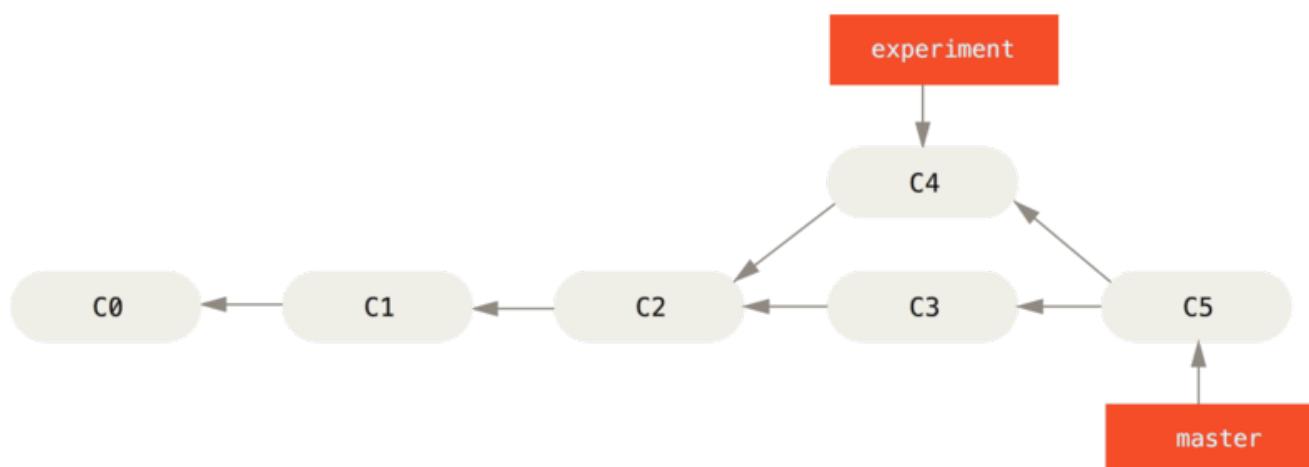
基本衍合

If you go back to an earlier example from [合併的基本用法](#), you can see that you diverged your work and made commits on two different branches.



图表 35. Simple divergent history

The easiest way to integrate the branches, as we've already covered, is the **merge** command. It performs a three-way merge between the two latest branch snapshots (C3 and C4) and the most recent common ancestor of the two (C2), creating a new snapshot (and commit).



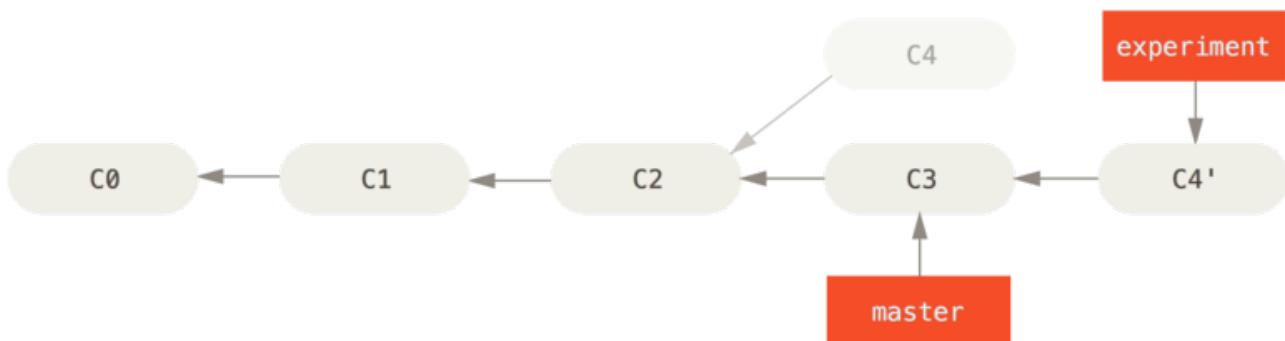
图表 36. Merging to integrate diverged work history

However, there is another way: you can take the patch of the change that was introduced in C4 and reapply it on top of C3. In Git, this is called **rebasing**. With the **rebase** command, you can take all the changes that were committed on one branch and replay them on another one.

In this example, you'd run the following:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

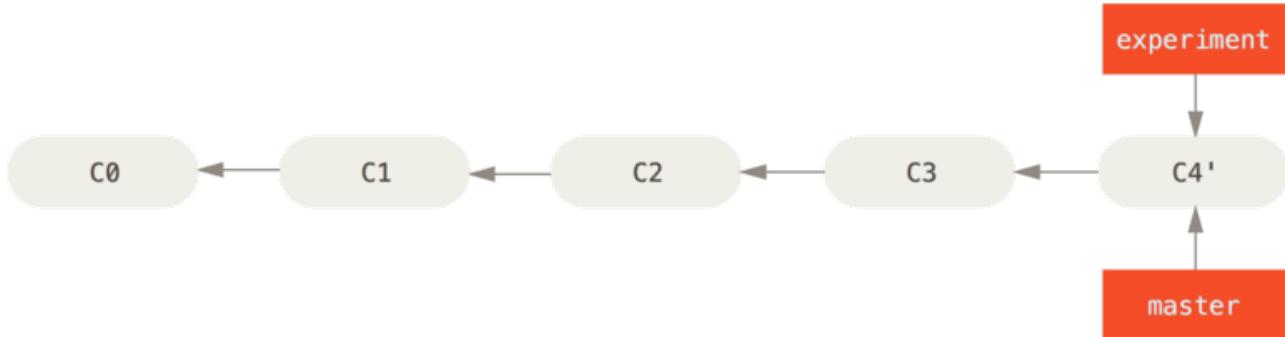
It works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.



圖表 37. Rebasing the change introduced in C4 onto C3

At this point, you can go back to the `master` branch and do a fast-forward merge.

```
$ git checkout master
$ git merge experiment
```



圖表 38. Fast-forwarding the master branch

Now, the snapshot pointed to by `C4'` is exactly the same as the one that was pointed to by `C5` in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

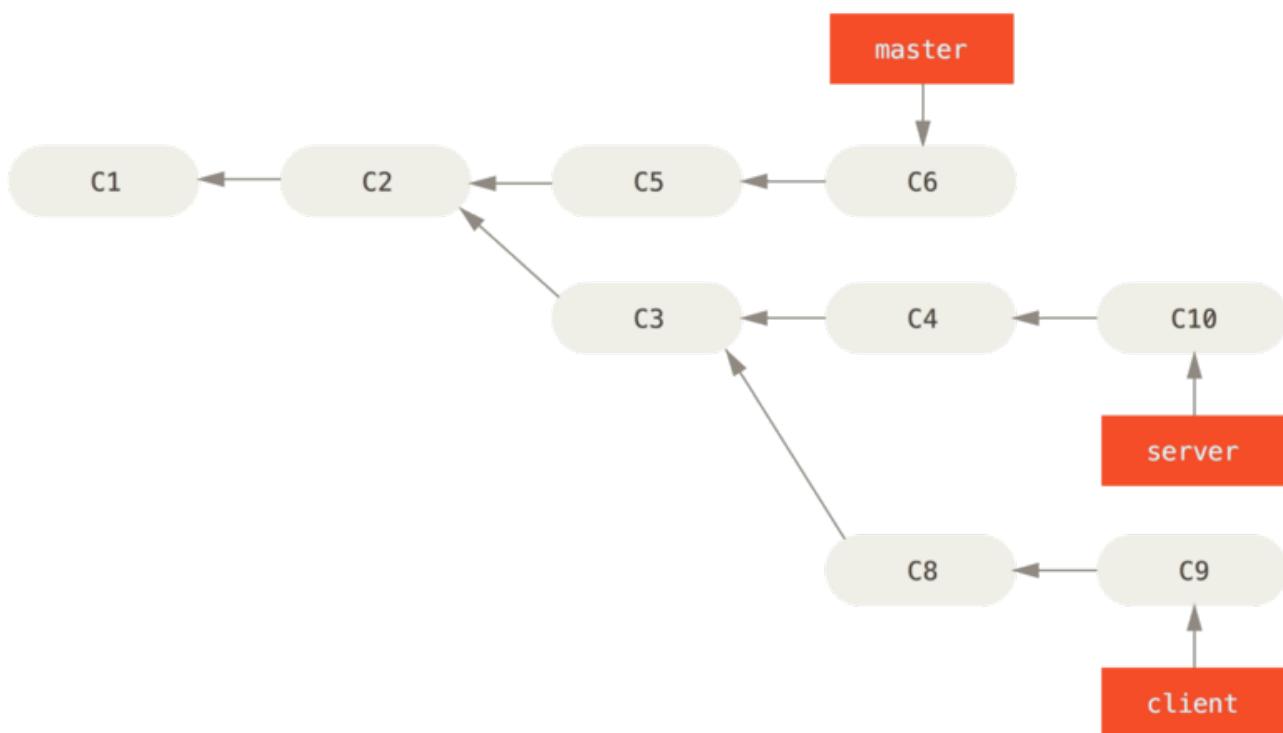
Often, you'll do this to make sure your commits apply cleanly on a remote branch – perhaps in a project to which you're trying to contribute but that you don't maintain. In this case, you'd do your work in a branch and then rebase your work onto `origin/master` when you were ready to

submit your patches to the main project. That way, the maintainer doesn't have to do any integration work – just a fast-forward or a clean apply.

Note that the snapshot pointed to by the final commit you end up with, whether it's the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot – it's only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

更多有趣的衍合

You can also have your rebase replay on something other than the rebase target branch. Take a history like [A history with a topic branch off another topic branch](#), for example. You branched a topic branch (`server`) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (`client`) and committed a few times. Finally, you went back to your server branch and did a few more commits.

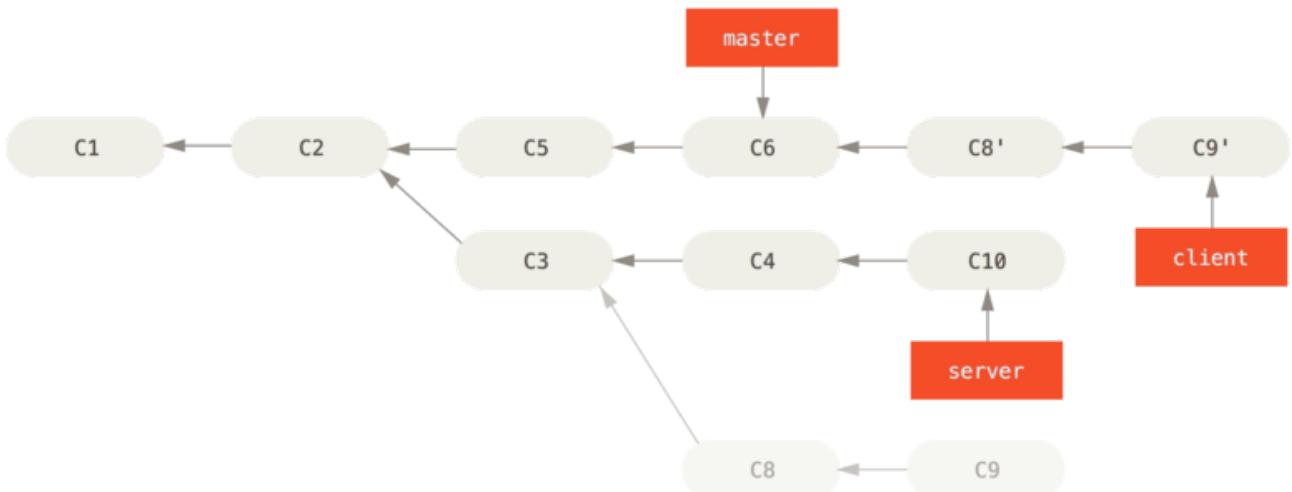


圖表 39. A history with a topic branch off another topic branch

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it's tested further. You can take the changes on client that aren't on server (`C8` and `C9`) and replay them on your `master` branch by using the `--onto` option of `git rebase`:

```
$ git rebase --onto master server client
```

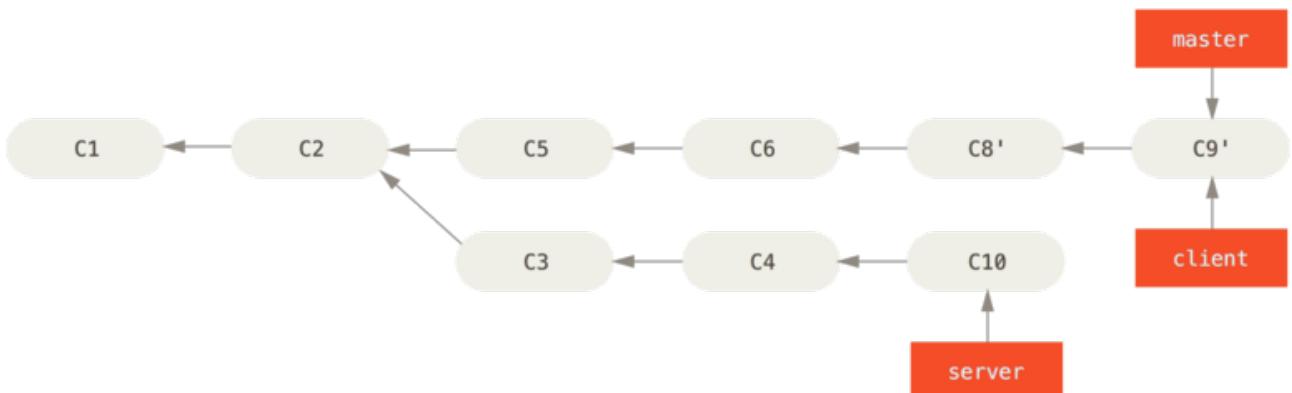
This basically says, “Check out the client branch, figure out the patches from the common ancestor of the `client` and `server` branches, and then replay them onto `master`.” It's a bit complex, but the result is pretty cool.



圖表 40. Rebasing a topic branch off another topic branch

Now you can fast-forward your `master` branch (see [Fast-forwarding your master branch to include the client branch changes](#)):

```
$ git checkout master
$ git merge client
```

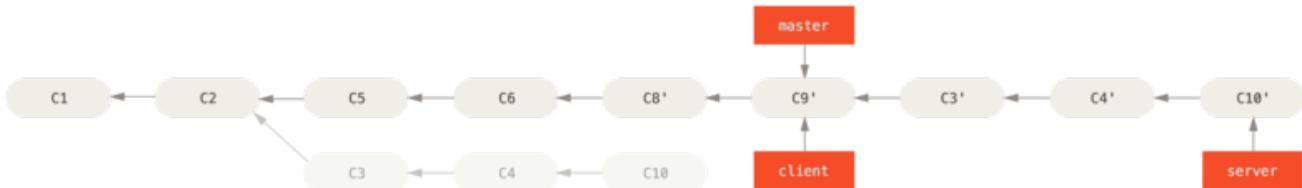


圖表 41. Fast-forwarding your master branch to include the client branch changes

Let's say you decide to pull in your server branch as well. You can rebase the server branch onto the `master` branch without having to check it out first by running `git rebase [basebranch] [topicbranch]` – which checks out the topic branch (in this case, `server`) for you and replays it onto the base branch (`master`):

```
$ git rebase master server
```

This replays your `server` work on top of your `master` work, as shown in [Rebasing your server branch on top of your master branch](#).



圖表 42. Rebasing your server branch on top of your master branch

Then, you can fast-forward the base branch (`master`):

```
$ git checkout master
$ git merge server
```

You can remove the `client` and `server` branches because all the work is integrated and you don't need them anymore, leaving your history for this entire process looking like [Final commit history](#):

```
$ git branch -d client
$ git branch -d server
```



圖表 43. Final commit history

使用衍和的危險

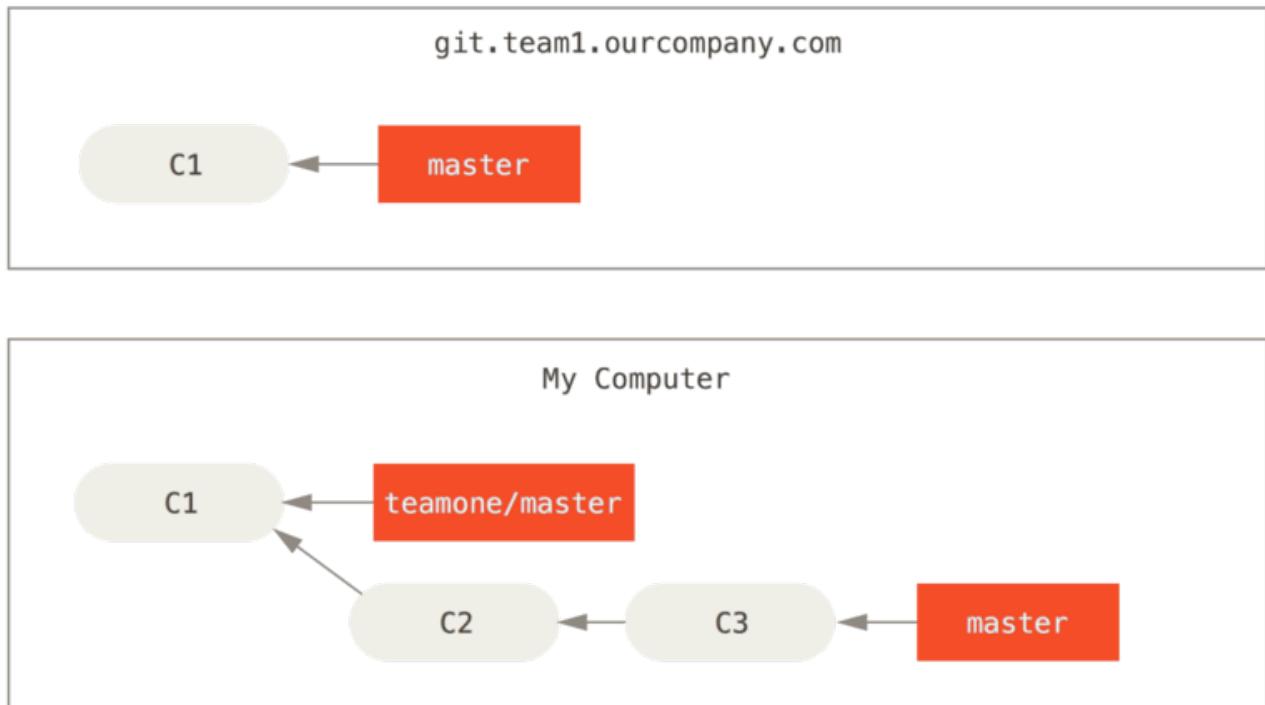
Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

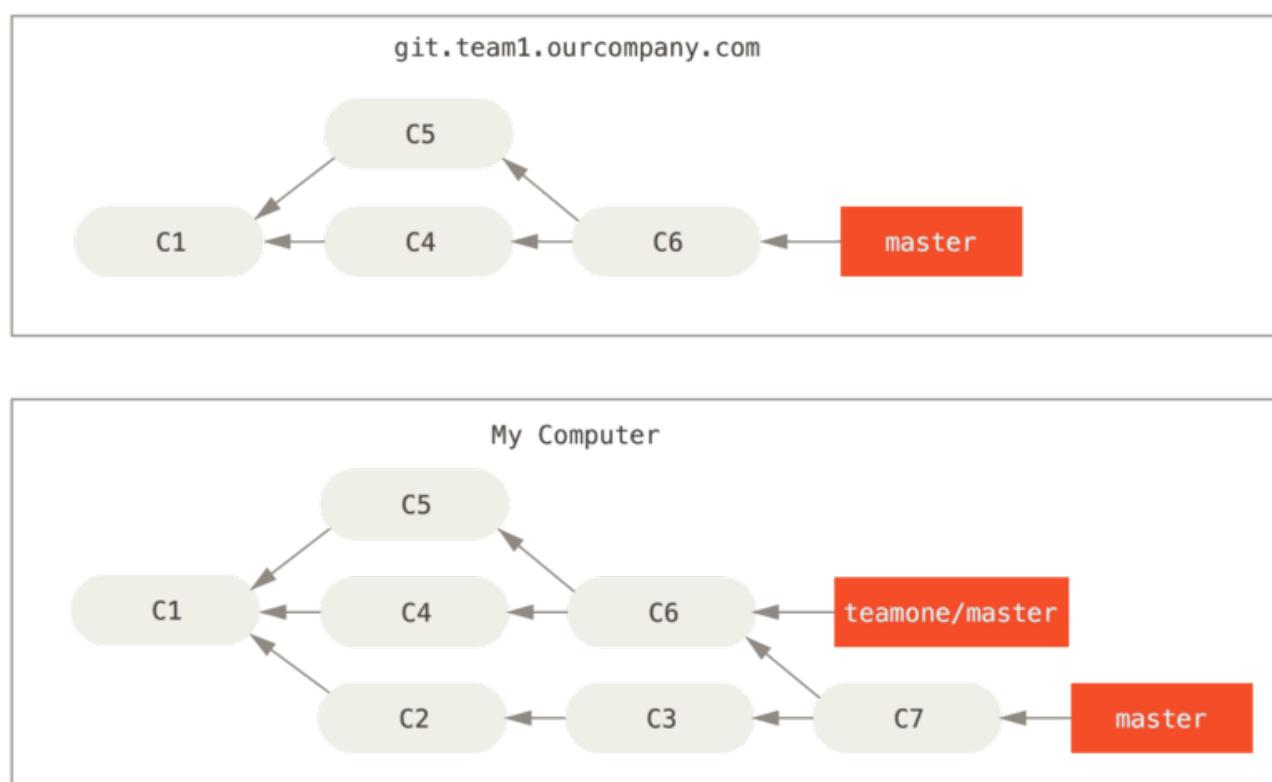
When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with `git rebase` and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let's look at an example of how rebasing work that you've made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:



圖表 44. Clone a repository, and base some work on it

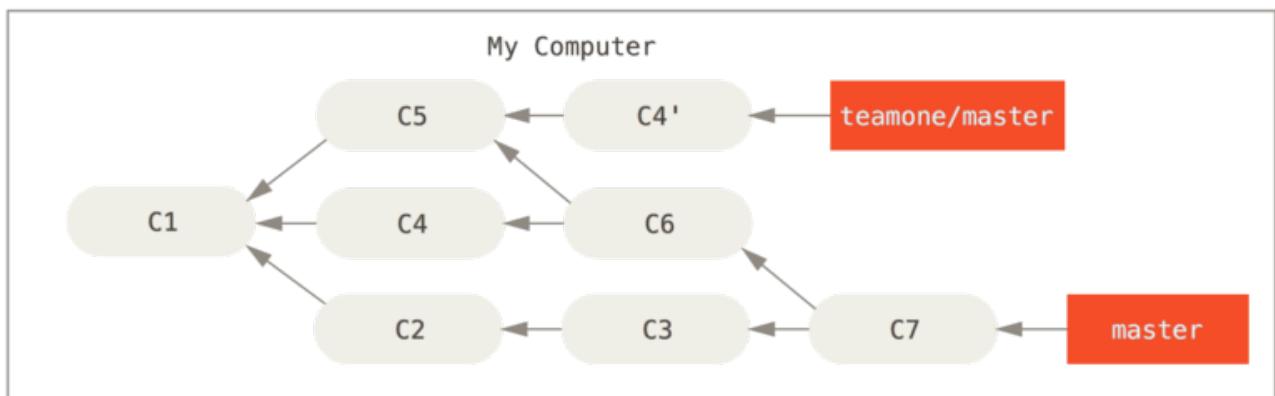
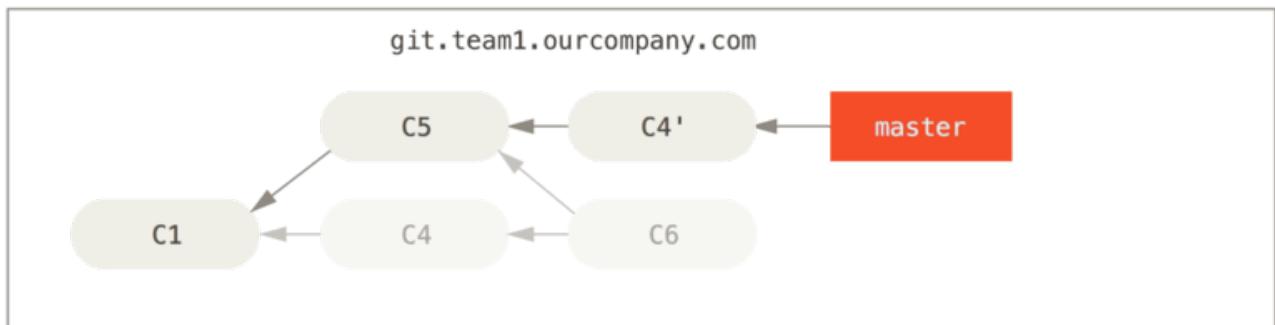
Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch it and merge the new remote branch into your work, making your history look something like this:



圖表 45. Fetch more commits, and merge them into your work

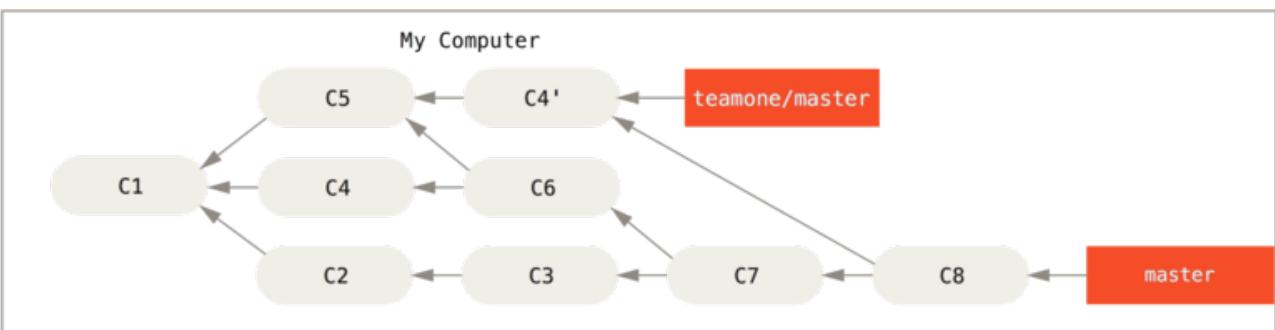
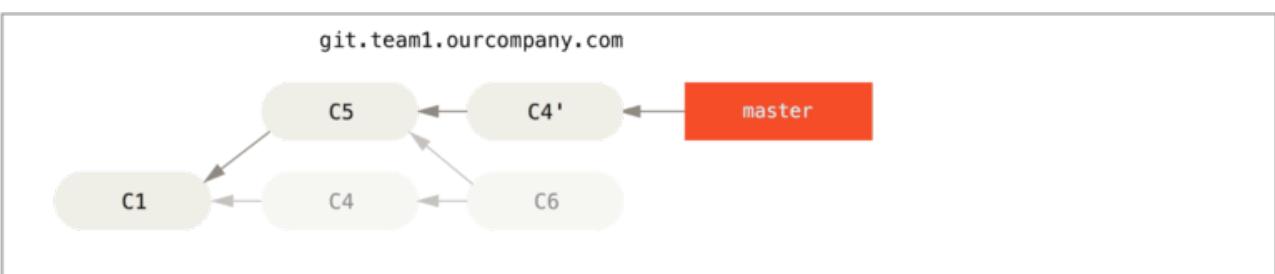
Next, the person who pushed the merged work decides to go back and rebase their work instead; they

do a `git push --force` to overwrite the history on the server. You then fetch from that server, bringing down the new commits.



圖表 46. Someone pushes rebased commits, abandoning commits you've based your work on

Now you're both in a pickle. If you do a `git pull`, you'll create a merge commit which includes both lines of history, and your repository will look like this:



圖表 47. You merge in the same work again into a new merge commit

If you run a `git log` when your history looks like this, you'll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to

the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people. It's pretty safe to assume that the other developer doesn't want C4 and C6 to be in the history; that's why they rebased in the first place.

Rebase When You Rebase

If you **do** find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you've based work on, your challenge is to figure out what is yours and what they've rewritten.

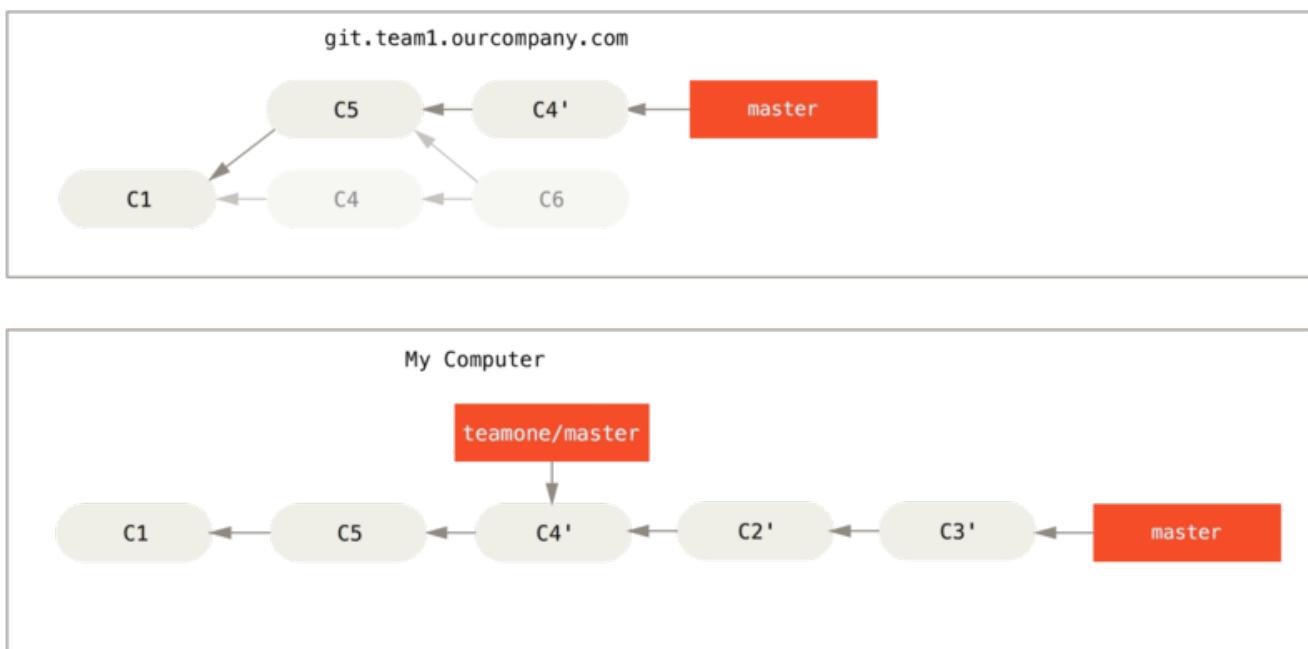
It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a "patch-id".

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we're at [Someone pushes rebased commits, abandoning commits you've based your work on](#) we run `git rebase teamone/master`, Git will:

- Determine what work is unique to our branch (C2, C3, C4, C6, C7)
- Determine which are not merge commits (C2, C3, C4)
- Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')
- Apply those commits to the top of `teamone/master`

So instead of the result we see in [You merge in the same work again into a new merge commit](#), we would end up with something more like [Rebase on top of force-pushed rebase work..](#)



圖表 48. *Rebase on top of force-pushed rebase work.*

This only works if C4 and C4' that your partner made are almost exactly the same patch. Otherwise the rebase won't be able to tell that it's a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running a `git pull --rebase` instead of a normal `git pull`. Or you could do it manually with a `git fetch` followed by a `git rebase teamone/master` in this case.

If you are using `git pull` and want to make `--rebase` the default, you can set the `pull.rebase` config value with something like `git config --global pull.rebase true`.

If you treat rebasing as a way to clean up and work with commits before you push them, and if you only rebase commits that have never been available publicly, then you'll be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run `git pull --rebase` to try to make the pain after it happens a little bit simpler.

衍合及合併的異同

Now that you've seen rebasing and merging in action, you may be wondering which one is better. Before we can answer this, let's step back a bit and talk about what history means.

One point of view on this is that your repository's commit history is a **record of what actually happened**. It's a historical document, valuable in its own right, and shouldn't be tampered with. From this angle, changing the commit history is almost blasphemous; you're *lying* about what actually transpired. So what if there was a messy series of merge commits? That's how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the **story of how your project was made**. You wouldn't publish the first draft of a book, and the manual for how to maintain your software deserves careful editing. This is the camp that uses tools like rebase and filter-branch to tell the story in the way that's best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you'll see that it's not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it's up to you to decide which one is best for your particular situation.

In general the way to get the best of both worlds is to rebase local changes you've made but haven't shared yet before you push them in order to clean up your story, but never rebase anything you've pushed somewhere.

總結

我們已介紹 Git 基本的分支和合併，你應該對於「建立並切換到新分支」、「在不同分支之間切換」、「合併本地分支」感到相當輕鬆寫意；你應該也能夠做到「把想要分享的分支推送到共用伺服器上」、「在共享的分支上與其他人協作」、「在分享自己的分支前先進行變基（rebase）」。下一章我們將介紹架設自己的 Git 版本庫託管伺服器所需要的知識。

伺服器上的 Git

At this point, you should be able to do most of the day-to-day tasks for which you'll be using Git. However, in order to do any collaboration in Git, you'll need to have a remote Git repository. Although you can technically push changes to and pull changes from individuals' repositories, doing so is discouraged because you can fairly easily confuse what they're working on if you're not careful. Furthermore, you want your collaborators to be able to access the repository even if your computer is offline – having a more reliable common repository is often useful. Therefore, the preferred method for collaborating with someone is to set up an intermediate repository that you both have access to, and push to and pull from that.

Running a Git server is fairly straightforward. First, you choose which protocols you want your server to communicate with. The first section of this chapter will cover the available protocols and the pros and cons of each. The next sections will explain some typical setups using those protocols and how to get your server running with them. Last, we'll go over a few hosted options, if you don't mind hosting your code on someone else's server and don't want to go through the hassle of setting up and maintaining your own server.

If you have no interest in running your own server, you can skip to the last section of the chapter to see some options for setting up a hosted account and then move on to the next chapter, where we discuss the various ins and outs of working in a distributed source control environment.

A remote repository is generally a *bare repository* – a Git repository that has no working directory. Because the repository is only used as a collaboration point, there is no reason to have a snapshot checked out on disk; it's just the Git data. In the simplest terms, a bare repository is the contents of your project's `.git` directory and nothing else.

通訊協定

Git can use four major protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git. Here we'll discuss what they are and in what basic circumstances you would want (or not want) to use them.

本機通訊協定

The most basic is the *Local protocol*, in which the remote repository is in another directory on disk. This is often used if everyone on your team has access to a shared filesystem such as an NFS mount, or in the less likely case that everyone logs in to the same computer. The latter wouldn't be ideal, because all your code repository instances would reside on the same computer, making a catastrophic loss much more likely.

If you have a shared mounted filesystem, then you can clone, push to, and pull from a local file-based repository. To clone a repository like this or to add one as a remote to an existing project, use the path to the repository as the URL. For example, to clone a local repository, you can run something like this:

```
$ git clone /opt/git/project.git
```

Or you can do this:

```
$ git clone file:///opt/git/project.git
```

Git operates slightly differently if you explicitly specify `file://` at the beginning of the URL. If you just specify the path, Git tries to use hardlinks or directly copy the files it needs. If you specify `file://`, Git fires up the processes that it normally uses to transfer data over a network which is generally a lot less efficient method of transferring the data. The main reason to specify the `file://` prefix is if you want a clean copy of the repository with extraneous references or objects left out – generally after an import from another version-control system or something similar (see [Git Internals](#) for maintenance tasks). We'll use the normal path here because doing so is almost always faster.

To add a local repository to an existing Git project, you can run something like this:

```
$ git remote add local_proj /opt/git/project.git
```

Then, you can push to and pull from that remote as though you were doing so over a network.

優點

The pros of file-based repositories are that they're simple and they use existing file permissions and network access. If you already have a shared filesystem to which your whole team has access, setting up a repository is very easy. You stick the bare repository copy somewhere everyone has shared access to and set the read/write permissions as you would for any other shared directory. We'll discuss how to export a bare repository copy for this purpose in [在伺服器上佈署 Git](#).

This is also a nice option for quickly grabbing work from someone else's working repository. If you and a co-worker are working on the same project and they want you to check something out, running a command like `git pull /home/john/project` is often easier than them pushing to a remote server and you pulling down.

缺點

The cons of this method are that shared access is generally more difficult to set up and reach from multiple locations than basic network access. If you want to push from your laptop when you're at home, you have to mount the remote disk, which can be difficult and slow compared to network-based access.

It's important to mention that this isn't necessarily the fastest option if you're using a shared mount of some kind. A local repository is fast only if you have fast access to the data. A repository on NFS is often slower than the repository over SSH on the same server, allowing Git to run off local disks on each system.

Finally, this protocol does not protect the repository against accidental damage. Every user has full shell access to the "remote" directory, and there is nothing preventing them from changing or removing internal Git files and corrupting the repository.

HTTP 通訊協定

Git can communicate over HTTP in two different modes. Prior to Git 1.6.6 there was only one way it could do this which was very simple and generally read-only. In version 1.6.6 a new, smarter protocol

was introduced that involved Git being able to intelligently negotiate data transfer in a manner similar to how it does over SSH. In the last few years, this new HTTP protocol has become very popular since it's simpler for the user and smarter about how it communicates. The newer version is often referred to as the "Smart" HTTP protocol and the older way as "Dumb" HTTP. We'll cover the newer "smart" HTTP protocol first.

Smart HTTP

The "smart" HTTP protocol operates very similarly to the SSH or Git protocols but runs over standard HTTP/S ports and can use various HTTP authentication mechanisms, meaning it's often easier on the user than something like SSH, since you can use things like username/password basic authentication rather than having to set up SSH keys.

It has probably become the most popular way to use Git now, since it can be set up to both serve anonymously like the `git://` protocol, and can also be pushed over with authentication and encryption like the SSH protocol. Instead of having to set up different URLs for these things, you can now use a single URL for both. If you try to push and the repository requires authentication (which it normally should), the server can prompt for a username and password. The same goes for read access.

In fact, for services like GitHub, the URL you use to view the repository online (for example, "`https://github.com/schacon/simplegit[]`") is the same URL you can use to clone and, if you have access, push over.

Dumb HTTP

If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler "dumb" HTTP protocol. The Dumb protocol expects the bare Git repository to be served like normal files from the web server. The beauty of the Dumb HTTP protocol is the simplicity of setting it up. Basically, all you have to do is put a bare Git repository under your HTTP document root and set up a specific `post-update` hook, and you're done (See [Git Hooks](#)). At that point, anyone who can access the web server under which you put the repository can also clone your repository. To allow read access to your repository over HTTP, do something like this:

```
$ cd /var/www/htdocs/  
$ git clone --bare /path/to/git_project gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```

That's all. The `post-update` hook that comes with Git by default runs the appropriate command (`git update-server-info`) to make HTTP fetching and cloning work properly. This command is run when you push to this repository (over SSH perhaps); then, other people can clone via something like

```
$ git clone https://example.com/gitproject.git
```

In this particular case, we're using the `/var/www/htdocs` path that is common for Apache setups, but you can use any static web server – just put the bare repository in its path. The Git data is served as basic static files (see [Git Internals](#) for details about exactly how it's served).

Generally you would either choose to run a read/write Smart HTTP server or simply have the files

accessible as read-only in the Dumb manner. It's rare to run a mix of the two services.

優點

We'll concentrate on the pros of the Smart version of the HTTP protocol.

The simplicity of having a single URL for all types of access and having the server prompt only when authentication is needed makes things very easy for the end user. Being able to authenticate with a username and password is also a big advantage over SSH, since users don't have to generate SSH keys locally and upload their public key to the server before being able to interact with it. For less sophisticated users, or users on systems where SSH is less common, this is a major advantage in usability. It is also a very fast and efficient protocol, similar to the SSH one.

You can also serve your repositories read-only over HTTPS, which means you can encrypt the content transfer; or you can go so far as to make the clients use specific signed SSL certificates.

Another nice thing is that HTTP/S are such commonly used protocols that corporate firewalls are often set up to allow traffic through these ports.

缺點

Git over HTTP/S can be a little more tricky to set up compared to SSH on some servers. Other than that, there is very little advantage that other protocols have over the "Smart" HTTP protocol for serving Git.

If you're using HTTP for authenticated pushing, providing your credentials is sometimes more complicated than using keys over SSH. There are however several credential caching tools you can use, including Keychain access on OSX and Credential Manager on Windows, to make this pretty painless. Read [Credential Storage](#) to see how to set up secure HTTP password caching on your system.

SSH 通訊協定

A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places – and if it isn't, it's easy to do. SSH is also an authenticated network protocol; and because it's ubiquitous, it's generally easy to set up and use.

To clone a Git repository over SSH, you can specify ssh:// URL like this:

```
$ git clone ssh://user@server/project.git
```

Or you can use the shorter scp-like syntax for the SSH protocol:

```
$ git clone user@server:project.git
```

You can also not specify a user, and Git assumes the user you're currently logged in as.

優點

The pros of using SSH are many. First, SSH is relatively easy to set up – SSH daemons are commonplace, many network admins have experience with them, and many OS distributions are set up with them or have tools to manage them. Next, access over SSH is secure – all data transfer is

encrypted and authenticated. Last, like the HTTP/S, Git and Local protocols, SSH is efficient, making the data as compact as possible before transferring it.

缺點

The negative aspect of SSH is that you can't serve anonymous access of your repository over it. People must have access to your machine over SSH to access it, even in a read-only capacity, which doesn't make SSH access conducive to open source projects. If you're using it only within your corporate network, SSH may be the only protocol you need to deal with. If you want to allow anonymous read-only access to your projects and also want to use SSH, you'll have to set up SSH for you to push over but something else for others to fetch over.

The Git Protocol

Next is the Git protocol. This is a special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication. In order for a repository to be served over the Git protocol, you must create the `git-daemon-export-ok` file – the daemon won't serve a repository without that file in it – but other than that there is no security. Either the Git repository is available for everyone to clone or it isn't. This means that there is generally no pushing over this protocol. You can enable push access; but given the lack of authentication, if you turn on push access, anyone on the internet who finds your project's URL could push to your project. Suffice it to say that this is rare.

優點

The Git protocol is often the fastest network transfer protocol available. If you're serving a lot of traffic for a public project or serving a very large project that doesn't require user authentication for read access, it's likely that you'll want to set up a Git daemon to serve your project. It uses the same data-transfer mechanism as the SSH protocol but without the encryption and authentication overhead.

缺點

The downside of the Git protocol is the lack of authentication. It's generally undesirable for the Git protocol to be the only access to your project. Generally, you'll pair it with SSH or HTTPS access for the few developers who have push (write) access and have everyone else use `git://` for read-only access. It's also probably the most difficult protocol to set up. It must run its own daemon, which requires `xinetd` configuration or the like, which isn't always a walk in the park. It also requires firewall access to port 9418, which isn't a standard port that corporate firewalls always allow. Behind big corporate firewalls, this obscure port is commonly blocked.

在伺服器上佈署 Git

Now we'll cover setting up a Git service running these protocols on your own server.

筆記

Here we'll be demonstrating the commands and steps needed to do basic, simplified installations on a Linux based server, though it's also possible to run these services on Mac or Windows servers. Actually setting up a production server within your infrastructure will certainly entail differences in security measures or operating system tools, but hopefully this will give you the general idea of what's involved.

In order to initially set up any Git server, you have to export an existing repository into a new bare repository – a repository that doesn’t contain a working directory. This is generally straightforward to do. In order to clone your repository to create a new bare repository, you run the clone command with the `--bare` option. By convention, bare repository directories end in `.git`, like so:

```
$ git clone --bare my_project my_project.git  
Cloning into bare repository 'my_project.git'...  
done.
```

You should now have a copy of the Git directory data in your `my_project.git` directory.

This is roughly equivalent to something like

```
$ cp -Rf my_project/.git my_project.git
```

There are a couple of minor differences in the configuration file; but for your purpose, this is close to the same thing. It takes the Git repository by itself, without a working directory, and creates a directory specifically for it alone.

把 Bare Repository 放到伺服器上

Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols. Let’s say you’ve set up a server called `git.example.com` that you have SSH access to, and you want to store all your Git repositories under the `/srv/git` directory. Assuming that `/srv/git` exists on that server, you can set up your new repository by copying your bare repository over:

```
$ scp -r my_project.git user@git.example.com:/srv/git
```

At this point, other users who have SSH access to the same server which has read-access to the `/srv/git` directory can clone your repository by running

```
$ git clone user@git.example.com:/srv/git/my_project.git
```

If a user SSHs into a server and has write access to the `/srv/git/my_project.git` directory, they will also automatically have push access.

Git will automatically add group write permissions to a repository properly if you run the `git init` command with the `--shared` option.

```
$ ssh user@git.example.com  
$ cd /srv/git/my_project.git  
$ git init --bare --shared
```

You see how easy it is to take a Git repository, create a bare version, and place it on a server to which

you and your collaborators have SSH access. Now you’re ready to collaborate on the same project.

It’s important to note that this is literally all you need to do to run a useful Git server to which several people have access – just add SSH-able accounts on a server, and stick a bare repository somewhere that all those users have read and write access to. You’re ready to go – nothing else needed.

In the next few sections, you’ll see how to expand to more sophisticated setups. This discussion will include not having to create user accounts for each user, adding public read access to repositories, setting up web UIs and more. However, keep in mind that to collaborate with a couple of people on a private project, all you *need* is an SSH server and a bare repository.

小型安裝

If you’re a small outfit or are just trying out Git in your organization and have only a few developers, things can be simple for you. One of the most complicated aspects of setting up a Git server is user management. If you want some repositories to be read-only to certain users and read/write to others, access and permissions can be a bit more difficult to arrange.

SSH 存取

If you have a server to which all your developers already have SSH access, it’s generally easiest to set up your first repository there, because you have to do almost no work (as we covered in the last section). If you want more complex access control type permissions on your repositories, you can handle them with the normal filesystem permissions of the operating system your server runs.

If you want to place your repositories on a server that doesn’t have accounts for everyone on your team whom you want to have write access, then you must set up SSH access for them. We assume that if you have a server with which to do this, you already have an SSH server installed, and that’s how you’re accessing the server.

There are a few ways you can give access to everyone on your team. The first is to set up accounts for everybody, which is straightforward but can be cumbersome. You may not want to run `adduser` and set temporary passwords for every user.

A second method is to create a single *git* user on the machine, ask every user who is to have write access to send you an SSH public key, and add that key to the `~/.ssh/authorized_keys` file of your new *git* user. At that point, everyone will be able to access that machine via the *git* user. This doesn’t affect the commit data in any way – the SSH user you connect as doesn’t affect the commits you’ve recorded.

Another way to do it is to have your SSH server authenticate from an LDAP server or some other centralized authentication source that you may already have set up. As long as each user can get shell access on the machine, any SSH authentication mechanism you can think of should work.

產生你的 SSH 公鑰

That being said, many Git servers authenticate using SSH public keys. In order to provide a public key, each user in your system must generate one if they don’t already have one. This process is similar across all operating systems. First, you should check to make sure you don’t already have a key. By default, a user’s SSH keys are stored in that user’s `~/.ssh` directory. You can easily check to see if you have a key already by going to that directory and listing the contents:

```
$ cd ~/.ssh  
$ ls  
authorized_keys2  id_dsa      known_hosts  
config           id_dsa.pub
```

You’re looking for a pair of files named something like `id_dsa` or `id_rsa` and a matching file with a `.pub` extension. The `.pub` file is your public key, and the other file is your private key. If you don’t have these files (or you don’t even have a `.ssh` directory), you can create them by running a program called `ssh-keygen`, which is provided with the SSH package on Linux/Mac systems and comes with Git for Windows:

```
$ ssh-keygen  
Generating public/private rsa key pair.  
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):  
Created directory '/home/schacon/.ssh'.  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /home/schacon/.ssh/id_rsa.  
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.  
The key fingerprint is:  
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

First it confirms where you want to save the key (`.ssh/id_rsa`), and then it asks twice for a passphrase, which you can leave empty if you don’t want to type a password when you use the key.

Now, each user that does this has to send their public key to you or whoever is administrating the Git server (assuming you’re using an SSH server setup that requires public keys). All they have to do is copy the contents of the `.pub` file and email it. The public keys look something like this:

```
$ cat ~/.ssh/id_rsa.pub  
ssh-rsa  
AAAA3NzaC1yc2EAAAABIwAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU  
GP1+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7x1ELEVf4h91FX5QVkbPppSwg0cda  
3  
Pbv7k0dJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvSlVK/7X  
A  
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprrX88XypNDvjYNby6vw/Pb0rwert/E  
n  
mZ+AW40ZPnPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsb  
x  
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

For a more in-depth tutorial on creating an SSH key on multiple operating systems, see the GitHub guide on SSH keys at <https://help.github.com/articles/generating-ssh-keys>.

設定伺服器

Let's walk through setting up SSH access on the server side. In this example, you'll use the `authorized_keys` method for authenticating your users. We also assume you're running a standard Linux distribution like Ubuntu. First, you create a `git` user and a `.ssh` directory for that user.

```
$ sudo adduser git  
$ su git  
$ cd  
$ mkdir .ssh && chmod 700 .ssh  
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Next, you need to add some developer SSH public keys to the `authorized_keys` file for the `git` user. Let's assume you have some trusted public keys and have saved them to temporary files. Again, the public keys look something like this:

```
$ cat /tmp/id_rsa.john.pub  
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L  
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k  
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpgW1GYEIgS9Ez  
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv  
07TCUSBdLQ1gMV0Fq1I2uPWQ0k0WQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq  
dAv8JggJICUvax2T9va5 gsg-keypair
```

You just append them to the `git` user's `authorized_keys` file in its `.ssh` directory:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys  
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys  
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Now, you can set up an empty repository for them by running `git init` with the `--bare` option, which initializes the repository without a working directory:

```
$ cd /opt/git  
$ mkdir project.git  
$ cd project.git  
$ git init --bare  
Initialized empty Git repository in /opt/git/project.git/
```

Then, John, Josie, or Jessica can push the first version of their project into that repository by adding it as a remote and pushing up a branch. Note that someone must shell onto the machine and create a bare repository every time you want to add a project. Let's use `gitserver` as the hostname of the server on which you've set up your `git` user and repository. If you're running it internally, and you set up DNS for `gitserver` to point to that server, then you can use the commands pretty much as is (assuming that `myproject` is an existing project with files in it):

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

At this point, the others can clone it down and push changes back up just as easily:

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

With this method, you can quickly get a read/write Git server up and running for a handful of developers.

You should note that currently all these users can also log into the server and get a shell as the `git` user. If you want to restrict that, you will have to change the shell to something else in the `passwd` file.

You can easily restrict the `git` user to only doing Git activities with a limited shell tool called `git-shell` that comes with Git. If you set this as your `git` user's login shell, then the `git` user can't have normal shell access to your server. To use this, specify `git-shell` instead of bash or csh for your user's login shell. To do so, you must first add `git-shell` to `/etc/shells` if it's not already there:

```
$ cat /etc/shells  # see if `git-shell` is already in there.  If not...
$ which git-shell  # make sure git-shell is installed on your system.
$ sudo vim /etc/shells  # and add the path to git-shell from last
command
```

Now you can edit the shell for a user using `chsh <username>`:

```
$ sudo chsh git  # and enter the path to git-shell, usually:
/usr/bin/git-shell
```

Now, the `git` user can only use the SSH connection to push and pull Git repositories and can't shell onto the machine. If you try, you'll see a login rejection like this:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute
access.
Connection to gitserver closed.
```

Now Git network commands will still work just fine but the users won't be able to get a shell. As the output states, you can also set up a directory in the `git` user's home directory that customizes the `git-shell` command a bit. For instance, you can restrict the Git commands that the server will accept or you can customize the message that users see if they try to SSH in like that. Run `git help shell` for more information on customizing the shell.

Git 常駐程式

Next we'll set up a daemon serving repositories over the "Git" protocol. This is common choice for fast, unauthenticated access to your Git data. Remember that since it's not an authenticated service, anything you serve over this protocol is public within its network.

If you're running this on a server outside your firewall, it should only be used for projects that are publicly visible to the world. If the server you're running it on is inside your firewall, you might use it for projects that a large number of people or computers (continuous integration or build servers) have read-only access to, when you don't want to have to add an SSH key for each.

In any case, the Git protocol is relatively easy to set up. Basically, you need to run this command in a daemonized manner:

```
$ git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` allows the server to restart without waiting for old connections to time out, the `--base-path` option allows people to clone projects without specifying the entire path, and the path at the end tells the Git daemon where to look for repositories to export. If you're running a firewall, you'll also need to punch a hole in it at port 9418 on the box you're setting this up on.

You can daemonize this process a number of ways, depending on the operating system you're running. On an Ubuntu machine, you can use an Upstart script. So, in the following file

```
/etc/init/local-git-daemon.conf
```

you put this script:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
--user=git --group=git \
--reuseaddr \
--base-path=/opt/git/ \
/opt/git/
respawn
```

For security reasons, it is strongly encouraged to have this daemon run as a user with read-only permissions to the repositories – you can easily do this by creating a new user `git-ro` and running the daemon as them. For the sake of simplicity we'll simply run it as the same `git` user that `git-shell` is running as.

When you restart your machine, your Git daemon will start automatically and respawn if it goes down. To get it running without having to reboot, you can run this:

```
$ initctl start local-git-daemon
```

On other systems, you may want to use `xinetd`, a script in your `sysvinit` system, or something else – as long as you get that command daemonized and watched somehow.

Next, you have to tell Git which repositories to allow unauthenticated Git server-based access to. You can do this in each repository by creating a file named `git-daemon-export-ok`.

```
$ cd /path/to/project.git  
$ touch git-daemon-export-ok
```

The presence of that file tells Git that it's OK to serve this project without authentication.

Smart HTTP

We now have authenticated access through SSH and unauthenticated access through `git://`, but there is also a protocol that can do both at the same time. Setting up Smart HTTP is basically just enabling a CGI script that is provided with Git called `git-http-backend` on the server. This CGI will read the path and headers sent by a `git fetch` or `git push` to an HTTP URL and determine if the client can communicate over HTTP (which is true for any client since version 1.6.6). If the CGI sees that the client is smart, it will communicate smartly with it, otherwise it will fall back to the dumb behavior (so it is backward compatible for reads with older clients).

Let's walk through a very basic setup. We'll set this up with Apache as the CGI server. If you don't have Apache setup, you can do so on a Linux box with something like this:

```
$ sudo apt-get install apache2 apache2-utils  
$ a2enmod cgi alias env rewrite
```

This also enables the `mod_cgi`, `mod_alias`, `mod_env`, and `mod_rewrite` modules, which are all needed for this to work properly.

You'll also need to set the Unix user group of the `/opt/git` directories to `www-data` so your web server can read- and write-access the repositories, because the Apache instance running the CGI script will (by default) be running as that user:

```
$ chgrp -R www-data /opt/git
```

Next we need to add some things to the Apache configuration to run the `git-http-backend` as the handler for anything coming into the `/git` path of your web server.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

If you leave out `GIT_HTTP_EXPORT_ALL` environment variable, then Git will only serve to unauthenticated clients the repositories with the `git-daemon-export-ok` file in them, just like the Git daemon did.

Finally you'll want to tell Apache to allow requests to `git-http-backend` and make writes be authenticated somehow, possibly with an Auth block like this:

```
RewriteEngine On
RewriteCond %{QUERY_STRING} service=git-receive-pack [OR]
RewriteCond %{REQUEST_URI} /git-receive-pack$
RewriteRule ^/git/ - [E=AUTHREQUIRED]

<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
    Order deny,allow
    Deny from env=AUTHREQUIRED
    Satisfy any
</Files>
```

That will require you to create a `.htpasswd` file containing the passwords of all the valid users. Here is an example of adding a “schacon” user to the file:

```
$ htpasswd -c /opt/git/.htpasswd schacon
```

There are tons of ways to have Apache authenticate users, you'll have to choose and implement one of them. This is just the simplest example we could come up with. You'll also almost certainly want to set this up over SSL so all this data is encrypted.

We don't want to go too far down the rabbit hole of Apache configuration specifics, since you could well be using a different server or have different authentication needs. The idea is that Git comes with a CGI called `git-http-backend` that when invoked will do all the negotiation to send and receive data over HTTP. It does not implement any authentication itself, but that can easily be controlled at the layer of the web server that invokes it. You can do this with nearly any CGI-capable web server, so go with the one that you know best.

筆記

For more information on configuring authentication in Apache, check out the Apache docs here: <http://httpd.apache.org/docs/current/howto/auth.html>

GitWeb

Now that you have basic read/write and read-only access to your project, you may want to set up a simple web-based visualizer. Git comes with a CGI script called GitWeb that is sometimes used for this.

The screenshot shows the GitWeb interface for a repository. At the top, there's a navigation bar with links for 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. On the right, there are buttons for 'commit', 'search' (with a search input field), and 're'. Below the navigation, there's a section for 'description' with the text 'Unnamed repository; edit this file 'description' to name the repository.', 'owner' (Ben Straub), and 'last change' (Wed, 11 Jun 2014 12:20:23 -0700 (21:20 +0200)). The main content area is divided into sections: 'shortlog' and 'tags'. The 'shortlog' section lists commits from June 2014, showing authors like Carlos Martí, Vicent Martí, and Philip Kelley, along with commit messages and dates. The 'tags' section lists various Git versions with their corresponding commit dates.

Date	Tag	Author	Commit Message
3 weeks ago	v0.21.0-rc1		
7 months ago	v0.20.0		
12 months ago	v0.19.0		
14 months ago	v0.18.0		
2 years ago	v0.17.0		
2 years ago	v0.16.0	libgit2 v0.16.0	
2 years ago	v0.15.0		
2 years ago	v0.14.0		
3 years ago	v0.13.0		
3 years ago	v0.12.0		
3 years ago	v0.11.0		

圖表 49. The GitWeb web-based user interface.

If you want to check out what GitWeb would look like for your project, Git comes with a command to fire up a temporary instance if you have a lightweight server on your system like `lighttpd` or `webrick`. On Linux machines, `lighttpd` is often installed, so you may be able to get it to run by typing `git instaweb` in your project directory. If you’re running a Mac, Leopard comes preinstalled with Ruby, so `webrick` may be your best bet. To start `instaweb` with a non-lighttpd handler, you can run it with the `--httpd` option.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

That starts up an HTTPD server on port 1234 and then automatically starts a web browser that opens on that page. It’s pretty easy on your part. When you’re done and want to shut down the server, you can run the same command with the `--stop` option:

```
$ git instaweb --httpd=webrick --stop
```

If you want to run the web interface on a server all the time for your team or for an open source project you’re hosting, you’ll need to set up the CGI script to be served by your normal web server. Some Linux distributions have a `gitweb` package that you may be able to install via `apt` or `yum`, so you may want to try that first. We’ll walk through installing GitWeb manually very quickly. First, you need to get the Git source code, which GitWeb comes with, and generate the custom CGI script:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
      SUBDIR gitweb
      SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
      GEN gitweb.cgi
      GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Notice that you have to tell the command where to find your Git repositories with the `GITWEB_PROJECTROOT` variable. Now, you need to make Apache use CGI for that script, for which you can add a `VirtualHost`:

```
<VirtualHost *:80>
  ServerName gitserver
  DocumentRoot /var/www/gitweb
  <Directory /var/www/gitweb>
    Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
    AllowOverride All
    order allow,deny
    Allow from all
    AddHandler cgi-script cgi
    DirectoryIndex gitweb.cgi
  </Directory>
</VirtualHost>
```

Again, GitWeb can be served with any CGI or Perl capable web server; if you prefer to use something else, it shouldn’t be difficult to set up. At this point, you should be able to visit `http://gitserver/` to view your repositories online.

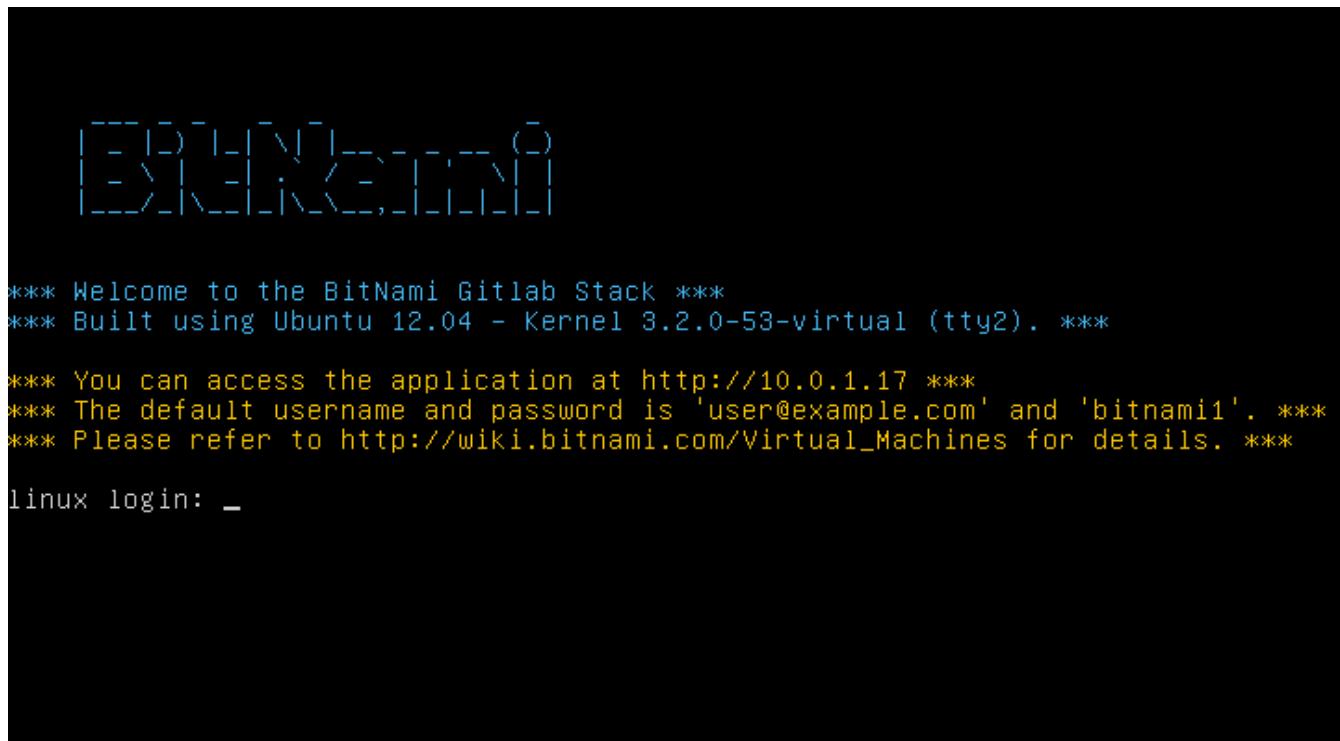
GitLab

GitWeb is pretty simplistic though. If you’re looking for a more modern, fully featured Git server, there are some several open source solutions out there that you can install instead. As GitLab is one of the more popular ones, we’ll cover installing and using it as an example. This is a bit more complex than the GitWeb option and likely requires more maintenance, but it is a much more fully featured option.

安裝

GitLab is a database-backed web application, so its installation is a bit more involved than some other git servers. Fortunately, this process is very well-documented and supported.

There are a few methods you can pursue to install GitLab. To get something up and running quickly, you can download a virtual machine image or a one-click installer from <https://bitnami.com/stack/gitlab>, and tweak the configuration to match your particular environment. One nice touch Bitnami has included is the login screen (accessed by typing alt→); it tells you the IP address and default username and password for the installed GitLab.



```
*** Welcome to the BitNami Gitlab Stack ***
*** Built using Ubuntu 12.04 - Kernel 3.2.0-53-virtual (tty2). ***

*** You can access the application at http://10.0.1.17 ***
*** The default username and password is 'user@example.com' and 'bitnami1'. ***
*** Please refer to http://wiki.bitnami.com/Virtual_Machines for details. ***

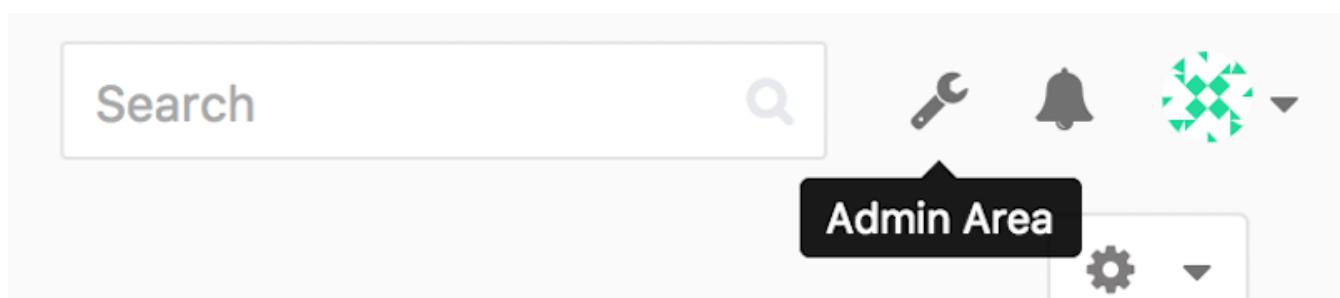
linux login: _
```

圖表 50. The Bitnami GitLab virtual machine login screen.

For anything else, follow the guidance in the GitLab Community Edition readme, which can be found at <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. There you'll find assistance for installing GitLab using Chef recipes, a virtual machine on Digital Ocean, and RPM and DEB packages (which, as of this writing, are in beta). There's also "unofficial" guidance on getting GitLab running with non-standard operating systems and databases, a fully-manual installation script, and many other topics.

管理員權限

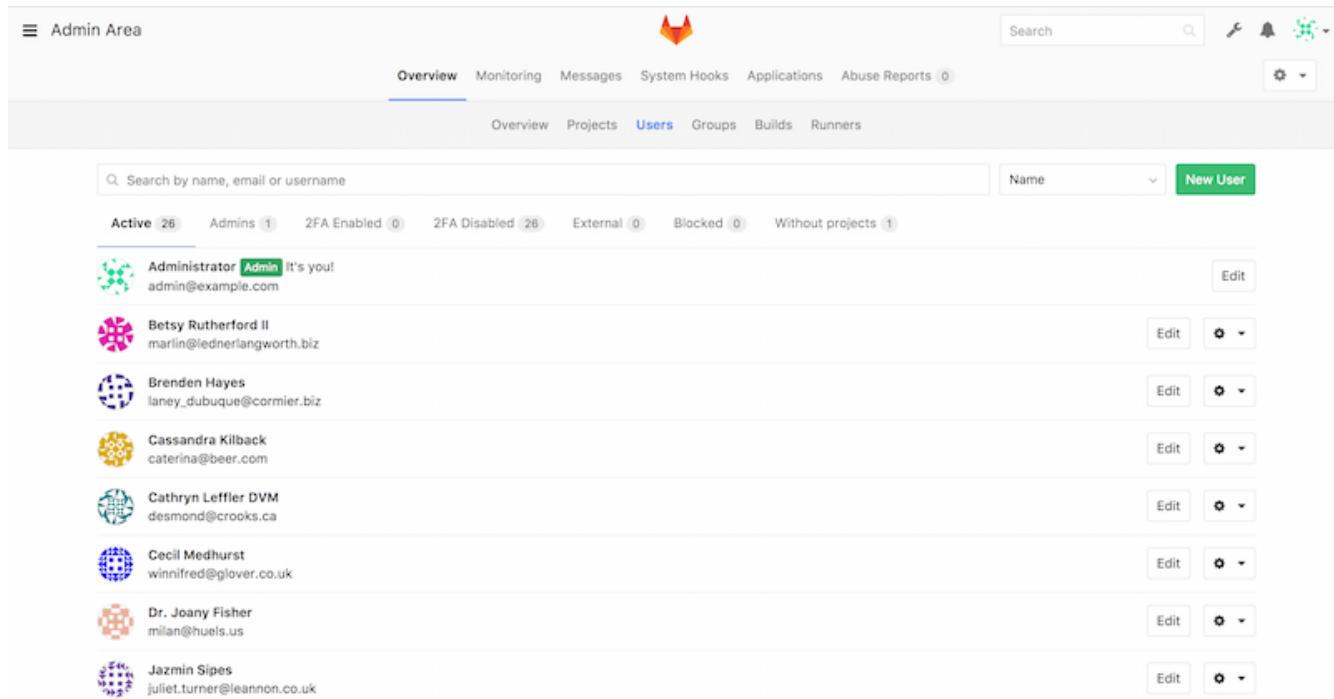
GitLab's administration interface is accessed over the web. Simply point your browser to the hostname or IP address where GitLab is installed, and log in as an admin user. The default username is `admin@local.host`, and the default password is `5iveL!fe` (which you will be prompted to change as soon as you enter it). Once logged in, click the "Admin area" icon in the menu at the top right.



圖表 51. The "Admin area" item in the GitLab menu.

使用者

Users in GitLab are accounts that correspond to people. User accounts don't have a lot of complexity; mainly it's a collection of personal information attached to login data. Each user account comes with a **namespace**, which is a logical grouping of projects that belong to that user. If the user **jane** had a project named **project**, that project's url would be <http://server/jane/project>.



The screenshot shows the GitLab user administration interface. At the top, there's a navigation bar with tabs for Overview, Monitoring, Messages, System Hooks, Applications, and Abuse Reports. Below that is a secondary navigation bar with tabs for Overview, Projects, Users (which is selected), Groups, Builds, and Runners. A search bar and a 'New User' button are also at the top. The main area lists users with their names, email addresses, and roles (e.g., Admin). Each user entry includes an 'Edit' button and a dropdown menu. The users listed are:

User	Role	Email	Action Buttons
Administrator	Admin	admin@example.com	Edit, ⚙️
Betsy Rutherford II		marlin@edinnerlangworth.biz	Edit, ⚙️
Brenden Hayes		laney_dubuque@cormier.biz	Edit, ⚙️
Cassandra Kilback		caterina@beer.com	Edit, ⚙️
Cathryn Leffler DVM		desmond@crooks.ca	Edit, ⚙️
Cecil Medhurst		winnifred@glover.co.uk	Edit, ⚙️
Dr. Joany Fisher		milan@huels.us	Edit, ⚙️
Jazmin Sipes		juliet.turner@leannon.co.uk	Edit, ⚙️

圖表 52. The GitLab user administration screen.

Removing a user can be done in two ways. “Blocking” a user prevents them from logging into the GitLab instance, but all of the data under that user’s namespace will be preserved, and commits signed with that user’s email address will still link back to their profile.

“Destroying” a user, on the other hand, completely removes them from the database and filesystem. All projects and data in their namespace is removed, and any groups they own will also be removed. This is obviously a much more permanent and destructive action, and its uses are rare.

群組

A GitLab group is an assemblage of projects, along with data about how users can access those projects. Each group has a project namespace (the same way that users do), so if the group **training** has a project **materials**, its url would be <http://server/training/materials>.

The screenshot shows the GitLab group administration interface for the group '@gitlab-org'. At the top, there's a navigation bar with links for Group, Activity, Labels, Milestones, Issues (8,501), Merge Requests (701), Members, and Contribution Analytics. Below the navigation is the group logo (@gitlab-org) and a tagline 'Open source software to collaborate on code'. There are buttons for 'Leave group' and 'Global'. A search bar labeled 'This group Search' is also present. The main content area displays a list of projects under the group, each with a small icon, the project name, a brief description, and two circular icons for more options.

All Projects	Shared Projects	Filter by name	Last updated
GitLab Development Kit Get started with GitLab Rails development			
kubernetes-gitlab-demo Idea to Production GitLab Demo running on Kubernetes			
omnibus-gitlab This project creates full-stack platform-specific downloadable packages for GitLab.			
GitLab Enterprise Edition GitLab Enterprise Edition			
gitlab-shell SSH access and repository management app for GitLab			
gitlab-ci-multi-runner GitLab Runner			

圖表 53. The GitLab group administration screen.

Each group is associated with a number of users, each of which has a level of permissions for the group’s projects and the group itself. These range from “Guest” (issues and chat only) to “Owner” (full control of the group, its members, and its projects). The types of permissions are too numerous to list here, but GitLab has a helpful link on the administration screen.

專案

A GitLab project roughly corresponds to a single git repository. Every project belongs to a single namespace, either a user or a group. If the project belongs to a user, the owner of the project has direct control over who has access to the project; if the project belongs to a group, the group’s user-level permissions will also take effect.

Every project also has a visibility level, which controls who has read access to that project’s pages and repository. If a project is *Private*, the project’s owner must explicitly grant access to specific users. An *Internal* project is visible to any logged-in user, and a *Public* project is visible to anyone. Note that this controls both git “fetch” access as well as access to the web UI for that project.

掛句

GitLab includes support for hooks, both at a project or system level. For either of these, the GitLab server will perform an HTTP POST with some descriptive JSON whenever relevant events occur. This is a great way to connect your git repositories and GitLab instance to the rest of your development automation, such as CI servers, chat rooms, or deployment tools.

基本使用

The first thing you’ll want to do with GitLab is create a new project. This is accomplished by clicking the “+” icon on the toolbar. You’ll be asked for the project’s name, which namespace it should belong to, and what its visibility level should be. Most of what you specify here isn’t permanent, and can be re-adjusted later through the settings interface. Click “Create Project”, and you’re done.

Once the project exists, you’ll probably want to connect it with a local Git repository. Each project is

accessible over HTTPS or SSH, either of which can be used to configure a Git remote. The URLs are visible at the top of the project’s home page. For an existing local repository, this command will create a remote named `gitlab` to the hosted location:

```
$ git remote add gitlab https://server/namespace/project.git
```

If you don’t have a local copy of the repository, you can simply do this:

```
$ git clone https://server/namespace/project.git
```

The web UI provides access to several useful views of the repository itself. Each project’s home page shows recent activity, and links along the top will lead you to views of the project’s files and commit log.

協同工作

The simplest way of working together on a GitLab project is by giving another user direct push access to the git repository. You can add a user to a project by going to the “Members” section of that project’s settings, and associating the new user with an access level (the different access levels are discussed a bit in [群組](#)). By giving a user an access level of “Developer” or above, that user can push commits and branches directly to the repository with impunity.

Another, more decoupled way of collaboration is by using merge requests. This feature enables any user that can see a project to contribute to it in a controlled way. Users with direct access can simply create a branch, push commits to it, and open a merge request from their branch back into `master` or any other branch. Users who don’t have push permissions for a repository can “fork” it (create their own copy), push commits to *that* copy, and open a merge request from their fork back to the main project. This model allows the owner to be in full control of what goes into the repository and when, while allowing contributions from untrusted users.

Merge requests and issues are the main units of long-lived discussion in GitLab. Each merge request allows a line-by-line discussion of the proposed change (which supports a lightweight kind of code review), as well as a general overall discussion thread. Both can be assigned to users, or organized into milestones.

This section is focused mainly on the Git-related features of GitLab, but as a mature project, it provides many other features to help your team work together, such as project wikis and system maintenance tools. One benefit to GitLab is that, once the server is set up and running, you’ll rarely need to tweak a configuration file or access the server via SSH; most administration and general usage can be accomplished through the in-browser interface.

第3方 Git 託管方案

If you don’t want to go through all of the work involved in setting up your own Git server, you have several options for hosting your Git projects on an external dedicated hosting site. Doing so offers a number of advantages: a hosting site is generally quick to set up and easy to start projects on, and no server maintenance or monitoring is involved. Even if you set up and run your own server internally, you may still want to use a public hosting site for your open source code – it’s generally easier for the open source community to find and help you with.

These days, you have a huge number of hosting options to choose from, each with different advantages and disadvantages. To see an up-to-date list, check out the GitHosting page on the main Git wiki at <https://git.wiki.kernel.org/index.php/GitHosting>

We'll cover using GitHub in detail in [GitHub](#), as it is the largest Git host out there and you may need to interact with projects hosted on it in any case, but there are dozens more to choose from should you not want to set up your own Git server.

總結

You have several options to get a remote Git repository up and running so that you can collaborate with others or share your work.

Running your own server gives you a lot of control and allows you to run the server within your own firewall, but such a server generally requires a fair amount of your time to set up and maintain. If you place your data on a hosted server, it's easy to set up and maintain; however, you have to be able to keep your code on someone else's servers, and some organizations don't allow that.

It should be fairly straightforward to determine which solution or combination of solutions is appropriate for you and your organization.

分散式的 Git

Now that you have a remote Git repository set up as a point for all the developers to share their code, and you’re familiar with basic Git commands in a local workflow, you’ll look at how to utilize some of the distributed workflows that Git affords you.

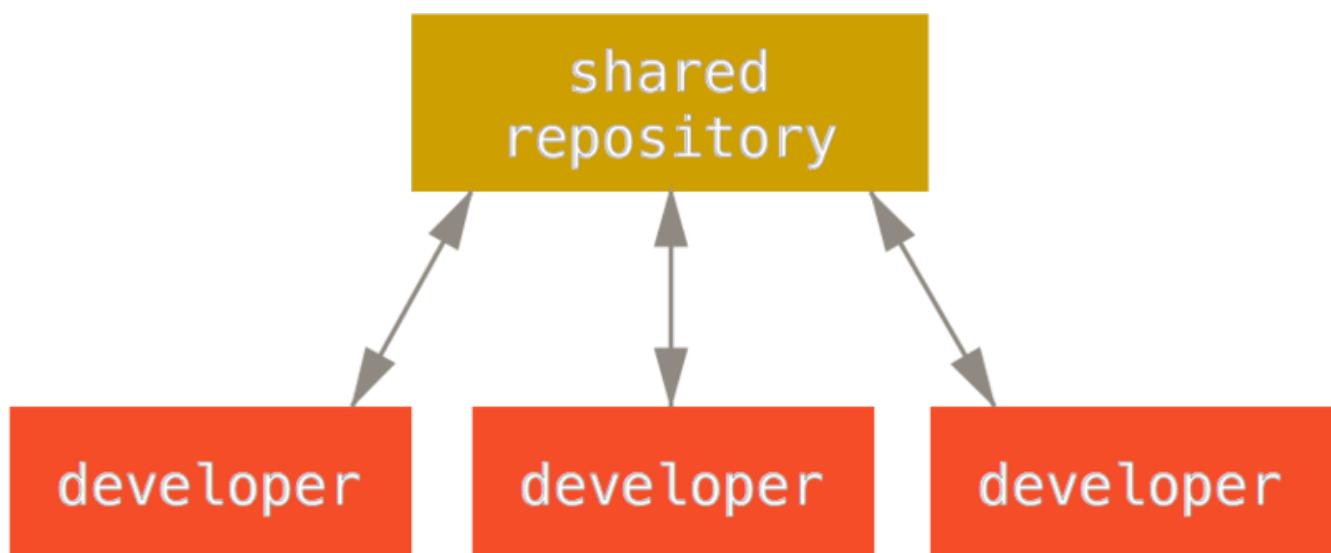
In this chapter, you’ll see how to work with Git in a distributed environment as a contributor and an integrator. That is, you’ll learn how to contribute code successfully to a project and make it as easy on you and the project maintainer as possible, and also how to maintain a project successfully with a number of developers contributing.

分散式工作流程

Unlike Centralized Version Control Systems (CVCSs), the distributed nature of Git allows you to be far more flexible in how developers collaborate on projects. In centralized systems, every developer is a node working more or less equally on a central hub. In Git, however, every developer is potentially both a node and a hub – that is, every developer can both contribute code to other repositories and maintain a public repository on which others can base their work and which they can contribute to. This opens a vast range of workflow possibilities for your project and/or your team, so we’ll cover a few common paradigms that take advantage of this flexibility. We’ll go over the strengths and possible weaknesses of each design; you can choose a single one to use, or you can mix and match features from each.

集中式工作流程

In centralized systems, there is generally a single collaboration model—the centralized workflow. One central hub, or repository, can accept code, and everyone synchronizes their work to it. A number of developers are nodes – consumers of that hub – and synchronize to that one place.



圖表 54. Centralized workflow.

This means that if two developers clone from the hub and both make changes, the first developer to push their changes back up can do so with no problems. The second developer must merge in the first one’s work before pushing changes up, so as not to overwrite the first developer’s changes. This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

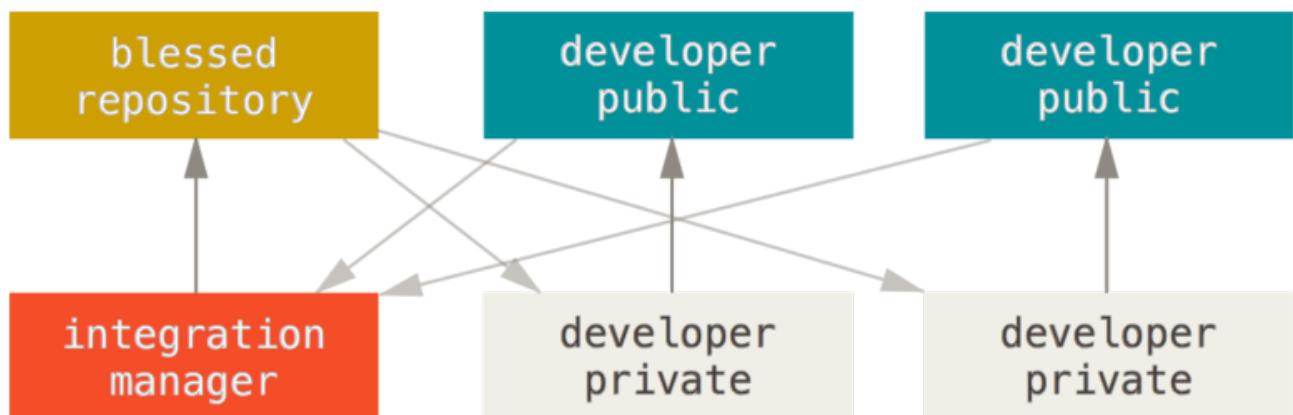
If you are already comfortable with a centralized workflow in your company or team, you can easily continue using that workflow with Git. Simply set up a single repository, and give everyone on your team push access; Git won't let users overwrite each other. Say John and Jessica both start working at the same time. John finishes his change and pushes it to the server. Then Jessica tries to push her changes, but the server rejects them. She is told that she's trying to push non-fast-forward changes and that she won't be able to do so until she fetches and merges. This workflow is attractive to a lot of people because it's a paradigm that many are familiar and comfortable with.

This is also not limited to small teams. With Git's branching model, it's possible for hundreds of developers to successfully work on a single project through dozens of branches simultaneously.

整合式管理員工作流程

Because Git allows you to have multiple remote repositories, it's possible to have a workflow where each developer has write access to their own public repository and read access to everyone else's. This scenario often includes a canonical repository that represents the "official" project. To contribute to that project, you create your own public clone of the project and push your changes to it. Then, you can send a request to the maintainer of the main project to pull in your changes. The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository. The process works as follows (see [Integration-manager workflow](#).):

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor's repo as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.



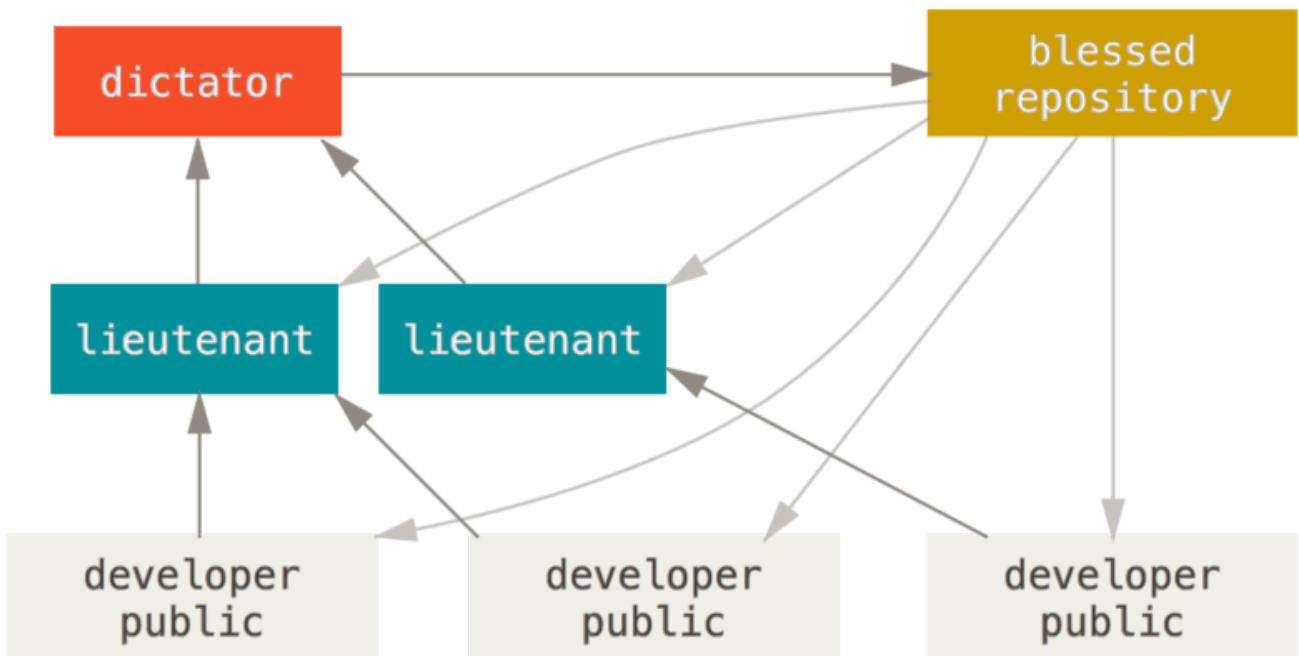
圖表 55. *Integration-manager workflow*.

This is a very common workflow with hub-based tools like GitHub or GitLab, where it's easy to fork a project and push your changes into your fork for everyone to see. One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time. Contributors don't have to wait for the project to incorporate their changes – each party can work at their own pace.

司令官與副官工作流程

This is a variant of a multiple-repository workflow. It's generally used by huge projects with hundreds of collaborators; one famous example is the Linux kernel. Various integration managers are in charge of certain parts of the repository; they're called lieutenants. All the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator's repository serves as the reference repository from which all the collaborators need to pull. The process works like this (see [Benevolent dictator workflow](#)):

1. Regular developers work on their topic branch and rebase their work on top of `master`. The `master` branch is that of the dictator.
2. Lieutenants merge the developers' topic branches into their `master` branch.
3. The dictator merges the lieutenants' `master` branches into the dictator's `master` branch.
4. The dictator pushes their `master` to the reference repository so the other developers can rebase on it.



圖表 56. Benevolent dictator workflow.

This kind of workflow isn't common, but can be useful in very big projects, or in highly hierarchical environments. It allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them.

工作流程總結

These are some commonly used workflows that are possible with a distributed system like Git, but you can see that many variations are possible to suit your particular real-world workflow. Now that you can (hopefully) determine which workflow combination may work for you, we'll cover some more specific examples of how to accomplish the main roles that make up the different flows. In the next section, you'll learn about a few common patterns for contributing to a project.

對專案進行貢獻

The main difficulty with describing how to contribute to a project is that there are a huge number of variations on how it's done. Because Git is very flexible, people can and do work together in many ways, and it's problematic to describe how you should contribute – every project is a bit different. Some of the variables involved are active contributor count, chosen workflow, your commit access, and possibly the external contribution method.

The first variable is active contributor count – how many users are actively contributing code to this project, and how often? In many instances, you'll have two or three developers with a few commits a day, or possibly less for somewhat dormant projects. For larger companies or projects, the number of developers could be in the thousands, with hundreds or thousands of commits coming in each day. This is important because with more and more developers, you run into more issues with making sure your code applies cleanly or can be easily merged. Changes you submit may be rendered obsolete or severely broken by work that is merged in while you were working or while your changes were waiting to be approved or applied. How can you keep your code consistently up to date and your commits valid?

The next variable is the workflow in use for the project. Is it centralized, with each developer having equal write access to the main codeline? Does the project have a maintainer or integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your work to them first?

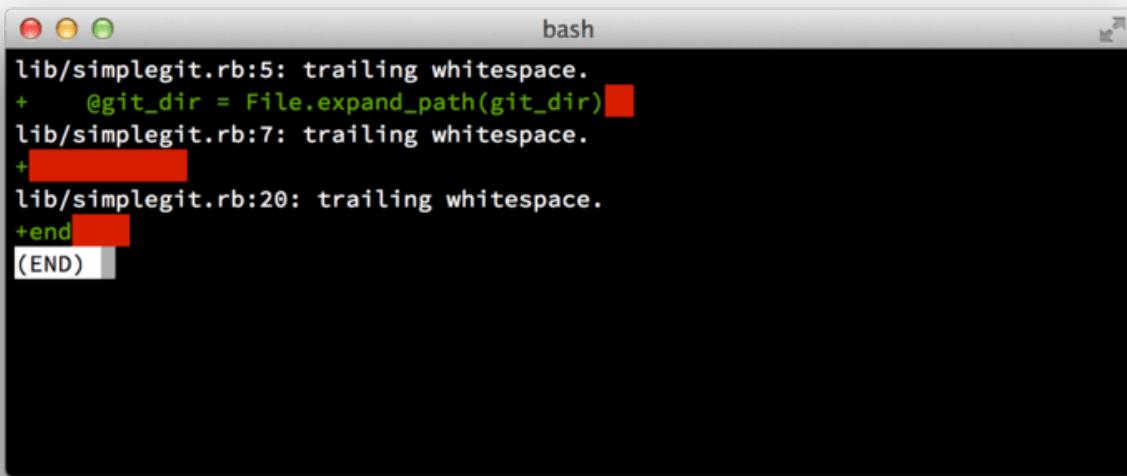
The next issue is your commit access. The workflow required in order to contribute to a project is much different if you have write access to the project than if you don't. If you don't have write access, how does the project prefer to accept contributed work? Does it even have a policy? How much work are you contributing at a time? How often do you contribute?

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available to you. We'll cover aspects of each of these in a series of use cases, moving from simple to more complex; you should be able to construct the specific workflows you need in practice from these examples.

提交指南

Before we start looking at the specific use cases, here's a quick note about commit messages. Having a good guideline for creating commits and sticking to it makes working with Git and collaborating with others a lot easier. The Git project provides a document that lays out a number of good tips for creating commits from which to submit patches – you can read it in the Git source code in the [Documentation/SubmittingPatches](#) file.

First, you don't want to submit any whitespace errors. Git provides an easy way to check for this – before you commit, run `git diff --check`, which identifies possible whitespace errors and lists them for you.



```
lib/simplegit.rb:5: trailing whitespace.  
+ @git_dir = File.expand_path(git_dir)  
lib/simplegit.rb:7: trailing whitespace.  
+  
lib/simplegit.rb:20: trailing whitespace.  
+end  
(END)
```

圖表 57. Output of `git diff --check`.

If you run that command before committing, you can tell if you’re about to commit whitespace issues that may annoy other developers.

Next, try to make each commit a logically separate changeset. If you can, try to make your changes digestible – don’t code for a whole weekend on five different issues and then submit them all as one massive commit on Monday. Even if you don’t commit during the weekend, use the staging area on Monday to split your work into at least one commit per issue, with a useful message per commit. If some of the changes modify the same file, try to use `git add --patch` to partially stage files (covered in detail in [Interactive Staging](#)). The project snapshot at the tip of the branch is identical whether you do one commit or five, as long as all the changes are added at some point, so try to make things easier on your fellow developers when they have to review your changes. This approach also makes it easier to pull out or revert one of the changesets if you need to later. [Rewriting History](#) describes a number of useful Git tricks for rewriting history and interactively staging files – use these tools to help craft a clean and understandable history before sending the work to someone else.

The last thing to keep in mind is the commit message. Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier. As a general rule, your messages should start with a single line that’s no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation. The Git project requires that the more detailed explanation include your motivation for the change and contrast its implementation with previous behavior – this is a good guideline to follow. It’s also a good idea to use the imperative present tense in these messages. In other words, use commands. Instead of “I added tests for” or “Adding tests for,” use “Add tests for.” Here is a template originally written by Tim Pope:

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

If all your commit messages look like this, things will be a lot easier for you and the developers you work with. The Git project has well-formatted commit messages – try running `git log --no-merges` there to see what a nicely formatted project-commit history looks like.

In the following examples, and throughout most of this book, for the sake of brevity this book doesn't have nicely-formatted messages like this; instead, we use the `-m` option to `git commit`. Do as we say, not as we do.

私有的小團隊

The simplest setup you're likely to encounter is a private project with one or two other developers. "Private," in this context, means closed-source – not accessible to the outside world. You and the other developers all have push access to the repository.

In this environment, you can follow a workflow similar to what you might do when using Subversion or another centralized system. You still get the advantages of things like offline committing and vastly simpler branching and merging, but the workflow can be very similar; the main difference is that merges happen client-side rather than on the server at commit time. Let's see what it might look like when two developers start to work together with a shared repository. The first developer, John, clones the repository, makes a change, and commits locally. (The protocol messages have been replaced with `...` in these examples to shorten them somewhat.)

```
# John's Machine
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The second developer, Jessica, does the same thing – clones the repository and commits a change:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Now, Jessica pushes her work up to the server:

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
 1edee6b..fbff5bc  master -> master
```

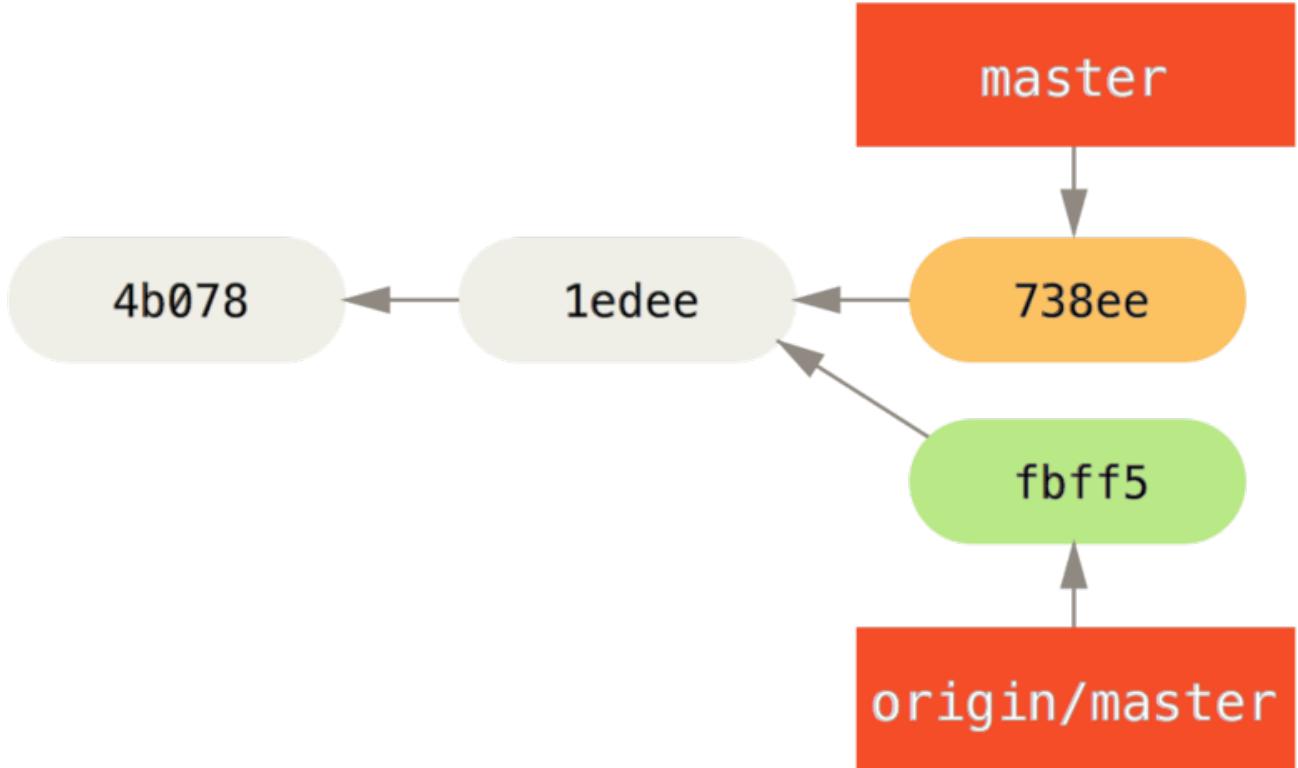
John tries to push his change up, too:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]      master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

John isn't allowed to push because Jessica has pushed in the meantime. This is especially important to understand if you're used to Subversion, because you'll notice that the two developers didn't edit the same file. Although Subversion automatically does such a merge on the server if different files are edited, in Git you must merge the commits locally. John has to fetch Jessica's changes and merge them in before he will be allowed to push:

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master      -> origin/master
```

At this point, John's local repository looks something like this:

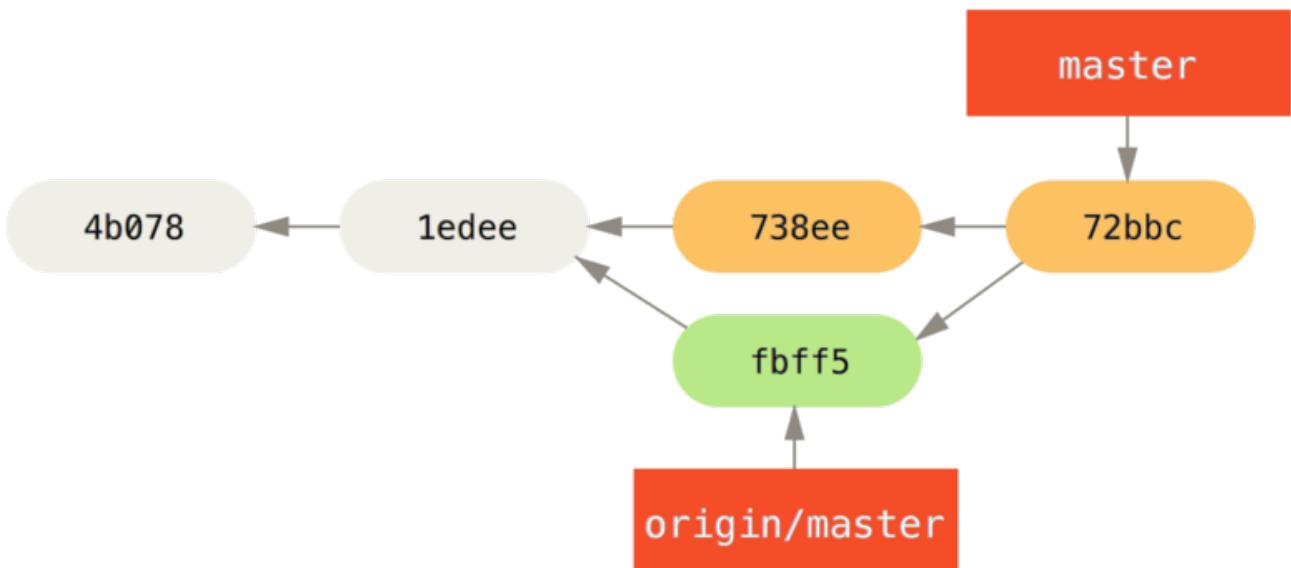


圖表 58. John' s divergent history.

John has a reference to the changes Jessica pushed up, but he has to merge them into his own work before he is allowed to push:

```
$ git merge origin/master
Merge made by recursive.
TODO | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

The merge goes smoothly – John' s commit history now looks like this:

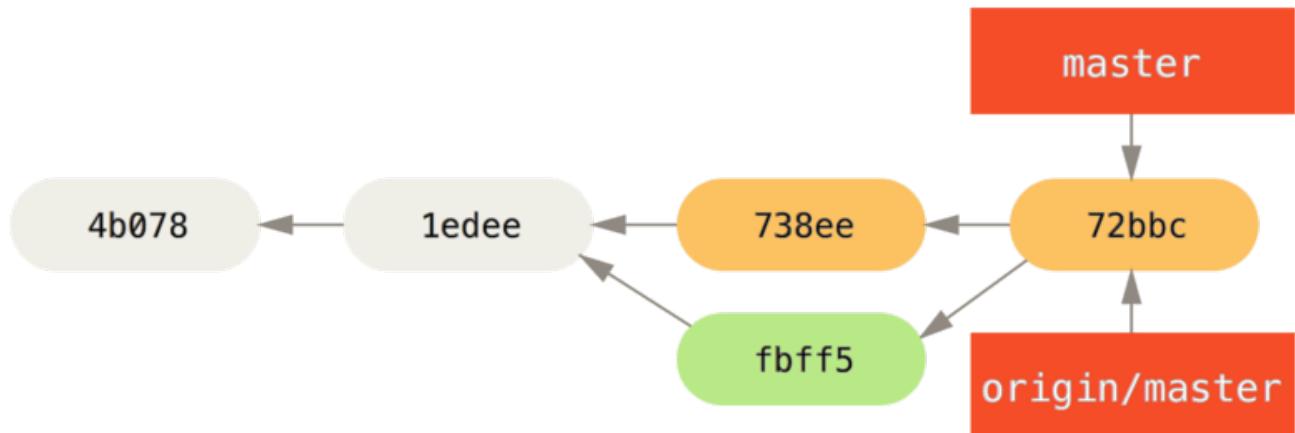


圖表 59. John' s repository after merging origin/master.

Now, John can test his code to make sure it still works properly, and then he can push his new merged work up to the server:

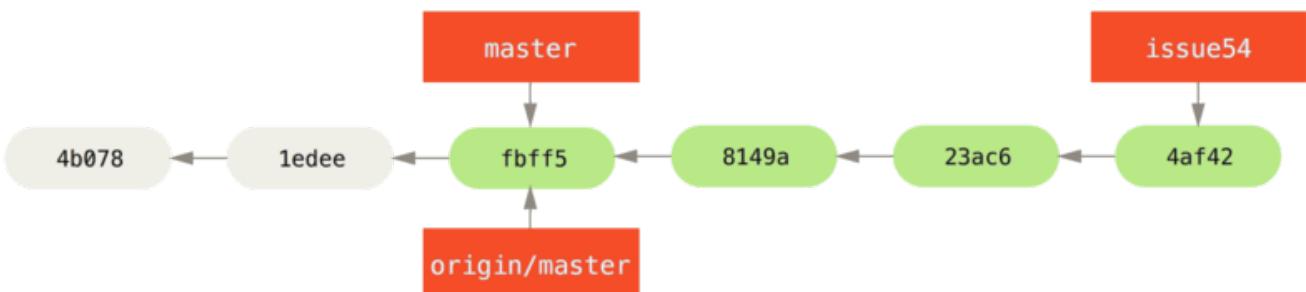
```
$ git push origin master
...
To john@githost:simplegit.git
  fbff5bc..72bbc59  master -> master
```

Finally, John's commit history looks like this:



圖表 60. John's history after pushing to the origin server.

In the meantime, Jessica has been working on a topic branch. She's created a topic branch called **issue54** and done three commits on that branch. She hasn't fetched John's changes yet, so her commit history looks like this:

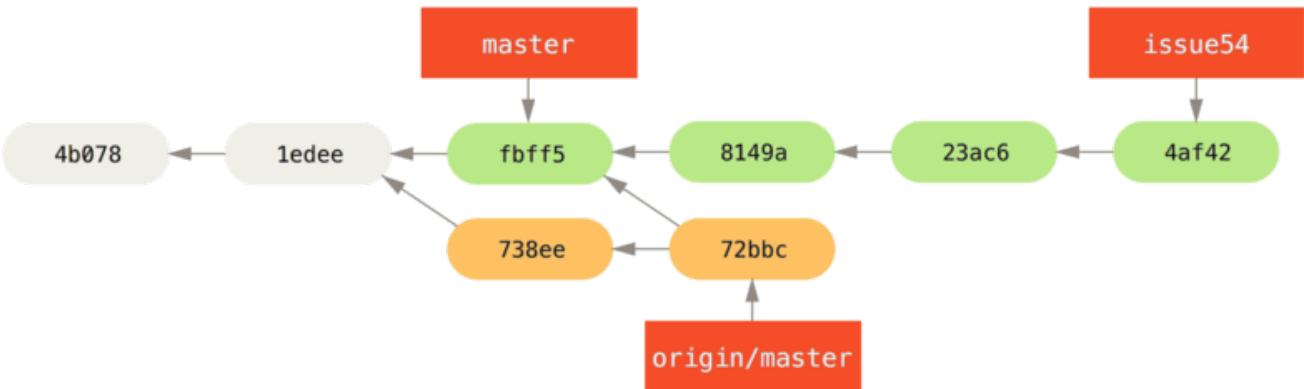


圖表 61. Jessica's topic branch.

Jessica wants to sync up with John, so she fetches:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
  fbff5bc..72bbc59  master      -> origin/master
```

That pulls down the work John has pushed up in the meantime. Jessica's history now looks like this:



圖表 62. Jessica' s history after fetching John' s changes.

Jessica thinks her topic branch is ready, but she wants to know what she has to merge into her work so that she can push. She runs `git log` to find out:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    removed invalid default value
```

The `issue54..origin/master` syntax is a log filter that asks Git to only show the list of commits that are on the latter branch (in this case `origin/master`) that are not on the first branch (in this case `issue54`). We'll go over this syntax in detail in [Commit Ranges](#).

For now, we can see from the output that there is a single commit that John has made that Jessica has not merged in. If she merges `origin/master`, that is the single commit that will modify her local work.

Now, Jessica can merge her topic work into her master branch, merge John's work (`origin/master`) into her `master` branch, and then push back to the server again. First, she switches back to her master branch to integrate all this work:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-
forwarded.
```

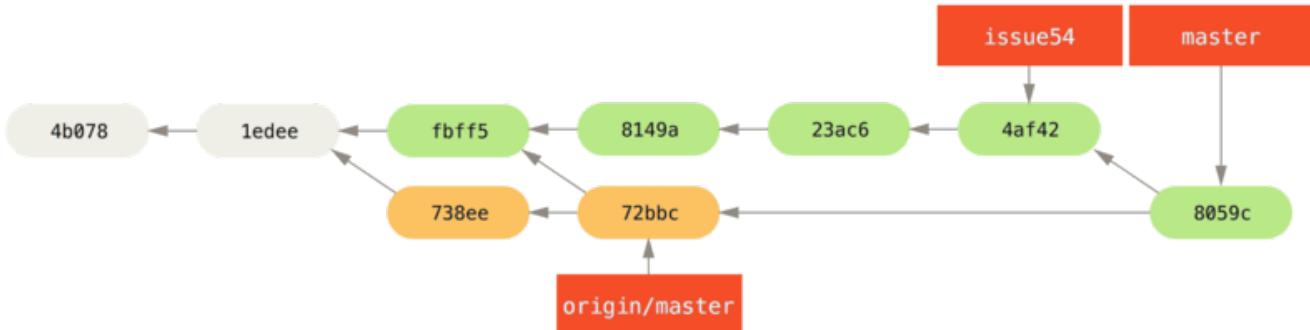
She can merge either `origin/master` or `issue54` first – they're both upstream, so the order doesn't matter. The end snapshot should be identical no matter which order she chooses; only the history will be slightly different. She chooses to merge in `issue54` first:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README          |      1 +
 lib/simplegit.rb |     6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

No problems occur; as you can see it was a simple fast-forward. Now Jessica merges in John's work (`origin/master`):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Everything merges cleanly, and Jessica's history looks like this:

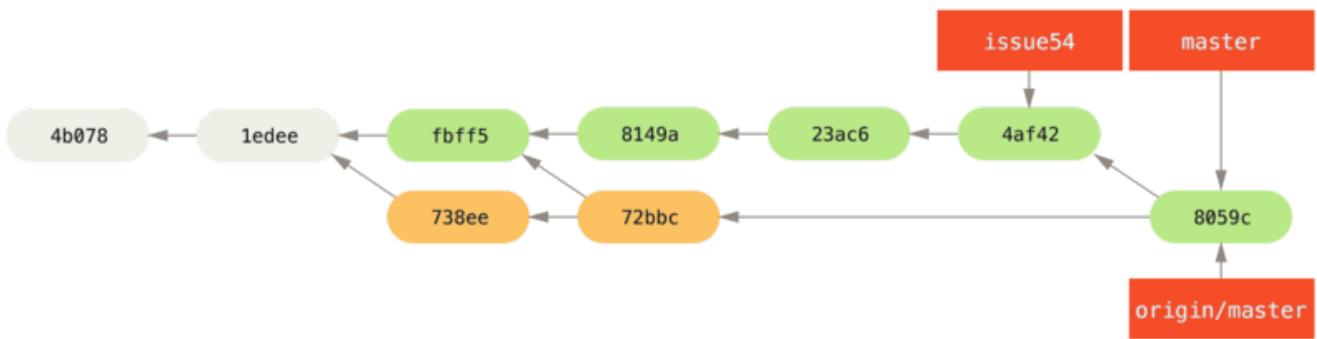


圖表 63. Jessica's history after merging John's changes.

Now `origin/master` is reachable from Jessica's `master` branch, so she should be able to successfully push (assuming John hasn't pushed again in the meantime):

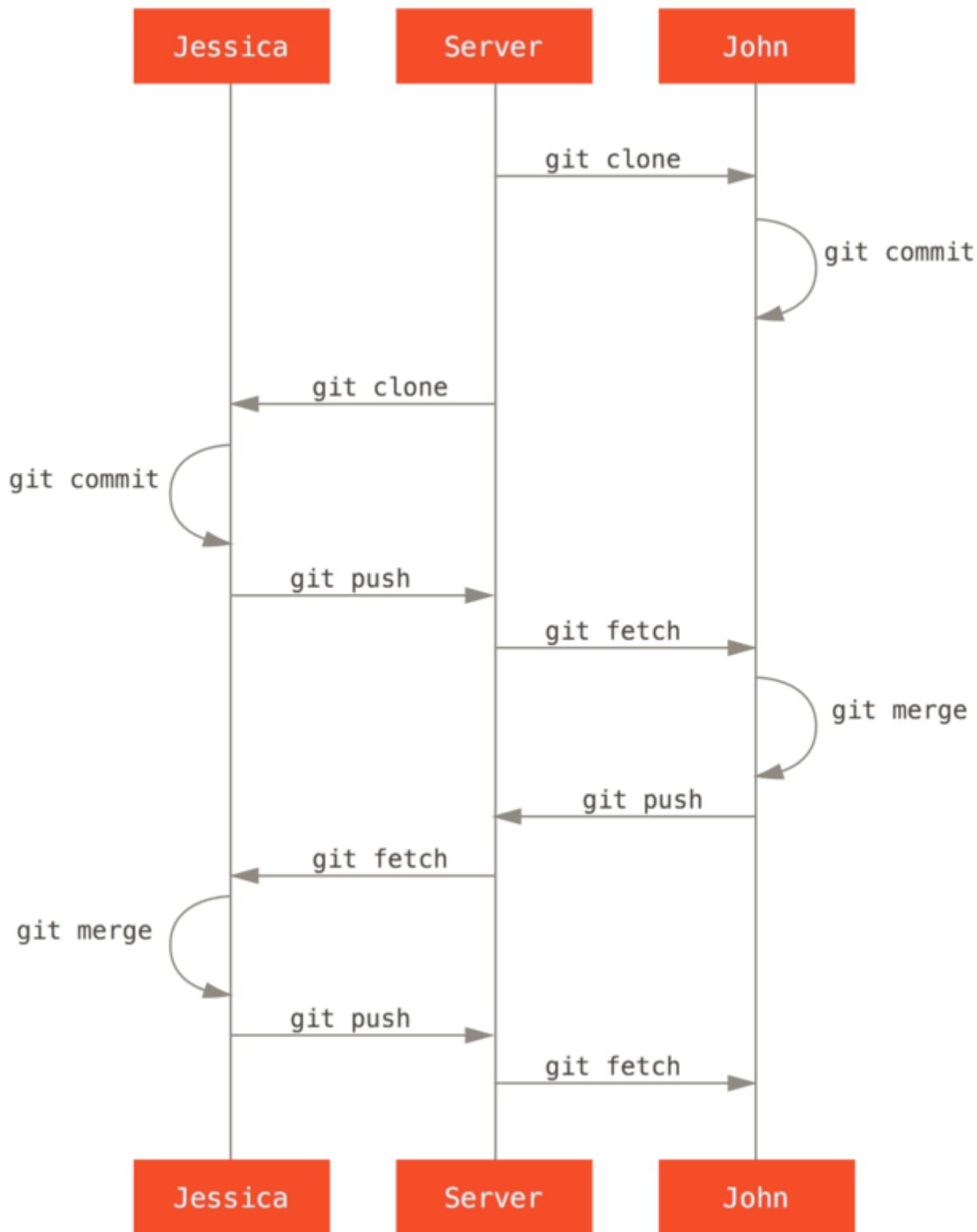
```
$ git push origin master
...
To jessica@githost:simplegit.git
 72bbc59..8059c15  master -> master
```

Each developer has committed a few times and merged each other's work successfully.



圖表 64. Jessica's history after pushing all changes back to the server.

That is one of the simplest workflows. You work for a while, generally in a topic branch, and merge into your master branch when it's ready to be integrated. When you want to share that work, you fetch and merge your master from `origin/master` if it has changed, and finally push to the `master` branch on the server. The general sequence is something like this:



圖表 65. General sequence of events for a simple multiple-developer Git workflow.

私有團隊

In this next scenario, you'll look at contributor roles in a larger private group. You'll learn how to work in an environment where small groups collaborate on features and then those team-based contributions are integrated by another party.

Let's say that John and Jessica are working together on one feature, while Jessica and Josie are working on a second. In this case, the company is using a type of integration-manager workflow where

the work of the individual groups is integrated only by certain engineers, and the `master` branch of the main repo can be updated only by those engineers. In this scenario, all work is done in team-based branches and pulled together by the integrators later.

Let's follow Jessica's workflow as she works on her two features, collaborating in parallel with two different developers in this environment. Assuming she already has her repository cloned, she decides to work on `featureA` first. She creates a new branch for the feature and does some work on it there:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

At this point, she needs to share her work with John, so she pushes her `featureA` branch commits up to the server. Jessica doesn't have push access to the `master` branch – only the integrators do – so she has to push to another branch in order to collaborate with John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

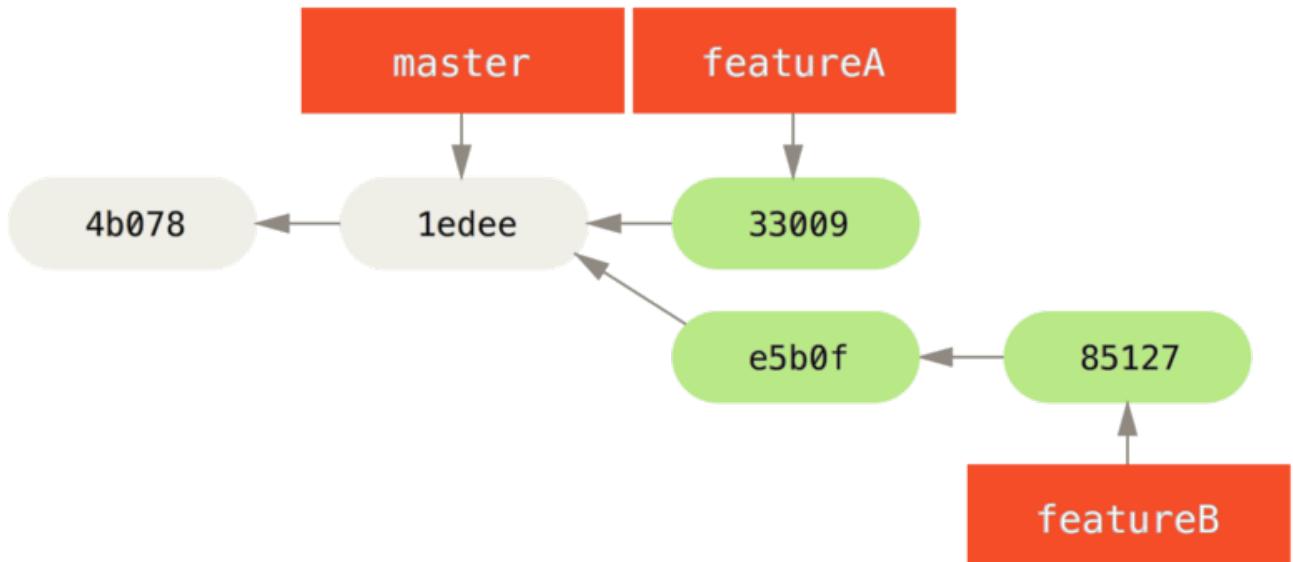
Jessica emails John to tell him that she's pushed some work into a branch named `featureA` and he can look at it now. While she waits for feedback from John, Jessica decides to start working on `featureB` with Josie. To begin, she starts a new feature branch, basing it off the server's `master` branch:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Now, Jessica makes a couple of commits on the `featureB` branch:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessica's repository looks like this:



圖表 66. Jessica’ s initial commit history.

She’s ready to push up her work, but gets an email from Josie that a branch with some initial work on it was already pushed to the server as `featureBee`. Jessica first needs to merge those changes in with her own before she can push to the server. She can then fetch Josie’s changes down with `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica can now merge this into the work she did with `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 +---
 1 files changed, 4 insertions(+), 0 deletions(-)
```

There is a bit of a problem – she needs to push the merged work in her `featureB` branch to the `featureBee` branch on the server. She can do so by specifying the local branch followed by a colon (:) followed by the remote branch to the `git push` command:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
 fba9af8..cd685d1  featureB -> featureBee
```

This is called a *refspec*. See [The Refspec](#) for a more detailed discussion of Git refspecs and different things you can do with them. Also notice the `-u` flag; this is short for `--set-upstream`, which configures the branches for easier pushing and pulling later.

Next, John emails Jessica to say he's pushed some changes to the `featureA` branch and asks her to verify them. She runs a `git fetch` to pull down those changes:

```
$ git fetch origin  
...  
From jessica@githost:simplegit  
 3300904..aad881d  featureA    -> origin/featureA
```

Then, she can see what has been changed with `git log`:

```
$ git log featureA..origin/featureA  
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6  
Author: John Smith <jsmith@example.com>  
Date:   Fri May 29 19:57:33 2009 -0700  
  
        changed log output to 30 from 25
```

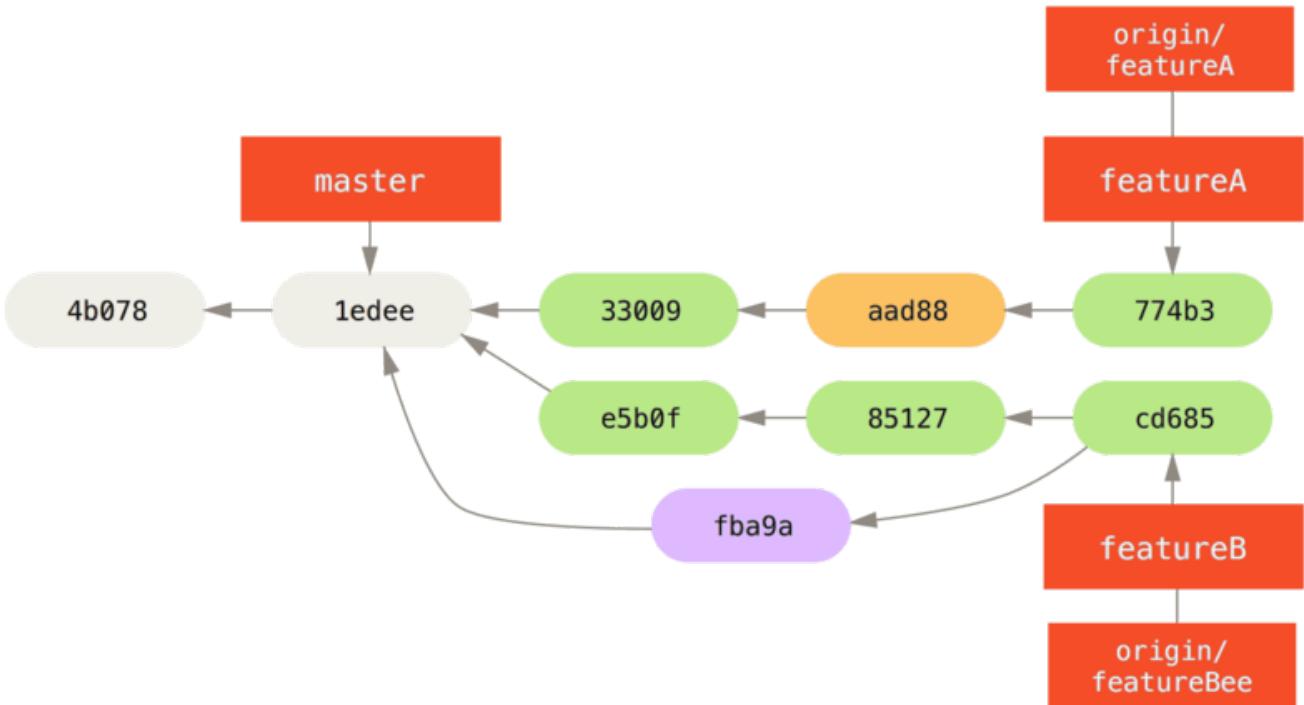
Finally, she merges John's work into her own `featureA` branch:

```
$ git checkout featureA  
Switched to branch 'featureA'  
$ git merge origin/featureA  
Updating 3300904..aad881d  
Fast forward  
 lib/simplegit.rb |    10 ++++++++-  
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica wants to tweak something, so she commits again and then pushes this back up to the server:

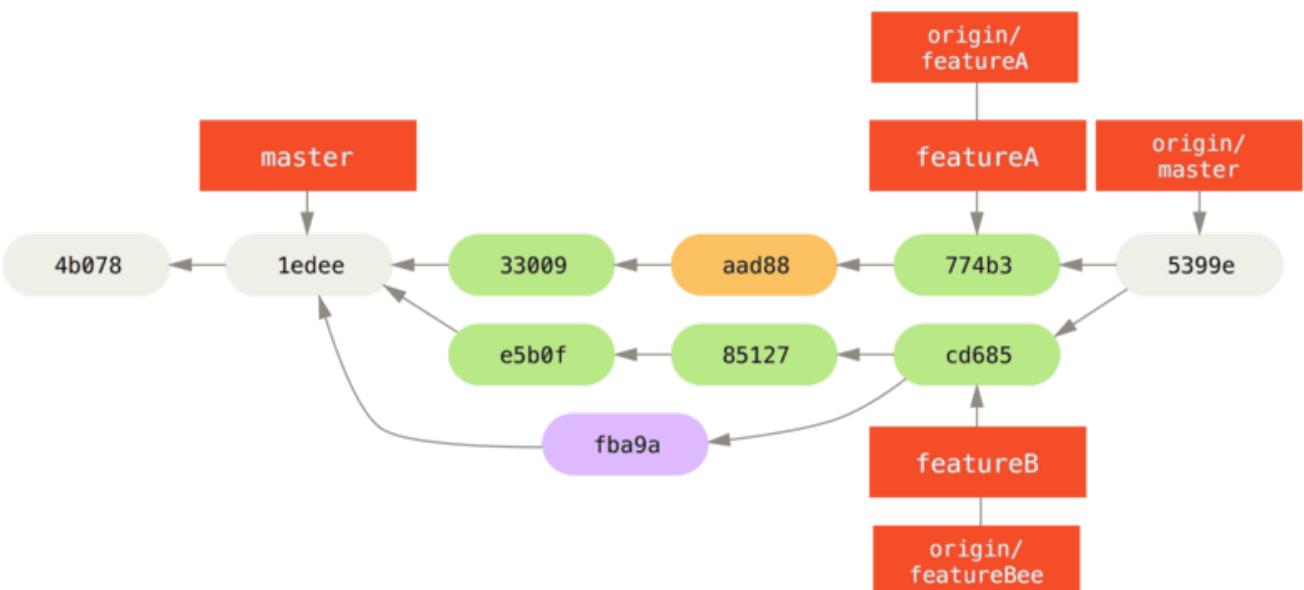
```
$ git commit -am 'small tweak'  
[featureA 774b3ed] small tweak  
 1 files changed, 1 insertions(+), 1 deletions(-)  
$ git push  
...  
To jessica@githost:simplegit.git  
 3300904..774b3ed  featureA -> featureA
```

Jessica's commit history now looks something like this:



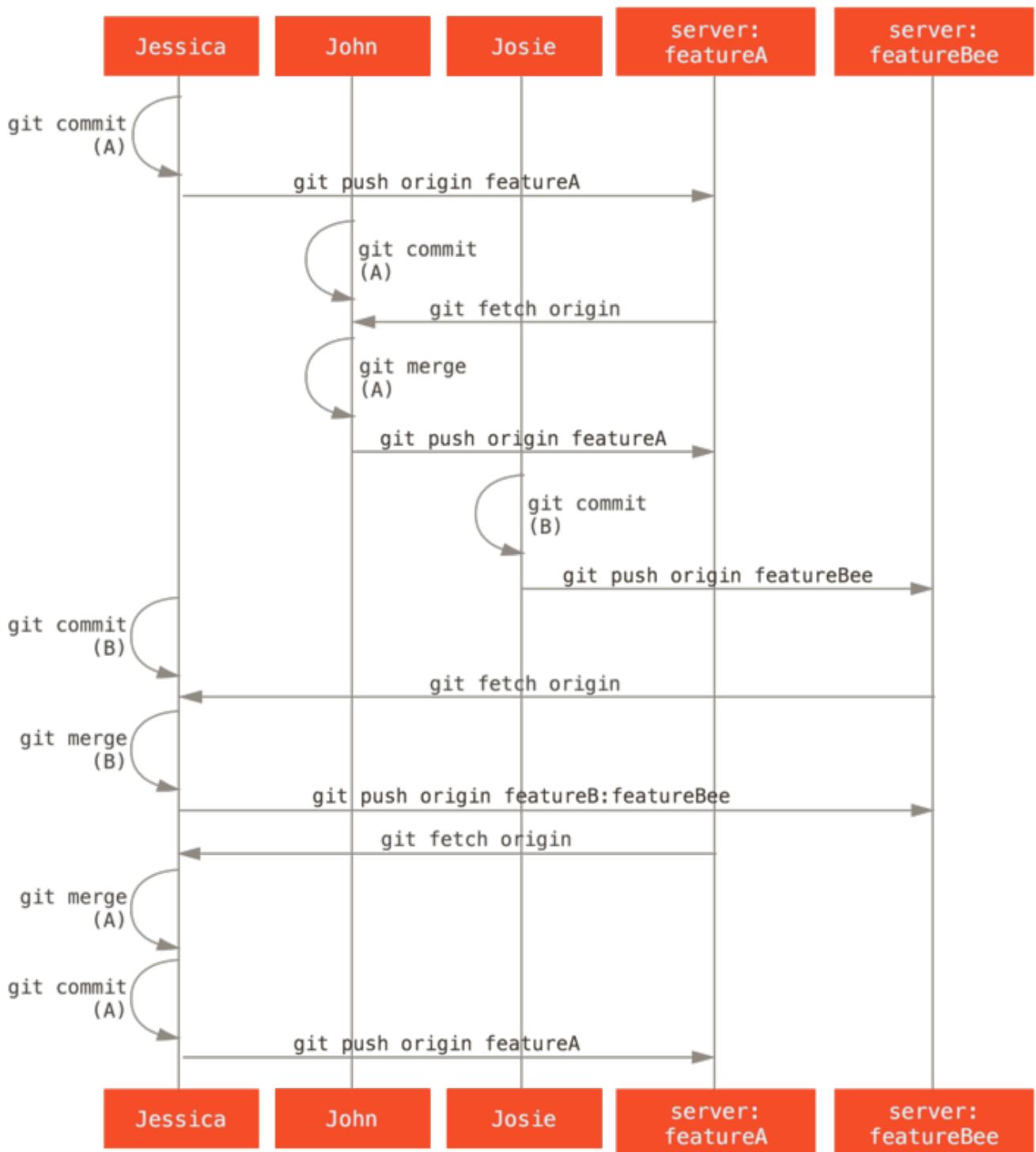
圖表 67. Jessica’ s history after committing on a feature branch.

Jessica, Josie, and John inform the integrators that the `featureA` and `featureBee` branches on the server are ready for integration into the mainline. After the integrators merge these branches into the mainline, a fetch will bring down the new merge commit, making the history look like this:



圖表 68. Jessica’ s history after merging both her topic branches.

Many groups switch to Git because of this ability to have multiple teams working in parallel, merging the different lines of work late in the process. The ability of smaller subgroups of a team to collaborate via remote branches without necessarily having to involve or impede the entire team is a huge benefit of Git. The sequence for the workflow you saw here is something like this:



圖表 69. Basic sequence of this managed-team workflow.

Fork 公眾專案

Contributing to public projects is a bit different. Because you don't have the permissions to directly update branches on the project, you have to get the work to the maintainers some other way. This first example describes contributing via forking on Git hosts that support easy forking. Many hosting sites support this (including GitHub, BitBucket, Google Code, repo.or.cz, and others), and many project maintainers expect this style of contribution. The next section deals with projects that prefer to accept contributed patches via email.

First, you'll probably want to clone the main repository, create a topic branch for the patch or patch series you're planning to contribute, and do your work there. The sequence looks basically like this:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

筆記 You may want to use `rebase -i` to squash your work down to a single commit, or rearrange the work in the commits to make the patch easier for the maintainer to review – see [Rewriting History](#) for more information about interactive rebasing.

When your branch work is finished and you’re ready to contribute it back to the maintainers, go to the original project page and click the “Fork” button, creating your own writable fork of the project. You then need to add in this new repository URL as a second remote, in this case named `myfork`:

```
$ git remote add myfork (url)
```

Then you need to push your work up to it. It’s easiest to push the topic branch you’re working on up to your repository, rather than merging into your master branch and pushing that up. The reason is that if the work isn’t accepted or is cherry picked, you don’t have to rewind your master branch. If the maintainers merge, rebase, or cherry-pick your work, you’ll eventually get it back via pulling from their repository anyhow:

```
$ git push -u myfork featureA
```

When your work has been pushed up to your fork, you need to notify the maintainer. This is often called a pull request, and you can either generate it via the website – GitHub has its own Pull Request mechanism that we’ll go over in [GitHub](#) – or you can run the `git request-pull` command and email the output to the project maintainer manually.

The `request-pull` command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and outputs a summary of all the changes you’re asking to be pulled in. For instance, if Jessica wants to send John a pull request, and she’s done two commits on the topic branch she just pushed up, she can run this:

```

$ git request-pull origin/master myfork
The following changes since commit
1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

  Jessica Smith (2):
    add limit to log function
    change log output to 30 from 25

  lib/simplegit.rb |   10 ++++++----
  1 files changed, 9 insertions(+), 1 deletions(-)

```

The output can be sent to the maintainer – it tells them where the work was branched from, summarizes the commits, and tells where to pull this work from.

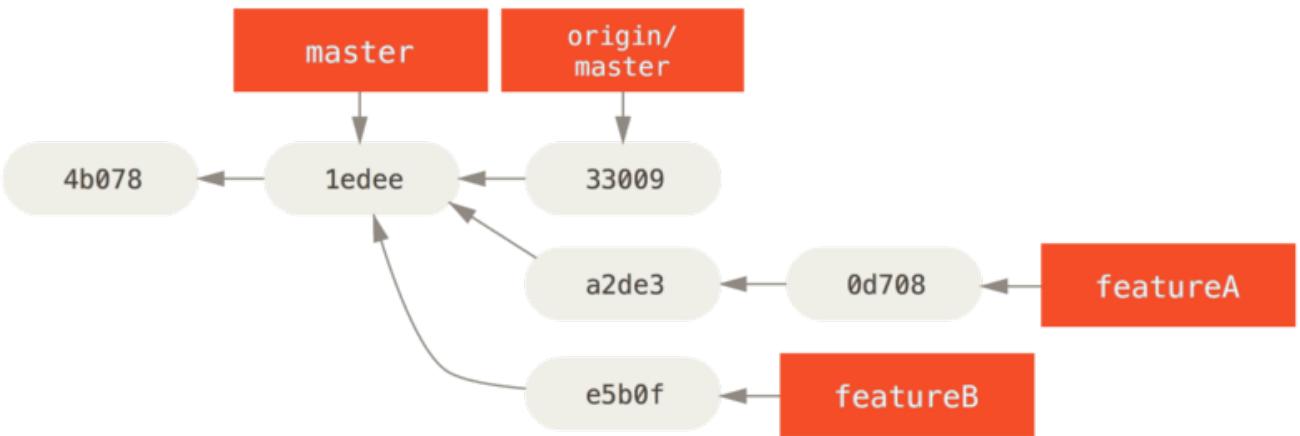
On a project for which you’re not the maintainer, it’s generally easier to have a branch like **master** always track **origin/master** and to do your work in topic branches that you can easily discard if they’re rejected. Having work themes isolated into topic branches also makes it easier for you to rebase your work if the tip of the main repository has moved in the meantime and your commits no longer apply cleanly. For example, if you want to submit a second topic of work to the project, don’t continue working on the topic branch you just pushed up – start over from the main repository’s **master** branch:

```

$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin

```

Now, each of your topics is contained within a silo – similar to a patch queue – that you can rewrite, rebase, and modify without the topics interfering or interdepending on each other, like so:

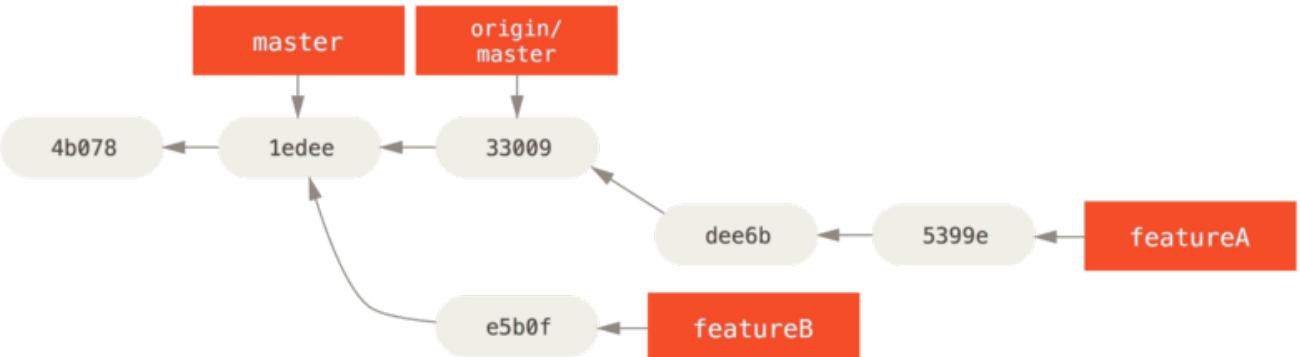


圖表 70. Initial commit history with **featureB** work.

Let's say the project maintainer has pulled in a bunch of other patches and tried your first branch, but it no longer cleanly merges. In this case, you can try to rebase that branch on top of **origin/master**, resolve the conflicts for the maintainer, and then resubmit your changes:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

This rewrites your history to now look like Commit history after **featureA** work..



圖表 71. Commit history after **featureA** work.

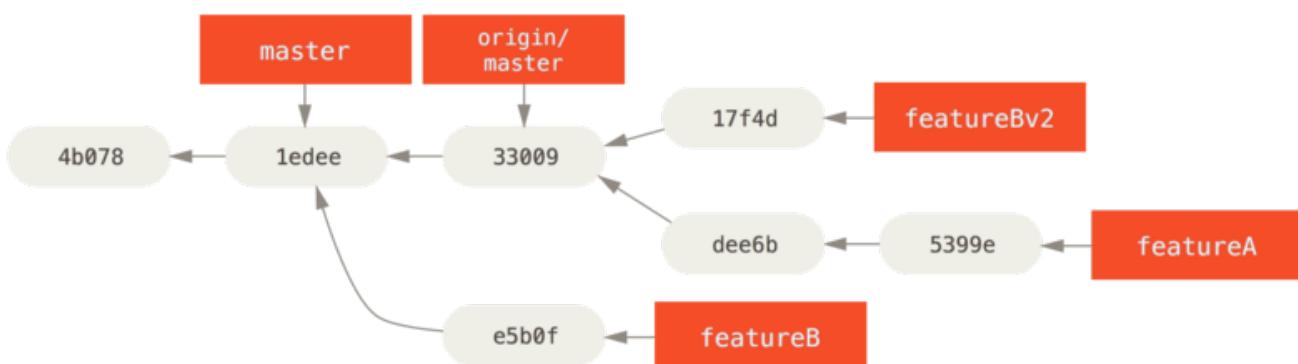
Because you rebased the branch, you have to specify the **-f** to your push command in order to be able to replace the **featureA** branch on the server with a commit that isn't a descendant of it. An alternative would be to push this new work to a different branch on the server (perhaps called **featureAv2**).

Let's look at one more possible scenario: the maintainer has looked at work in your second branch and likes the concept but would like you to change an implementation detail. You'll also take this opportunity to move the work to be based off the project's current **master** branch. You start a new branch based off the current **origin/master** branch, squash the **featureB** changes there, resolve any conflicts, make the implementation change, and then push that up as a new branch:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

The `--squash` option takes all the work on the merged branch and squashes it into one changeset producing the repository state as if a real merge happened, without actually making a merge commit. This means your future commit will have one parent only and allows you to introduce all the changes from another branch and then make more changes before recording the new commit. Also the `--no-commit` option can be useful to delay the merge commit in case of the default merge process.

Now you can send the maintainer a message that you've made the requested changes and they can find those changes in your `featureBv2` branch.



圖表 72. Commit history after `featureBv2` work.

透過電子郵件貢獻到公眾專案

Many projects have established procedures for accepting patches – you'll need to check the specific rules for each project, because they will differ. Since there are several older, larger projects which accept patches via a developer mailing list, we'll go over an example of that now.

The workflow is similar to the previous use case – you create topic branches for each patch series you work on. The difference is how you submit them to the project. Instead of forking the project and pushing to your own writable version, you generate email versions of each commit series and email them to the developer mailing list:

```
$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit
```

Now you have two commits that you want to send to the mailing list. You use `git format-patch` to generate the mbox-formatted files that you can email to the list – it turns each commit into an email message with the first line of the commit message as the subject and the rest of the message plus the patch that the commit introduces as the body. The nice thing about this is that applying a patch from

an email generated with `format-patch` preserves all the commit information properly.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

The `format-patch` command prints out the names of the patch files it creates. The `-M` switch tells Git to look for renames. The files end up looking like this:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
lib/simplegit.rb |    2 ++
1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
end

def log(treeish = 'master')
-  command("git log #{treeish}")
+  command("git log -n 20 #{treeish}")
end

def ls_tree(treeish = 'master')
--
2.1.0
```

You can also edit these patch files to add more information for the email list that you don't want to show up in the commit message. If you add text between the `---` line and the beginning of the patch (the `diff --git` line), then developers can read it; but applying the patch excludes it.

To email this to a mailing list, you can either paste the file into your email program or send it via a command-line program. Pasting the text often causes formatting issues, especially with “smarter” clients that don’t preserve newlines and other whitespace appropriately. Luckily, Git provides a tool to help you send properly formatted patches via IMAP, which may be easier for you. We’ll demonstrate how to send a patch via Gmail, which happens to be the email agent we know best; you can read detailed instructions for a number of mail programs at the end of the aforementioned [Documentation/SubmittingPatches](#) file in the Git source code.

First, you need to set up the `imap` section in your `~/.gitconfig` file. You can set each value

separately with a series of `git config` commands, or you can add them manually, but in the end your config file should look something like this:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

If your IMAP server doesn't use SSL, the last two lines probably aren't necessary, and the host value will be `imap://` instead of `imaps://`. When that is set up, you can use `git imap-send` to place the patch series in the Drafts folder of the specified IMAP server:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

At this point, you should be able to go to your Drafts folder, change the To field to the mailing list you're sending the patch to, possibly CC the maintainer or person responsible for that section, and send it off.

You can also send the patches through an SMTP server. As before, you can set each value separately with a series of `git config` commands, or you can add them manually in the sendemail section in your `~/.gitconfig` file:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

After this is done, you can use `git send-email` to send your patches:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith
<jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Then, Git spits out a bunch of log information looking something like this for each patch you’re sending:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
      \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

總結

This section has covered a number of common workflows for dealing with several very different types of Git projects you’re likely to encounter, and introduced a couple of new tools to help you manage this process. Next, you’ll see how to work the other side of the coin: maintaining a Git project. You’ll learn how to be a benevolent dictator or integration manager.

維護一個專案

In addition to knowing how to effectively contribute to a project, you’ll likely need to know how to maintain one. This can consist of accepting and applying patches generated via `format-patch` and emailed to you, or integrating changes in remote branches for repositories you’ve added as remotes to your project. Whether you maintain a canonical repository or want to help by verifying or approving patches, you need to know how to accept work in a way that is clearest for other contributors and sustainable by you over the long run.

使用有特定主題的分支工作

When you’re thinking of integrating new work, it’s generally a good idea to try it out in a topic branch – a temporary branch specifically made to try out that new work. This way, it’s easy to tweak a patch individually and leave it if it’s not working until you have time to come back to it. If you create a simple branch name based on the theme of the work you’re going to try, such as `ruby_client` or something similarly descriptive, you can easily remember it if you have to abandon it for a while and come back later. The maintainer of the Git project tends to namespace these branches as well – such as `sc/ruby_client`, where `sc` is short for the person who contributed the work. As you’ll remember, you can create the branch based off your master branch like this:

```
$ git branch sc/ruby_client master
```

Or, if you want to also switch to it immediately, you can use the `checkout -b` option:

```
$ git checkout -b sc/ruby_client master
```

Now you’re ready to add your contributed work into this topic branch and determine if you want to merge it into your longer-term branches.

套用從電子郵件來的補丁

If you receive a patch over email that you need to integrate into your project, you need to apply the patch in your topic branch to evaluate it. There are two ways to apply an emailed patch: with `git apply` or with `git am`.

使用 `apply` 命令套用補丁

If you received the patch from someone who generated it with the `git diff` or a Unix `diff` command (which is not recommended; see the next section), you can apply it with the `git apply` command. Assuming you saved the patch at `/tmp/patch-ruby-client.patch`, you can apply the patch like this:

```
$ git apply /tmp/patch-ruby-client.patch
```

This modifies the files in your working directory. It’s almost identical to running a `patch -p1` command to apply the patch, although it’s more paranoid and accepts fewer fuzzy matches than `patch`. It also handles file adds, deletes, and renames if they’re described in the `git diff` format, which `patch` won’t do. Finally, `git apply` is an “apply all or abort all” model where either everything is applied or nothing is, whereas `patch` can partially apply patchfiles, leaving your working directory in a weird state. `git apply` is overall much more conservative than `patch`. It won’t create a commit for you – after running it, you must stage and commit the changes introduced manually.

You can also use `git apply` to see if a patch applies cleanly before you try actually applying it – you can run `git apply --check` with the patch:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

If there is no output, then the patch should apply cleanly. This command also exits with a non-zero status if the check fails, so you can use it in scripts if you want.

使用 `am` 命令套用補丁

If the contributor is a Git user and was good enough to use the `format-patch` command to generate their patch, then your job is easier because the patch contains author information and a commit message for you. If you can, encourage your contributors to use `format-patch` instead of `diff` to generate patches for you. You should only have to use `git apply` for legacy patches and things like that.

To apply a patch generated by `format-patch`, you use `git am`. Technically, `git am` is built to read an `mbox` file, which is a simple, plain-text format for storing one or more email messages in one text file. It

looks something like this:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function
```

```
Limit log functionality to the first 20
```

This is the beginning of the output of the `format-patch` command that you saw in the previous section. This is also a valid mbox email format. If someone has emailed you the patch properly using `git send-email`, and you download that into an mbox format, then you can point `git am` to that mbox file, and it will start applying all the patches it sees. If you run a mail client that can save several emails out in mbox format, you can save entire patch series into a file and then use `git am` to apply them one at a time.

However, if someone uploaded a patch file generated via `format-patch` to a ticketing system or something similar, you can save the file locally and then pass that file saved on your disk to `git am` to apply it:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

You can see that it applied cleanly and automatically created the new commit for you. The author information is taken from the email's `From` and `Date` headers, and the message of the commit is taken from the `Subject` and body (before the patch) of the email. For example, if this patch was applied from the mbox example above, the commit generated would look something like this:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author: Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit: Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

add limit to log function

Limit log functionality to the first 20
```

The `Commit` information indicates the person who applied the patch and the time it was applied. The `Author` information is the individual who originally created the patch and when it was originally created.

But it's possible that the patch won't apply cleanly. Perhaps your main branch has diverged too far from the branch the patch was built from, or the patch depends on another patch you haven't applied yet. In that case, the `git am` process will fail and ask you what you want to do:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation. You solve this issue much the same way – edit the file to resolve the conflict, stage the new file, and then run `git am --resolved` to continue to the next patch:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

If you want Git to try a bit more intelligently to resolve the conflict, you can pass a `-3` option to it, which makes Git attempt a three-way merge. This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository. If you do have that commit – if the patch was based on a public commit – then the `-3` option is generally much smarter about applying a conflicting patch:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In this case, this patch had already been applied. Without the `-3` option, it looks like a conflict.

If you're applying a number of patches from an mbox, you can also run the `am` command in interactive mode, which stops at each patch it finds and asks if you want to apply it:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

This is nice if you have a number of patches saved, because you can view the patch first if you don't remember what it is, or not apply the patch if you've already done so.

When all the patches for your topic are applied and committed into your branch, you can choose whether and how to integrate them into a longer-running branch.

切換到遠端分支

If your contribution came from a Git user who set up their own repository, pushed a number of changes into it, and then sent you the URL to the repository and the name of the remote branch the changes are in, you can add them as a remote and do merges locally.

For instance, if Jessica sends you an email saying that she has a great new feature in the `ruby-client` branch of her repository, you can test it by adding the remote and checking out that branch locally:

```
$ git remote add jessica git://github.com/jessica/myproject.git  
$ git fetch jessica  
$ git checkout -b rubyclient jessica/ruby-client
```

If she emails you again later with another branch containing another great feature, you can fetch and check out because you already have the remote setup.

This is most useful if you’re working with a person consistently. If someone only has a single patch to contribute once in a while, then accepting it over email may be less time consuming than requiring everyone to run their own server and having to continually add and remove remotes to get a few patches. You’re also unlikely to want to have hundreds of remotes, each for someone who contributes only a patch or two. However, scripts and hosted services may make this easier – it depends largely on how you develop and how your contributors develop.

The other advantage of this approach is that you get the history of the commits as well. Although you may have legitimate merge issues, you know where in your history their work is based; a proper three-way merge is the default rather than having to supply a `-3` and hope the patch was generated off a public commit to which you have access.

If you aren’t working with a person consistently but still want to pull from them in this way, you can provide the URL of the remote repository to the `git pull` command. This does a one-time pull and doesn’t save the URL as a remote reference:

```
$ git pull https://github.com/onetimeguy/project  
From https://github.com/onetimeguy/project  
 * branch HEAD      -> FETCH_HEAD  
Merge made by recursive.
```

決定要提到哪些資訊

Now you have a topic branch that contains contributed work. At this point, you can determine what you’d like to do with it. This section revisits a couple of commands so you can see how you can use them to review exactly what you’ll be introducing if you merge this into your main branch.

It’s often helpful to get a review of all the commits that are in this branch but that aren’t in your master branch. You can exclude commits in the master branch by adding the `--not` option before the branch name. This does the same thing as the `master..contrib` format that we used earlier. For

example, if your contributor sends you two patches and you create a branch called `contrib` and applied those patches there, you can run this:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

To see what changes each commit introduces, remember that you can pass the `-p` option to `git log` and it will append the diff introduced to each commit.

To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results. You may think to run this:

```
$ git diff master
```

This command gives you a diff, but it may be misleading. If your `master` branch has moved forward since you created the topic branch from it, then you'll get seemingly strange results. This happens because Git directly compares the snapshots of the last commit of the topic branch you're on and the snapshot of the last commit on the `master` branch. For example, if you've added a line in a file on the `master` branch, a direct comparison of the snapshots will look like the topic branch is going to remove that line.

If `master` is a direct ancestor of your topic branch, this isn't a problem; but if the two histories have diverged, the diff will look like you're adding all the new stuff in your topic branch and removing everything unique to the `master` branch.

What you really want to see are the changes added to the topic branch – the work you'll introduce if you merge this branch with master. You do that by having Git compare the last commit on your topic branch with the first common ancestor it has with the master branch.

Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

However, that isn't convenient, so Git provides another shorthand for doing the same thing: the triple-dot syntax. In the context of the `diff` command, you can put three periods after another branch

to do a **diff** between the last commit of the branch you’re on and its common ancestor with another branch:

```
$ git diff master...contrib
```

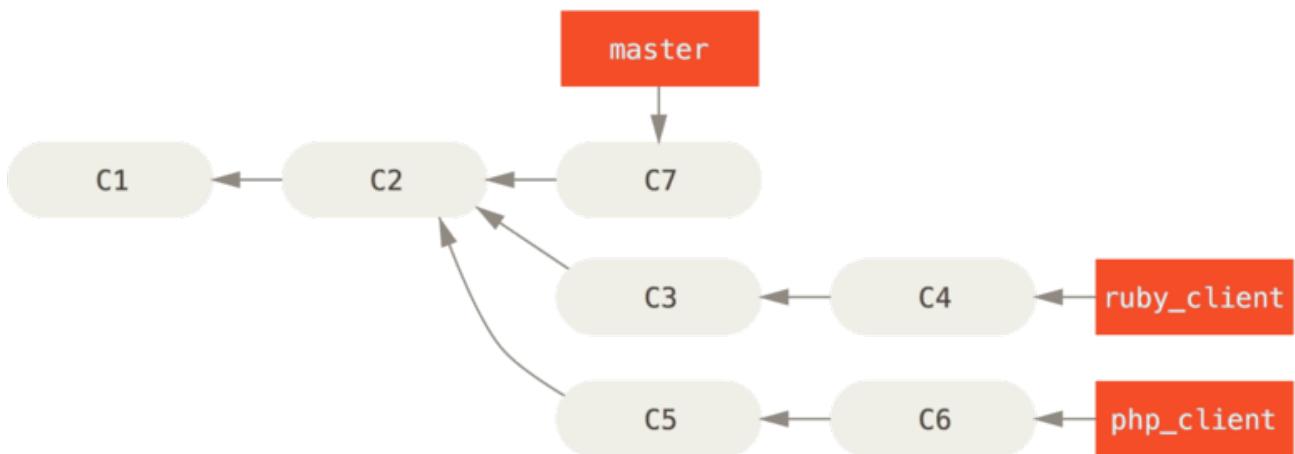
This command shows you only the work your current topic branch has introduced since its common ancestor with master. That is a very useful syntax to remember.

整合貢獻工作

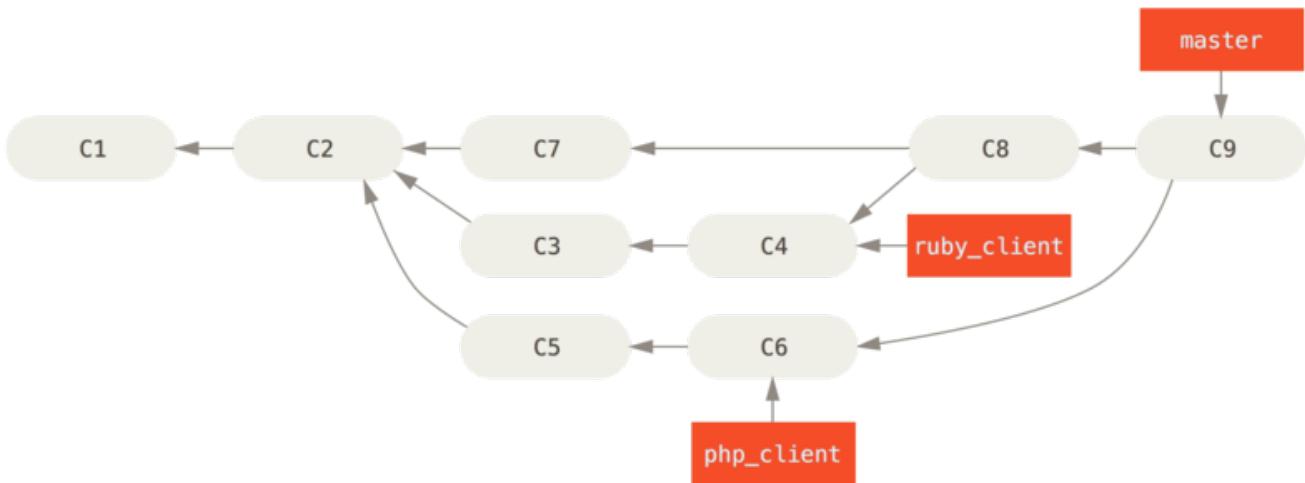
When all the work in your topic branch is ready to be integrated into a more mainline branch, the question is how to do it. Furthermore, what overall workflow do you want to use to maintain your project? You have a number of choices, so we’ll cover a few of them.

合併工作流程

One simple workflow merges your work into your **master** branch. In this scenario, you have a **master** branch that contains basically stable code. When you have work in a topic branch that you’ve done or that someone has contributed and you’ve verified, you merge it into your master branch, delete the topic branch, and then continue the process. If we have a repository with work in two branches named **ruby_client** and **php_client** that looks like [History with several topic branches](#), and merge **ruby_client** first and then **php_client** next, then your history will end up looking like [After a topic branch merge..](#)



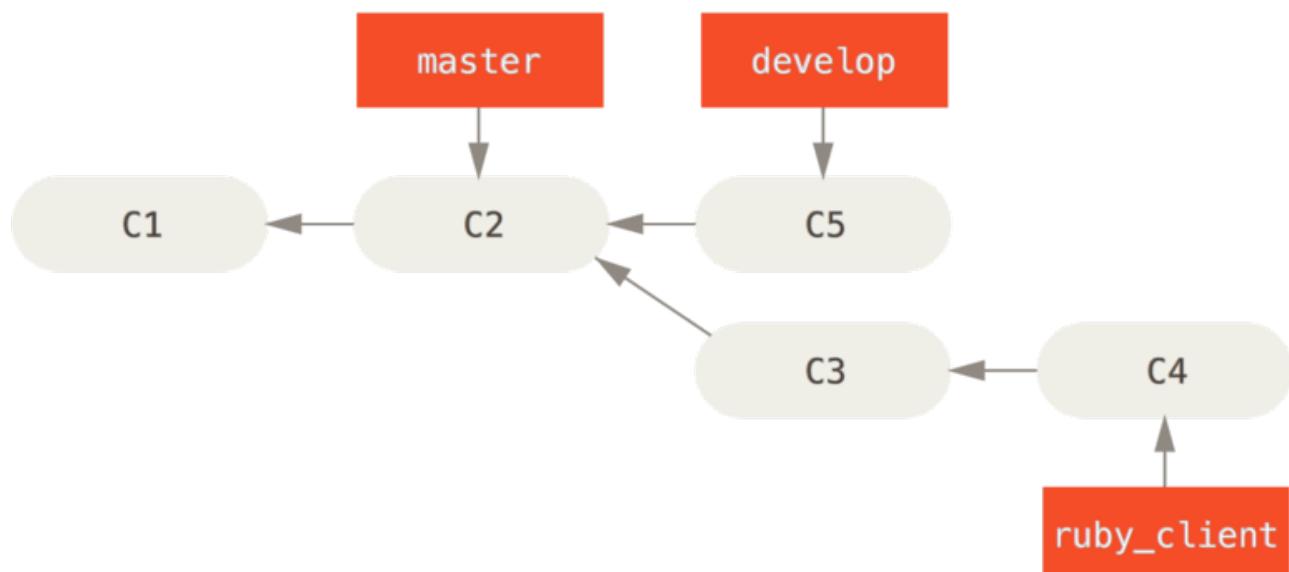
圖表 73. *History with several topic branches.*



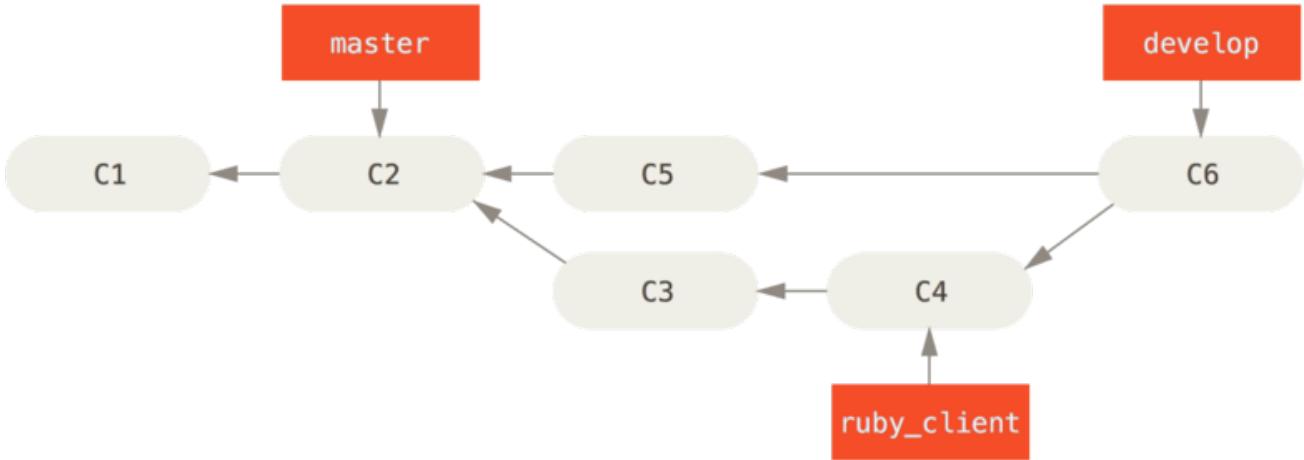
圖表 74. After a topic branch merge.

That is probably the simplest workflow, but it can possibly be problematic if you’re dealing with larger or more stable projects where you want to be really careful about what you introduce.

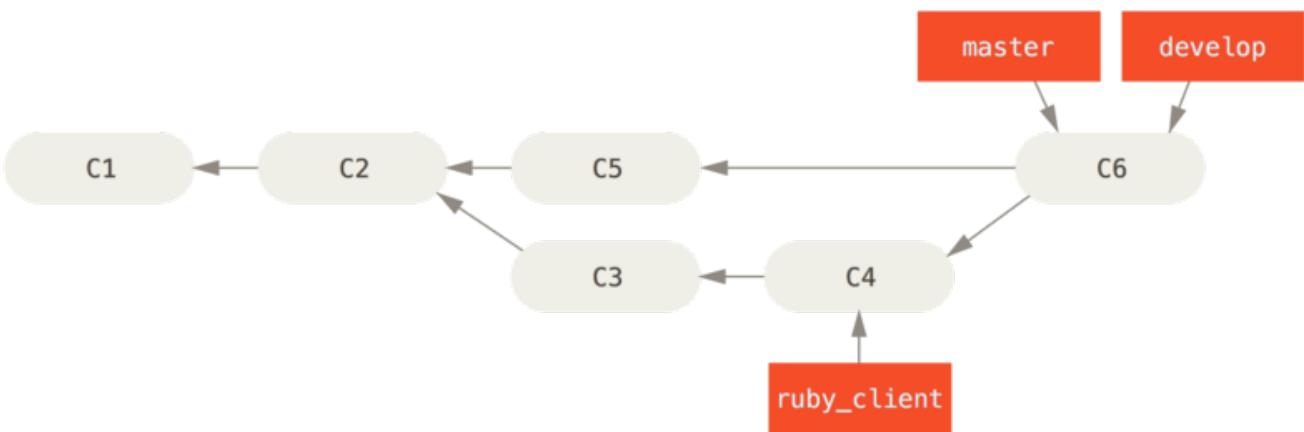
If you have a more important project, you might want to use a two-phase merge cycle. In this scenario, you have two long-running branches, `master` and `develop`, in which you determine that `master` is updated only when a very stable release is cut and all new code is integrated into the `develop` branch. You regularly push both of these branches to the public repository. Each time you have a new topic branch to merge in ([Before a topic branch merge.](#)), you merge it into `develop` ([After a topic branch merge.](#)); then, when you tag a release, you fast-forward `master` to wherever the now-stable `develop` branch is ([After a project release.](#)).



圖表 75. Before a topic branch merge.



圖表 76. After a topic branch merge.

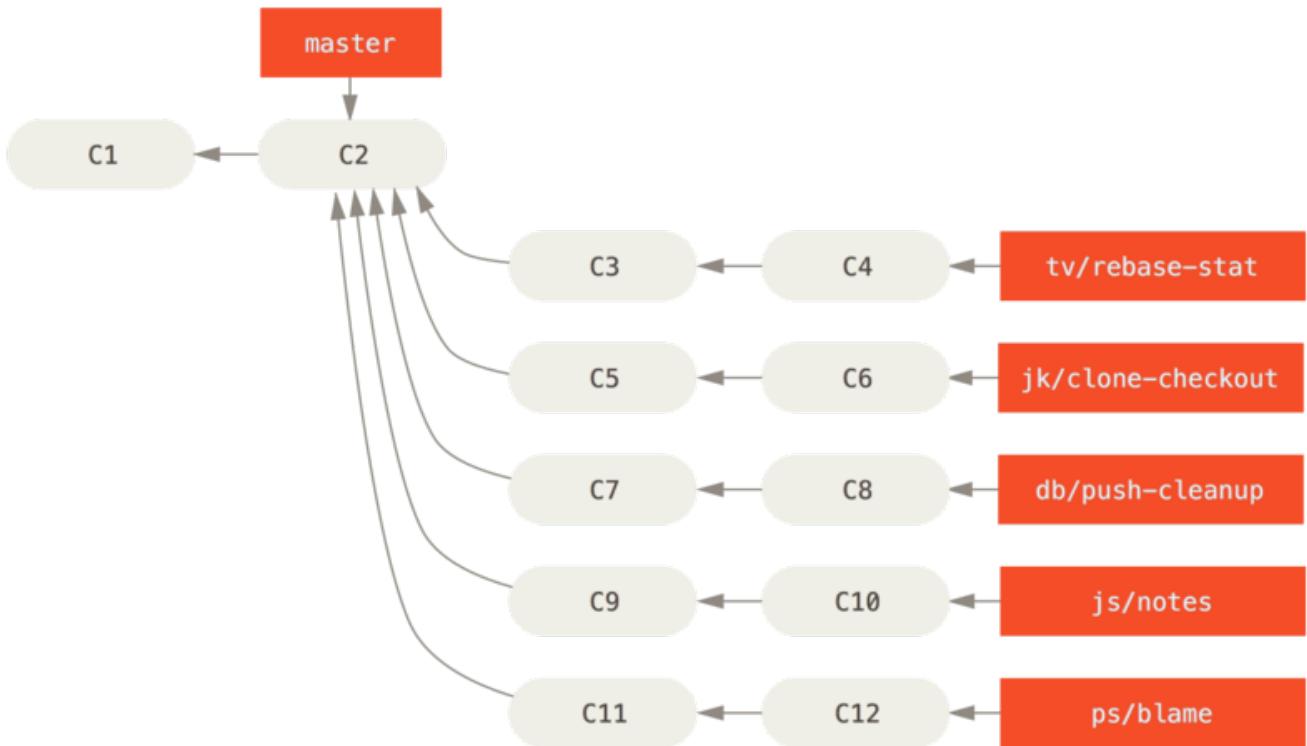


圖表 77. After a project release.

This way, when people clone your project's repository, they can either check out master to build the latest stable version and keep up to date on that easily, or they can check out develop, which is the more cutting-edge stuff. You can also continue this concept, having an integrate branch where all the work is merged together. Then, when the codebase on that branch is stable and passes tests, you merge it into a develop branch; and when that has proven itself stable for a while, you fast-forward your master branch.

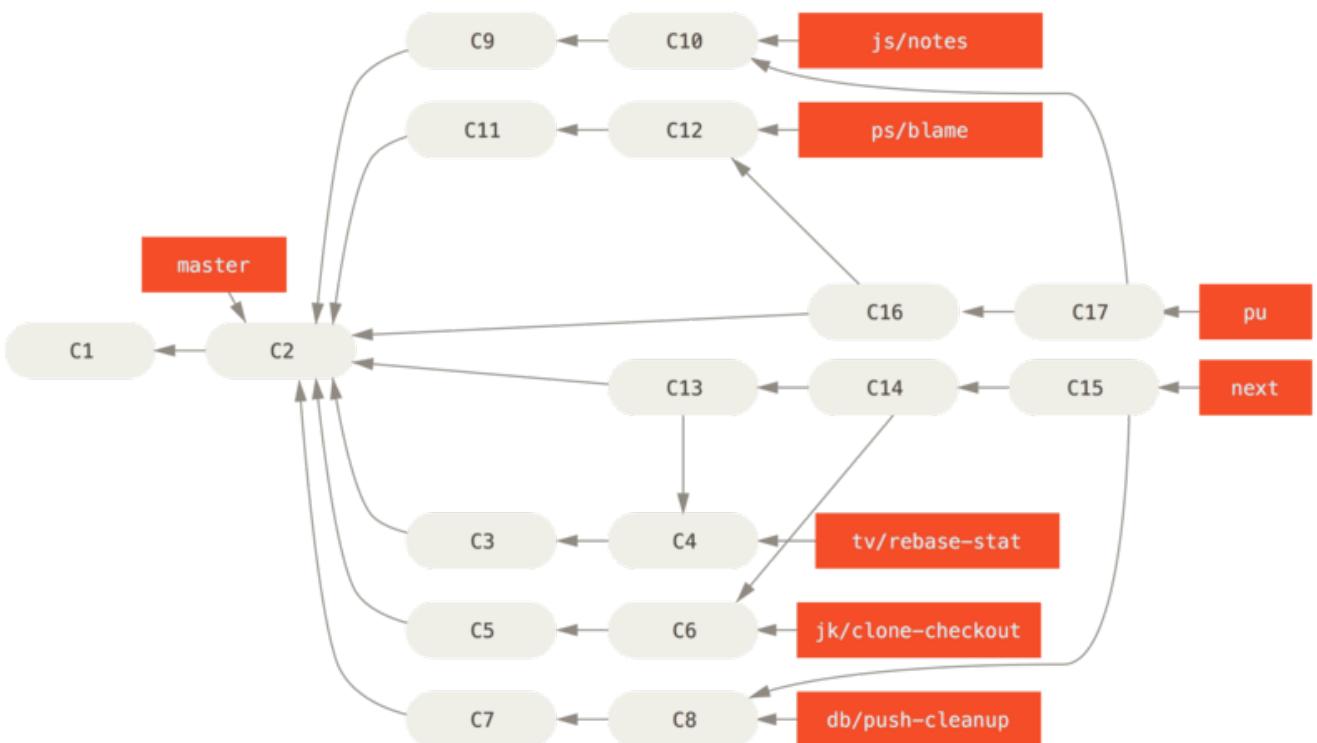
大量合併的工作六步

The Git project has four long-running branches: `master`, `next`, and `pu` (proposed updates) for new work, and `maint` for maintenance backports. When new work is introduced by contributors, it's collected into topic branches in the maintainer's repository in a manner similar to what we've described (see [Managing a complex series of parallel contributed topic branches](#)). At this point, the topics are evaluated to determine whether they're safe and ready for consumption or whether they need more work. If they're safe, they're merged into `next`, and that branch is pushed up so everyone can try the topics integrated together.



圖表 78. Managing a complex series of parallel contributed topic branches.

If the topics still need work, they’re merged into `pu` instead. When it’s determined that they’re totally stable, the topics are re-merged into `master` and are then rebuilt from the topics that were in `next` but didn’t yet graduate to `master`. This means `master` almost always moves forward, `next` is rebased occasionally, and `pu` is rebased even more often:



圖表 79. Merging contributed topic branches into long-term integration branches.

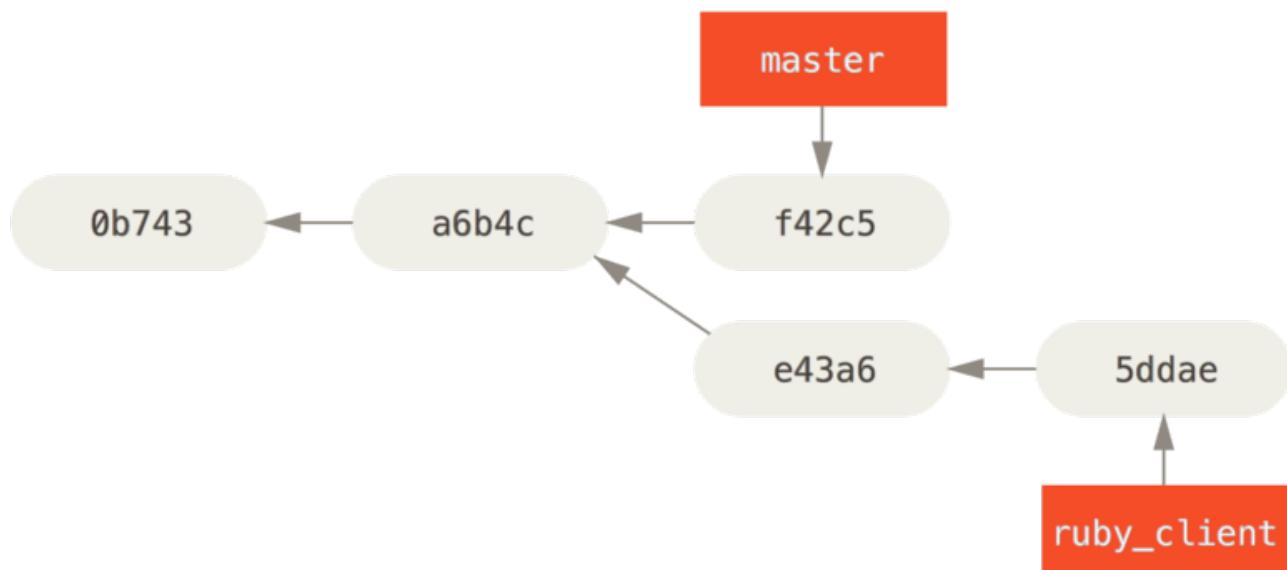
When a topic branch has finally been merged into `master`, it’s removed from the repository. The Git project also has a `maint` branch that is forked off from the last release to provide backported patches

in case a maintenance release is required. Thus, when you clone the Git repository, you have four branches that you can check out to evaluate the project in different stages of development, depending on how cutting edge you want to be or how you want to contribute; and the maintainer has a structured workflow to help them vet new contributions.

衍合與挑揀的工作流程

Other maintainers prefer to rebase or cherry-pick contributed work on top of their master branch, rather than merging it in, to keep a mostly linear history. When you have work in a topic branch and have determined that you want to integrate it, you move to that branch and run the rebase command to rebuild the changes on top of your current master (or `develop`, and so on) branch. If that works well, you can fast-forward your `master` branch, and you'll end up with a linear project history.

The other way to move introduced work from one branch to another is to cherry-pick it. A cherry-pick in Git is like a rebase for a single commit. It takes the patch that was introduced in a commit and tries to reapply it on the branch you're currently on. This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you'd prefer to cherry-pick it rather than run rebase. For example, suppose you have a project that looks like this:

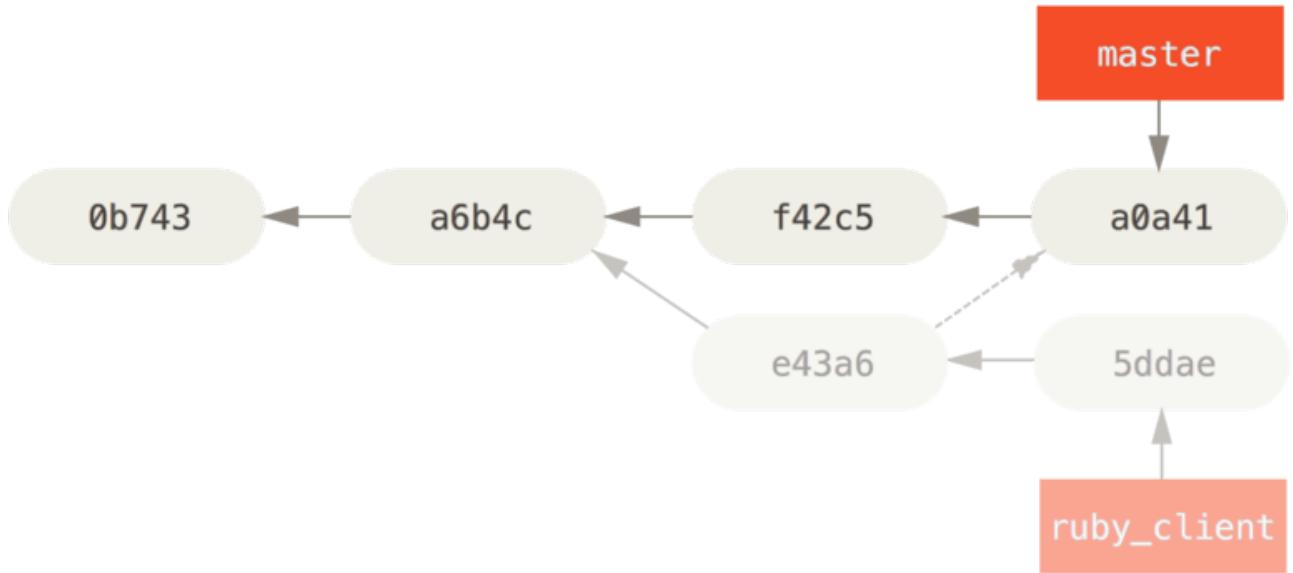


圖表 80. Example history before a cherry-pick.

If you want to pull commit `e43a6` into your master branch, you can run

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index
fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

This pulls the same change introduced in `e43a6`, but you get a new commit SHA-1 value, because the date applied is different. Now your history looks like this:



圖表 81. History after cherry-picking a commit on a topic branch.

Now you can remove your topic branch and drop the commits you didn't want to pull in.

Rerere

If you're doing lots of merging and rebasing, or you're maintaining a long-lived topic branch, Git has a feature called "rerere" that can help.

Rerere stands for "reuse recorded resolution" – it's a way of shortcircuiting manual conflict resolution. When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there's a conflict that looks exactly like one you've already fixed, it'll just use the fix from last time, without bothering you with it.

This feature comes in two parts: a configuration setting and a command. The configuration setting is `rerere.enabled`, and it's handy enough to put in your global config:

```
$ git config --global rerere.enabled true
```

Now, whenever you do a merge that resolves conflicts, the resolution will be recorded in the cache in case you need it in the future.

If you need to, you can interact with the rerere cache using the `git rerere` command. When it's invoked alone, Git checks its database of resolutions and tries to find a match with any current merge conflicts and resolve them (although this is done automatically if `rerere.enabled` is set to `true`). There are also subcommands to see what will be recorded, to erase specific resolution from the cache, and to clear the entire cache. We will cover rerere in more detail in [Rerere](#).

為釋出的版本加上標籤

When you've decided to cut a release, you'll probably want to drop a tag so you can re-create that release at any point going forward. You can create a new tag as discussed in [Git 基礎](#). If you decide to sign the tag as the maintainer, the tagging may look something like this:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon <schacon@gmail.com>"  
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you do sign your tags, you may have the problem of distributing the public PGP key used to sign your tags. The maintainer of the Git project has solved this issue by including their public key as a blob in the repository and then adding a tag that points directly to that content. To do this, you can figure out which key you want by running `gpg --list-keys`:

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg  
-----  
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]  
uid Scott Chacon <schacon@gmail.com>  
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Then, you can directly import the key into the Git database by exporting it and piping that through `git hash-object`, which writes a new blob with those contents into Git and gives you back the SHA-1 of the blob:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin  
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Now that you have the contents of your key in Git, you can create a tag that points directly to it by specifying the new SHA-1 value that the `hash-object` command gave you:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

If you run `git push --tags`, the `maintainer-pgp-pub` tag will be shared with everyone. If anyone wants to verify a tag, they can directly import your PGP key by pulling the blob directly out of the database and importing it into GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

They can use that key to verify all your signed tags. Also, if you include instructions in the tag message, running `git show <tag>` will let you give the end user more specific instructions about tag verification.

產生一個建置編號

Because Git doesn't have monotonically increasing numbers like `v123` or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run `git describe` on that commit. Git gives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit you're describing:

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

This way, you can export a snapshot or build and name it something understandable to people. In fact, if you build Git from source code cloned from the Git repository, `git --version` gives you something that looks like this. If you’re describing a commit that you have directly tagged, it gives you the tag name.

The `git describe` command favors annotated tags (tags created with the `-a` or `-s` flag), so release tags should be created this way if you’re using `git describe`, to ensure the commit is named properly when described. You can also use this string as the target of a checkout or show command, although it relies on the abbreviated SHA-1 value at the end, so it may not be valid forever. For instance, the Linux kernel recently jumped from 8 to 10 characters to ensure SHA-1 object uniqueness, so older `git describe` output names were invalidated.

準備釋出一個版本

Now you want to release a build. One of the things you’ll want to do is create an archive of the latest snapshot of your code for those poor souls who don’t use Git. The command to do this is `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe  
master`.tar.gz  
$ ls *.tar.gz  
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

If someone opens that tarball, they get the latest snapshot of your project under a project directory. You can also create a zip archive in much the same way, but by passing the `--format=zip` option to `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe  
master`.zip
```

You now have a nice tarball and a zip archive of your project release that you can upload to your website or email to people.

簡短的日誌

It’s time to email your mailing list of people who want to know what’s happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or email is to use the `git shortlog` command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray `puts`
    Make ls_tree ignore nils

Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gemspec for version 1.0.2
```

You get a clean summary of all the commits since v1.0.1, grouped by author, that you can email to your list.

Summary

You should feel fairly comfortable contributing to a project in Git as well as maintaining your own project or integrating other users' contributions. Congratulations on being an effective Git developer! In the next chapter, you'll learn about how to use the largest and most popular Git hosting service, GitHub.

GitHub

GitHub 是個上面有大量 Git 倉儲的主機，同時也是數以百萬計的開發者和專案的交流中心。有非常高比例的 Git 倉儲是被託管在 GitHub 上的。同時也有很多的開源專案使用 GitHub 來做 Git 託管、議題追蹤、程式碼審閱，還有其他各種用途。雖然這並非 Git 開源計畫的直接目的，但在你專業的使用 Git 時，你還是有很大的機會會想要或需要使用 GitHub。

這章節是關於如何有效率的使用 GitHub。這會包含帳號的申請和管理、Git 倉儲的建立及使用、參與別人的專案以及管理自己的專案的一般流程、GitHub 的應用程式介面，以及各式各樣能讓你更加便利的小技巧。

如果你對於如何在 GitHub 上託管自己的專案或是與其他人一起在 GitHub 上的專案上合作不感興趣的話，你可以放心地直接前進到下個章節 [Git Tools](#)。

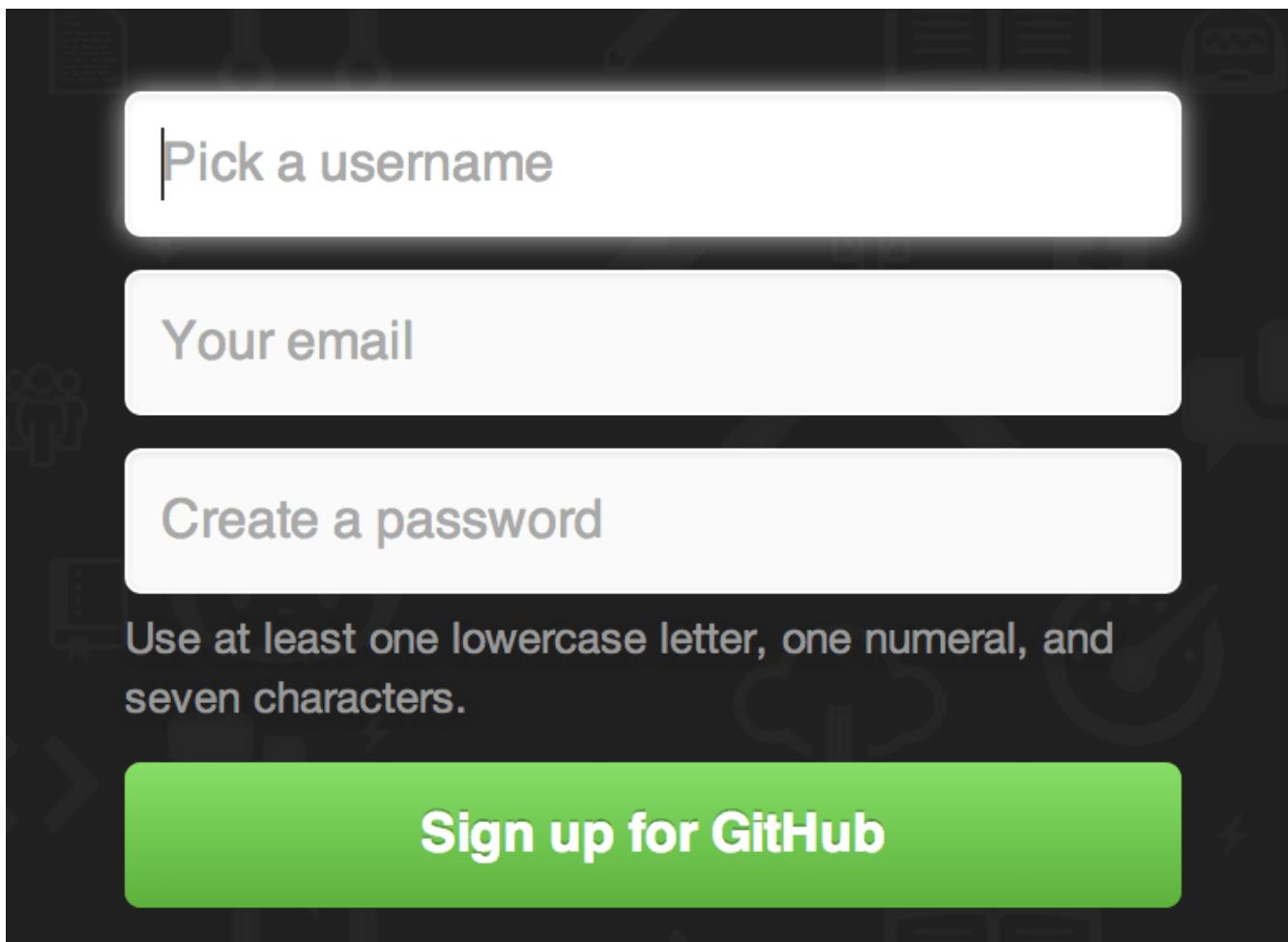
介面變更

警告

有件很重要的事情是：就像其他的活動中的網站一樣，截圖中的使用者介面會隨時間改變。希望這並不會影響到我們要做的事情，但如果你想要最新版的截圖，建議您去瀏覽線上版。

建立帳戶及設定

你第一件要做的事就是建立一個免費的使用者帳戶。只要去 <http://github.com>，選擇一個尚未被使用過的使用者名稱，輸入一個電郵地址以及一個密碼，然後按下那個大大的綠色「Sign up for GitHub」按鈕就可以了。



圖表 82. GitHub 的帳號申請畫面。

你會看到各種付費升級方案的頁面，但是在這裡我們只要選免費方案即可。之後 GitHub 會寄給你一封電子郵件來確認電郵地址。因為這很重要所快點去做（我們等一下會解釋）。

筆記 你可以用免費帳戶使用 GitHub 所有的功能，但是你所有的專案都只能完全公開（所有人都有讀取權限）。GitHub 的付費方案會提供數個私人專案的額度，但在本書中我們不會提及這個。

點擊畫面左上的 GitHub 圖示會連結到你的資訊主頁。你現在可以開始使用 GitHub 了。

SSH 存取

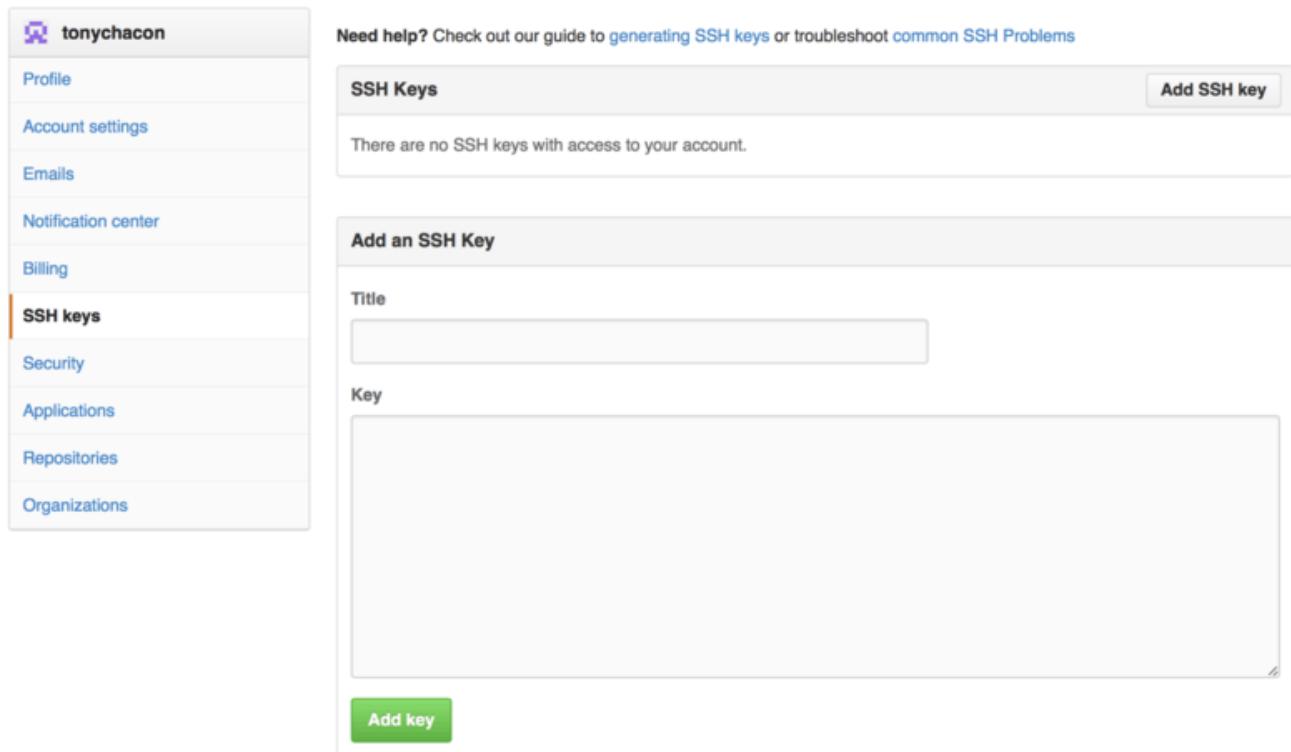
你現在就可以透過 <https://> 協定並使用你剛剛設定的帳號及密碼來認證以連接你的 GitHub 倉儲。但是，如果只是要克隆公開專案，其實你連註冊都不用 — 剛剛建立的帳號是在我們之後要 fork 專案或是推送變更到你複製的倉儲時才會用到。

如果想要使用 SSH 遠端，你必須要去設定一個公鑰。（如果你沒有公鑰的話請參考 [產生你的 SSH 公鑰](#)。）點擊視窗右上的連結開啟你的帳戶設定頁面：

“帳戶設定” 連結。

image::images/account-settings.png[``帳戶設定'' 連結。]

然後點選左側的「SSH keys」區塊。



圖表 83. “SSH keys” 連結。

點選 "Add an SSH key" 按鈕，之後替你的金鑰命名，複製你的 `~/.ssh/id_rsa.pub` (或是任意檔名) 公鑰檔的內容並貼在文字區塊內，然後點擊「Add key」。

建議替你的金鑰取個容易記憶的名字。你可以替你的每個金鑰取類似「我的筆電」或「工作帳號」的名字，這樣在之後要撤銷金鑰時，你可以很輕易地找出你要撤銷的那個金鑰。

你的頭像

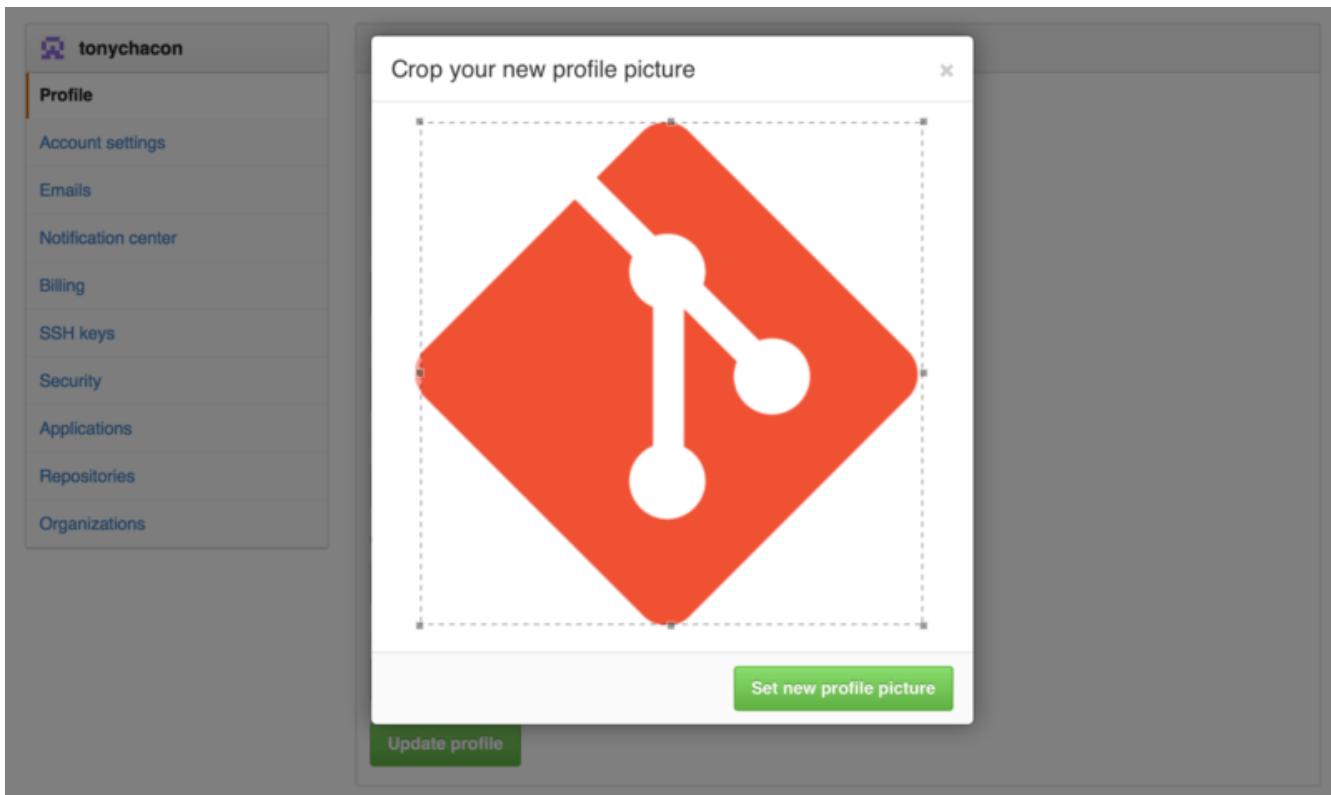
再來，你可以把那個自動生成的頭像換成你希望的那個圖像。首先，切換到「profile」分頁（在 SSH keys 的上面那一個），然後點選「Upload new picture」。

The screenshot shows the GitHub profile settings interface. On the left is a sidebar with a user icon and the name 'tonychacon'. Below this are several navigation links: Profile (which is highlighted in orange), Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications, Repositories, and Organizations.

The main content area is titled 'Public profile'. It contains fields for 'Profile picture' (with a placeholder image and a 'Upload new picture' button), 'Name' (empty input field), 'Email (will be public)' (empty input field), 'URL' (empty input field), 'Company' (empty input field), and 'Location' (empty input field). At the bottom is a green 'Update profile' button.

圖表 84. 「Profile」連結。

我們會拿個 Git 的圖示來用，而且我們會需要剪裁它。



圖表 85. 裁切你的頭像。

之後你在網站上的任何互動，大家都會看到你的使用者名稱旁有你的頭像。

如果你剛好有在那知名的 Gravatar 上傳過頭像的話（通常是 Wordpress 的帳戶會有用到），那個頭像會自動被當成預設的頭像，你就不需要執行這個步驟。

你的電子郵件地址

GitHub 是用電郵地址把 Git commits 和使用者關聯在一起的。如果你擁有多個電郵地址，而且你也想要 GitHub 把他們正確的和你關聯在一起，你需要去管理頁面的 Emails 區塊，加入所有你使用的電子郵件。

Your **primary GitHub email address** will be used for account-related notifications (e.g. account changes and billing receipts) as well as any web-based GitHub operations (e.g. edits and merges).

tonychacon@example.com	Primary	Public	
tchacon@example.com			
tony.chacon@example.com	Unverified		

Add email address

Keep my email address private
We will use **tonychacon@users.noreply.github.com** when performing Git operations and sending email on your behalf.

圖表 86. 加入電郵地址

在 [加入電郵地址](#) 這張圖裡面，我們可以看到各種狀態。第一個地址是已認證而且設定為主要的電郵地址，意味著所有的通知和其他信件都會寄到這個地址。第二個地址是已認證，所以可以隨時被切換成主要的地址。最後一個是未認證的，意味著那並不能作為你的主要地址使用。如果 GitHub 在站上的任意一個 Git 倉

儲的 commit 訊息裡發現有對應的電郵地址，它就會被連結到你的使用者。

二階段驗證

最後，為了得到額外的安全保障，你應該要設定二階段驗證（或稱作「2FA」）。二階段驗證是個越來越熱門的驗證技術，能夠在你的密碼被盜取的時候仍能保障你的帳戶的安全。開啟這個功能後，GitHub 會要求你提供兩個不同的方式來驗證。所以在其中一個失效時，攻擊者仍然不能存取你的帳號。

你可以在你的帳戶設定裡面的安全性分頁找到兩階段驗證的設定。

The screenshot shows the GitHub 'Security' settings page for the user 'tonychacon'. On the left sidebar, 'Security' is highlighted. The main content area is titled 'Two-factor authentication'. It shows the status as 'Off' with a red 'X'. A button labeled 'Set up two-factor authentication' is visible. Below this, a note explains that two-factor authentication provides another layer of security to your account, with a link to 'GitHub Help'. The 'Sessions' section lists a single session: 'Paris 85.168.227.34' (Your current session), which is a Safari browser on OS X 10.9.4, located in Paris, Ile-de-France, France, signed in on September 30, 2014.

圖表 87. 安全性專頁內的兩階段驗證

在點擊「Set up two-factor authentication」按鈕之後，他會連結到設定頁面，讓你可以設定手機 App 以產生第二階段驗證碼（一個基於時間產生的一次性密碼），或者你也可以讓 GitHub 在你每次要登入時透過簡訊發送一個驗證碼給你。

當你選擇並完成 2FA 的設定之後，你的帳戶會變得更安全。並在之後的每次登入，除了密碼你還要提供一個驗證碼才能登入 GitHub。

參與一個專案

現在帳號設定好了，來看看一些關於如何對現有專案做出貢獻的有用小細節吧。

Fork 專案

如果你想要參與一個你沒有推送權限的專案，你可以「fork」一份。這代表說 GitHub 會複製一份這個專案的副本給你，並且你對這副本有全部的權限。這副本會存在於你的帳號下，你可以對它進行推送。

筆記

歷史上，「fork」這件事情在程式開發的領域裡多少帶了點負面意味。因為有些人會透過這途徑將一個開源專案的發展帶往不同方向，甚至是創造出跟原本專案競爭的作品，進而導致貢獻者的分裂。在 GitHub 上，「fork」就是把一份相同的專案放在你的帳號之下，讓你能夠公開對這專案做變更，做為一個以更開放的方式來參與專案。

透過這方式，專案就不用去煩惱需要把所有協作者加入使用者來讓他們擁有推送的權限。所有人可以 fork 專案，對 fork 出來的專案推送變更，然後去發出我們等下會提到的 Pull Request，來把這些變更貢獻回原本的專案裡。這會開立一個能夠作程式碼審閱的討論串，然後擁有者能和貢獻者討論這個變更，直到擁有者覺得可以合併進原始專案裡面。

去到專案頁面，點下右上角的「Fork」鍵，就可以 fork 專案。



圖表 88. 「Fork」鈕

幾秒鐘之後，你就會被帶到你有寫入權限的新專案頁面。

GitHub 流程

GitHub 是基於一個以 Pull Request 為中心的特別合作流程而設計出來的。這個流程，不論是你在一個緊密連結的團隊裡共同在單一倉儲上合作；或是一個由散布全球的陌生人們構成的合作網路或是公司，透過大量的 fork 專案來對專案做出貢獻，都能運作。這一切都是基於我們在 [使用 Git 分支](#) 這章所講過的 [主題分支](#) 的工作流程。

一般情況下就是照著下面的程序運作的：

1. 從 `master` 建立一個主題分支。
2. 加入一些變更來改善這個專案。
3. 把這個分支推送到你的 GitHub 專案。
4. 在 GitHub 上建立一個 Pull Request。
5. 討論，並在需要的時候加入新的變更。
6. 專案擁有者視情況決定要把這個 Pull Request 合併進原始專案，或是關閉它。

這基本上就是我們在 [整合式管理員工作流程](#) 這部分提過的整合式管理流程，不過我們是使用 GitHub 的網頁工具來做溝通或是變更審閱，而非電子郵件。

我們來看看下面的例子來了解如何使用這個流程來對 GitHub 上的專案做出變更吧。

建立一個 Pull Request

Tony 在找能夠在他的 Arduino 可程式化微控制器上運作的程式碼。然後他在 GitHub 的這個專案 <https://github.com/schacon/blink> 找到了個很棒的程式碼。

The screenshot shows a GitHub repository page for 'schacon / blink'. At the top, there are buttons for 'Watch' (0), 'Star' (0), and 'Fork' (0). Below that, a dropdown says 'branch: master'. The main title is 'blink / blink.ino'. A profile picture of schacon is shown with the text 'schacon on Jun 12 my arduino blinking code (from arduino.cc)'. It says '1 contributor'. The code listing shows 25 lines (20 sloc) of 0.71 kb. The code itself is:

```
1  /*
2   * Blink
3   * Turns on an LED on for one second, then off for one second, repeatedly.
4   *
5   * This example code is in the public domain.
6   */
7
8 // Pin 13 has an LED connected on most Arduino boards.
9 // give it a name:
10 int led = 13;
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14     // initialize the digital pin as an output.
15     pinMode(led, OUTPUT);
16 }
17
18 // the Loop routine runs over and over again forever:
19 void loop() {
20     digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
21     delay(1000);              // wait for a second
22     digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23     delay(1000);              // wait for a second
24 }
```

圖表 89. 他想要做出貢獻的專案

唯一的小問題就是閃爍的頻率太高了，我們覺得放慢成 3 秒一次會比原本的 1 秒一次好。所以我們來改善這個程式並作為變更要求來提交回去吧。

首先，我們要先按下稍早提過的「Fork」鈕來取得這個專案的副本。我們在這邊使用的使用者名稱是「tonychacon」，所以這個專案的副本會放在 <https://github.com/tonychacon/blink>，而且我們能編輯這個副本。我們把它克隆一份到電腦上、建立主題分支、對程式碼作變更，最後推送回 GitHub。

```

$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH);      // turn the LED on (HIGH is the voltage
level)
    [-delay(1000);-]{+delay(3000);+}          // wait for a second
    digitalWrite(led, LOW);       // turn the LED off by making the voltage
LOW
    [-delay(1000);-]{+delay(3000);+}          // wait for a second
}

$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink

```

① 把我們 fork 的專案克隆一份到本機

② 建立名稱有意義的主題分支

③ 對程式碼作變更

④ 確認這個變更一切 OK

⑤ 把變更加入我們的主題分支

⑥ 把我們的新分支推送回 GitHub 的 fork 上

如果我們回到我們在 GitHub 上的 fork，我們可以看到 GitHub 發現我們推了新分支上來，並且顯示了一個大大的綠色按鈕讓我們可以檢視我們的變更，並能對原始專案開啟一個 Pull Request。

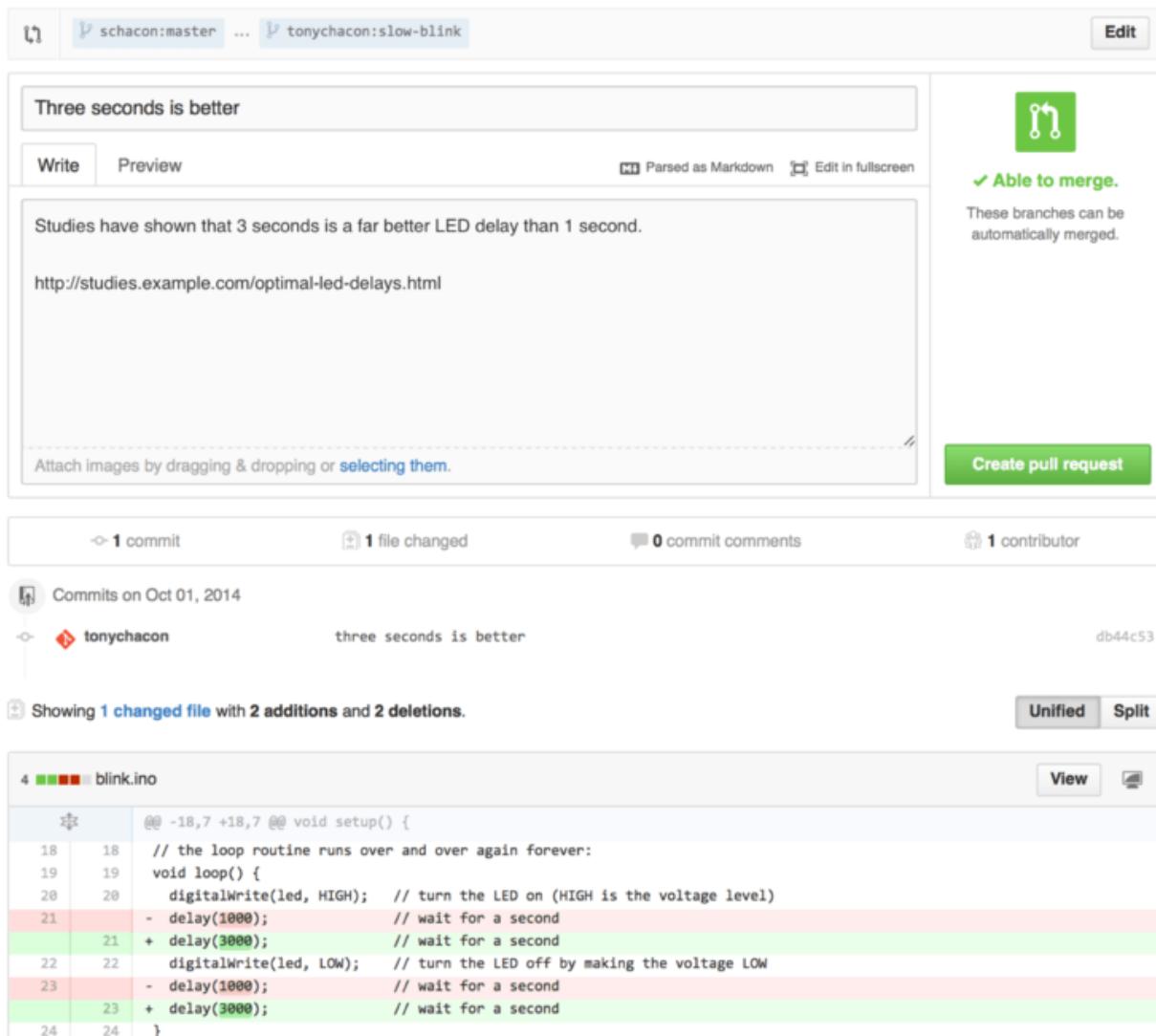
你也可以去到在 <https://github.com/<user>/<project>/branches> 上的「Branches」頁面去找出你的分支並從那邊開啟一個新的 Pull Request。

The screenshot shows a GitHub repository page for 'tonychacon / blink'. At the top right, there are buttons for 'Unwatch' (1), 'Star' (0), 'Fork' (1). Below the header, it says 'Example file to blink the LED on an Arduino — Edit'. A navigation bar shows '2 commits', '2 branches', '0 releases', and '1 contributor'. A sidebar on the right lists 'Code', 'Pull Requests' (0), 'Wiki', 'Pulse', 'Graphs', and 'Settings'. It also shows an 'HTTPS clone URL' field with the value 'https://github.com/tonychacon/blink'. Buttons for 'Clone in Desktop' and 'Download ZIP' are at the bottom of the sidebar. The main content area shows a branch named 'slow-blink' (less than a minute ago) and a green 'Compare & pull request' button. Below this, a list of files includes 'README.md', 'Create README.md', 'blink.ino', and 'README.md'. The 'README.md' file is expanded, showing its content: 'Blink'. A note below it says: 'This repository has an example file to blink the LED on an Arduino board.'

圖表 90. Pull Request 按鈕

如果我們按下綠色按鈕，我們會看到一個畫面，要求我們針對這次的 Pull Request 編寫「標題」和「描述」。花費一些心思在這上面總是值得的，因為好的說明可以幫助原專案的擁有者去確認「你所嘗試的事情」、「你提案的變更內容是否正確」以及「接受這些變更是否能改善到原專案」。

同時我們也會看到在主分支沒有的所有提交列表（在這個範例中只有一個提交），和一個彙整所有修改的差異資訊，以便讓其他人知道當原作者合併後會有哪些差異。



The screenshot shows a GitHub pull request page. At the top, it says "Three seconds is better". Below that are tabs for "Write" and "Preview". The "Preview" tab is selected, showing the content: "Studies have shown that 3 seconds is a far better LED delay than 1 second." followed by a link "http://studies.example.com/optimal-led-delays.html". To the right, there's a green icon with a gear and wrench, and the text "Able to merge. These branches can be automatically merged." A "Create pull request" button is also visible.

Below the preview, there are stats: "1 commit", "1 file changed", "0 commit comments", and "1 contributor". The commit details show "Commits on Oct 01, 2014" by "tonychacon" with the commit message "three seconds is better" and hash "db44c53".

At the bottom, it says "Showing 1 changed file with 2 additions and 2 deletions." with "Unified" and "Split" options. The diff view shows changes to "blink.ino":

```

4 @@ -18,7 +18,7 @@
 18 // the loop routine runs over and over again forever:
 19 void loop() {
 20     digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
 21     - delay(1000);           // wait for a second
 22     + delay(3000);           // wait for a second
 23     digitalWrite(led, LOW);   // turn the LED off by making the voltage LOW
 24     - delay(1000);           // wait for a second
 25     + delay(3000);           // wait for a second
}

```

圖表 91. Pull Request 建立頁面

當你按下畫面中的「Create pull request」按鈕時，你 fork 的來源專案的擁有者會收到一個通知，通知他有人建議一個變動並且會附上連往包含所有資訊的頁面連結。

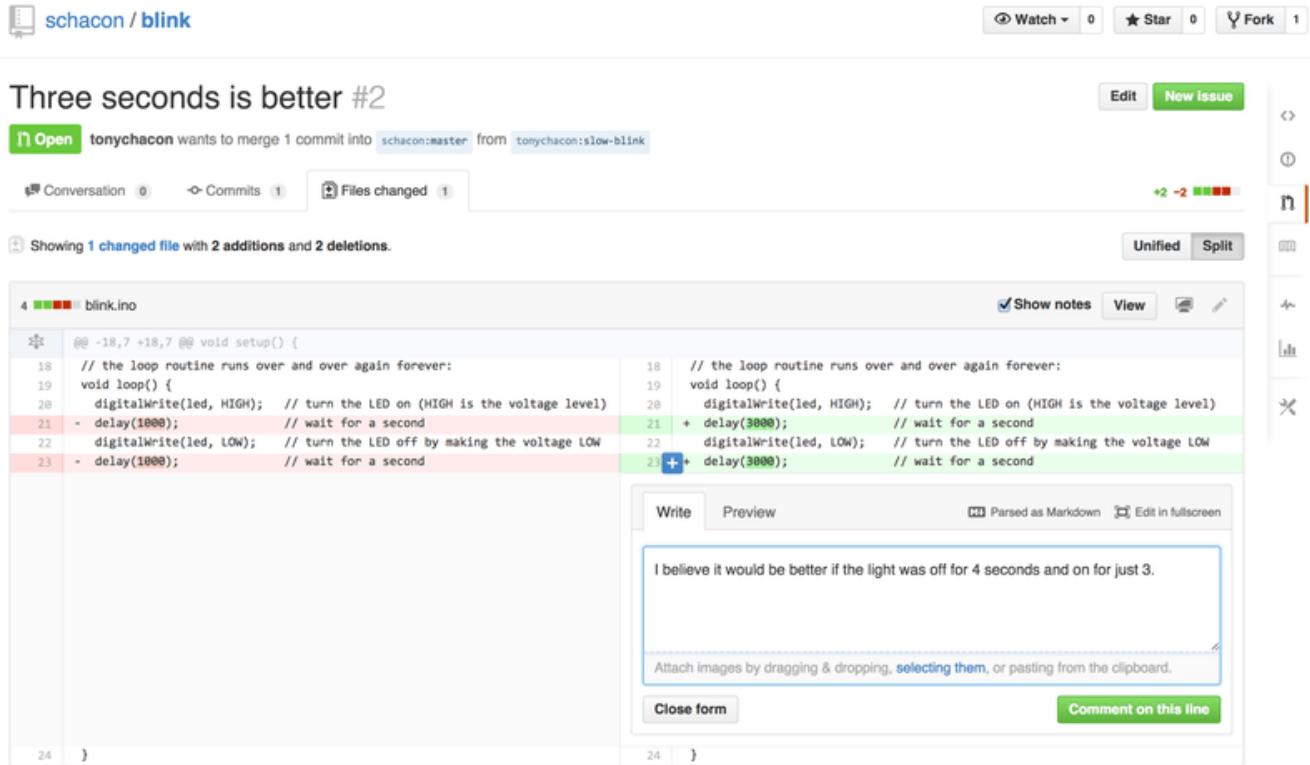
筆記

雖然在這種公開專案上，Pull Requests 通常都是在貢獻者已經準備好要加入的變更時才會發出；但是它常用於專案「剛開始」時的一些內部專案。基於 Pull Request 在建立「之後」仍然可以持續加入新的變更的特性，因此也常會在初期建立當作一個團隊合作的環境，而非在最後才使用。

重複使用一個 Pull Request

現在呢，專案的擁有者可以閱覽所有建議的變更，然後決定要合併進來、拒絕變更或是對這留下評論。在這邊我們當作他覺得他喜歡這點子好了，但是他覺得燈暗掉的時間要比亮的時間長一點。

這個互動或許會透過電子郵件並依照在 [分散式的 Git](#) 提到的工作流程運作；而在 GitHub 上，這是在線上運作的。專案擁有者可以在審閱差異總表時，點一下想要評論的那行內容並留下評論。



圖表 92. 對 Pull Request 的某行程式碼下評論

當維護者留下評論，建立這個 Pull Request 的人（以及所有關注這個倉儲的人）都會收到通知。我們等等會對這做自訂，不過如果有開啟電子郵件通知，Tony 會這樣的一封信：



圖表 93. 以電子郵件型式寄送的評論

其他人也可以對 Pull Request 留下一般評論。在 [Pull Request 討論頁](#) 裏面我們可以看到專案擁有者對某行程式碼做評論，同時也在討論區塊留了一般評論。你可以看到程式碼評論也會被帶到這個互動之中。

Three seconds is better #2

[Edit](#)[New issue](#)

[Open](#) tonychacon wants to merge 1 commit into `schacon:master` from `tonychacon:slow-blink`

Conversation 1 Commits 1 Files changed 1 +2 -2

tonychacon commented 6 minutes ago
Studies have shown that 3 seconds is a far better LED delay than 1 second.
<http://studies.example.com/optimal-led-delays.html>

three seconds is better db44c53

schacon commented on the diff just now

`blink.ino` View full changes

```
22 22 digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
23 - delay(1000); // wait for a second
23 + delay(3000); // wait for a second
```

schacon added a note just now
I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note

schacon commented just now
If you make that change, I'll be happy to merge this.

Labels None yet

Milestone No milestone

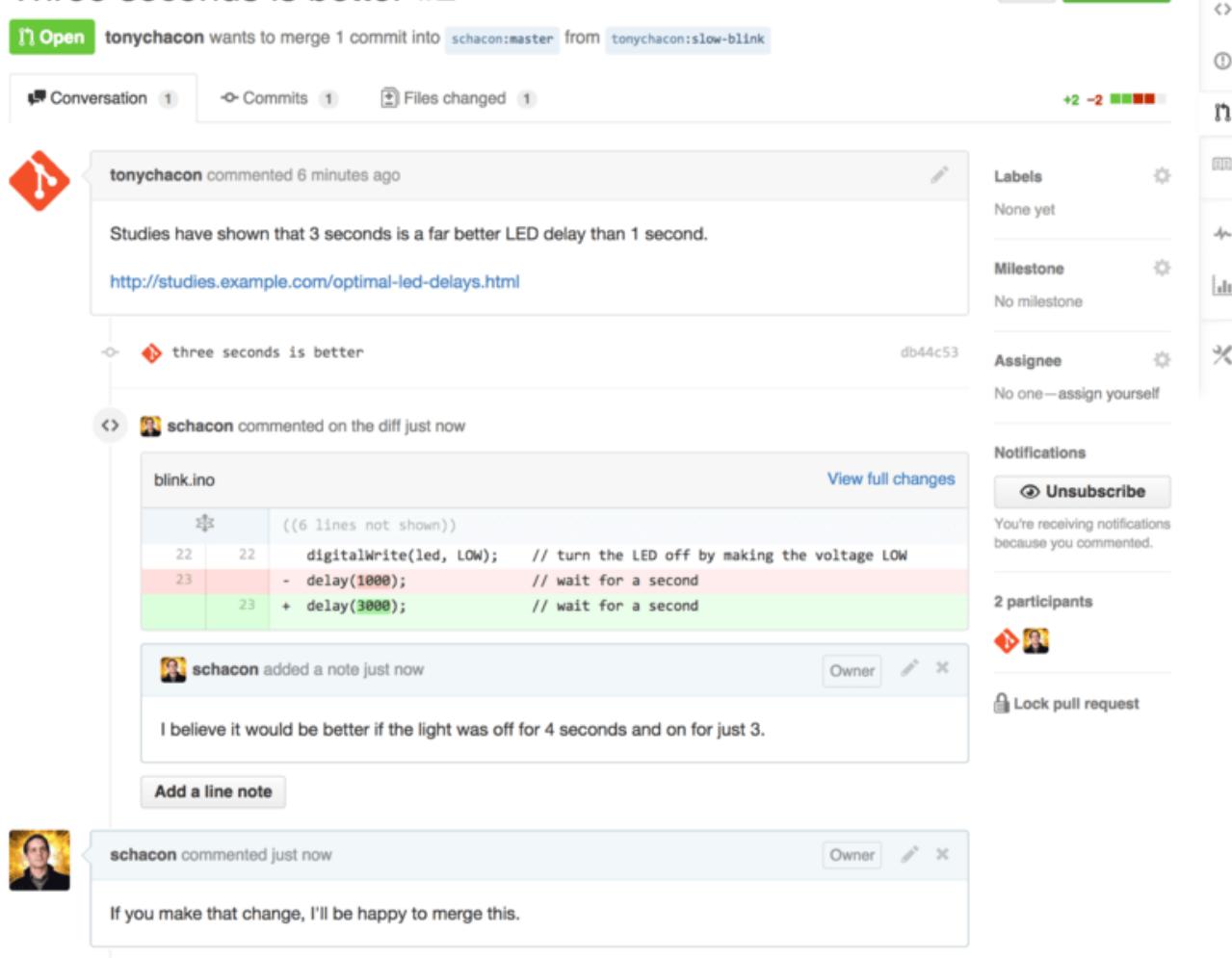
Assignee No one—assign yourself

Notifications

Unsubscribe You're receiving notifications because you commented.

2 participants

Lock pull request



圖表 94. Pull Request 討論頁

現在貢獻者就可以知道他要做哪些處理才能讓擁有者接受這個變更。幸好這事也很直觀。如果是透過電子郵件的話你需要把所有的變動重新執行一次然後重新上傳，但是在 GitHub 上你只要對主題分支再次做提交然後推送上去，即可更新該 Pull Request。在 [結束 Pull Request](#) 中，你也可以看到 Pull Request 中舊的程式碼上的評論都被折疊起來，這是因為它所評論的程式碼已經被更新了。

如果是在已發出的 Pull Request 中再加入新的提交並不會觸發通知，所以一旦 Tony 以這種方式推送修正，他需要再留下一個評論以通知專案擁有者：他已完成所要求的修改。

Three seconds is better #2

The screenshot shows a GitHub Pull Request titled "Three seconds is better #2". The pull request is open and merges the branch "tonychacon:slow-blink" into "schacon:master". There are 3 commits and 1 file changed. A comment from tonychacon links to a study on optimal LED delays. schacon comments on an outdated diff, and tonychacon adds commits to fix trailing whitespace. The pull request is automatically mergeable.

圖表 95. 結束 Pull Request

有個有趣的東西就是如果你點開 Pull Request 的「Files Changed」分頁，你會得到一份「統整過的」差異表——也就是所有當這個主題分支被合併進主要分支時會做的變動。以「git diff」的方式來講，就是自動顯示給你對 Pull Request 指定的主題分支做「git diff master…<branch>」的結果。看 [決定要提到哪些資訊](#) 來了解更多關於這種差異表的事情。

另外一件你會注意到的就是 GitHub 會確認這個 Pull Request 是否能直接合併，並顯示一個能讓你直接在伺服器上做合併的按鈕。這個按鈕只有在你對這個倉儲有寫入權限，而且能簡易的合併時才會出現。當你按下這個按鈕時，GitHub 會做一個「非快速向前」的合併，意味著即使這個合併「能」以快速向前的方式處理，GitHub 還是會建立一個合併的提交。

你可以基於你的偏好改用這樣的方式：pull 這個分支下來，然後在本機合併進去。如果你把這個分支合併進 `master` 分支並推送上 GitHub，對應的 Pull Request 會自動關閉。

大部分的 GitHub 專案都使用著這樣的基本流程。建立主題分支，基於這分支建立 Pull Request，針對這個

做討論，可能還會在這分支上做更多變更，最後這個要求就被關閉或合併了。

筆記

不只有 Forks

有件很重要的事情是：你也可以對同個倉儲的兩個分支做 Pull Request。如果你跟別人在一個雙方都有寫入權限的專案編寫新功能；你可以推送一個主題分支到倉儲裡，然後以這個分支對同個專案裡的 **master** 建立一個 Pull Request，藉此來做程式碼審閱以及討論。這不需要 Fork。

Pull Request 的進階用法

現在我們已經講完關於對 GitHub 上的專案做貢獻的基本部份了。來看看一些讓你可以更有效率的使用 Pull Request 的小技巧吧。

把 Pull Request 做成補丁

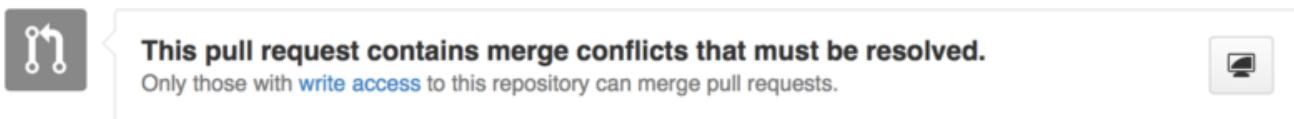
有件很重要的事情是：很多專案並不會把這些 Pull Request 當成一系列可以乾淨確實的使用的完美補丁，就像許多基於郵件清單運作的專案對系列補丁貢獻的看法。大多數的 GitHub 專案是把 Pull Request 分支用來做多次對期望變更的交流溝通，並將結果集中在一個差異檔，並用其做合併。

這是個很重要的差異，因為通常變動會在程式碼完備之前就被提出，這點跟基於郵件清單的運作模式是天差地遠的。這讓維護者們可以更早做溝通，讓適合的解決方案可以在接受更多社群能量下誕生。當有人使用 Pull Request 提出程式碼，然後維護者或是社群建議了一個變更，雖然這補丁系列不會重來，但相對的會以一個新的提交的形式加入這個分支，並讓討論和背景可以齊頭並進。

舉例來說，你可以回到 [結束 Pull Request](#) 這邊看看，你會注意到那個貢獻者並沒有把他的提交重組之後另外開個新的 Pull Request，而是加入新的提交並推送到原本的分支。如果之後你回去看看 Pull Request，你可以看到之前我們為何做了這樣的變動的背景。當按下網站上的「Merge」鈕時，會建立一個參考那個 Pull Request 的合併提交，這樣當有需要時你可以很容易的就找到它並研究當初的交談內容。

跟上上游

如果你的 Pull Request 因為過期或是其他原因導致不能很乾淨的合併，你就會希望去處理這個問題來方便維護者合併它。GitHub 會自動對這點做測試並在頁面最下方告訴你這個 Pull Request 是否能簡易的合併。



圖表 96. 不能乾淨的合併的 Pull Request

如果你看到類似 [不能乾淨的合併的 Pull Request](#) 的畫面，你就會希望去對你的分支做修正讓那個標示轉綠，之後維護者就不需要做額外的事。

你有兩個方式可以來處理這個狀況。你可以用變基把你的分支接在目標分支 (通常會是你 fork 的專案的 **master** 分支) 上，或是把目標分支合併進你的分支。

大部分在 GitHub 上的開發者都會選擇後者，基於上個章節所提的理由：我們看重的是歷史紀錄和最終的合併，所以變基除了給你一個乾淨一點點的歷史之外，你得到的只會是「非常」大的困難並且更容易犯錯。

如果你想要先對目標分支合併使得你的 Pull Request 能被自動合併，你可以把原始倉儲設定成一個新的遠端、從上面擷取資訊、把那個倉儲的主要分支合併進你的主題分支，修正任何可能的問題，最後再推送回你的開 Pull Request 的 主題分支。

舉之前「tonychacon」的例子來說，原始作者做了一個會和 Pull Request 衝突的變更。所以我們來看看解決這個問題的步驟吧。

```
$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master      -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
  ef4725c..3c8d735  slower-blink -> slow-blink
```

① 將原本的倉儲新增為遠端，並取名為「upstream」

② 從這個遠端擷取最新的內容

③ 把主要分支合併進你的主題分支

④ 修正產生的衝突

⑤ 再推送回同一個主題分支

當你做完上述步驟，Pull Request 會自動更新並檢查是否能乾淨的合併。

The screenshot shows a GitHub pull request interface. At the top, a comment from user **tonychacon** is visible, stating: "I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?". Below this, a commit message says: "Merge remote-tracking branch 'upstream/master' into slower-blink ...". To the right of the commit message is the hash **3c8d735**. A note below the commit message reads: "Add more commits by pushing to the **slow-blink** branch on **tonychacon/blink**". A green box highlights a message: "This pull request can be automatically merged by project collaborators. Only those with [write access](#) to this repository can merge pull requests." An icon of a computer monitor is shown next to this message.

圖表 97. Pull Request 現在能乾淨的合併了

Git 偉大的事情之一就是你可以一直重複這個過程。當你有個長期運作的專案時，你可以很簡單的重複對目標分支做合併，而你只要對最近一次的合併產生的衝突做處理即可，讓這個程序易於管理。

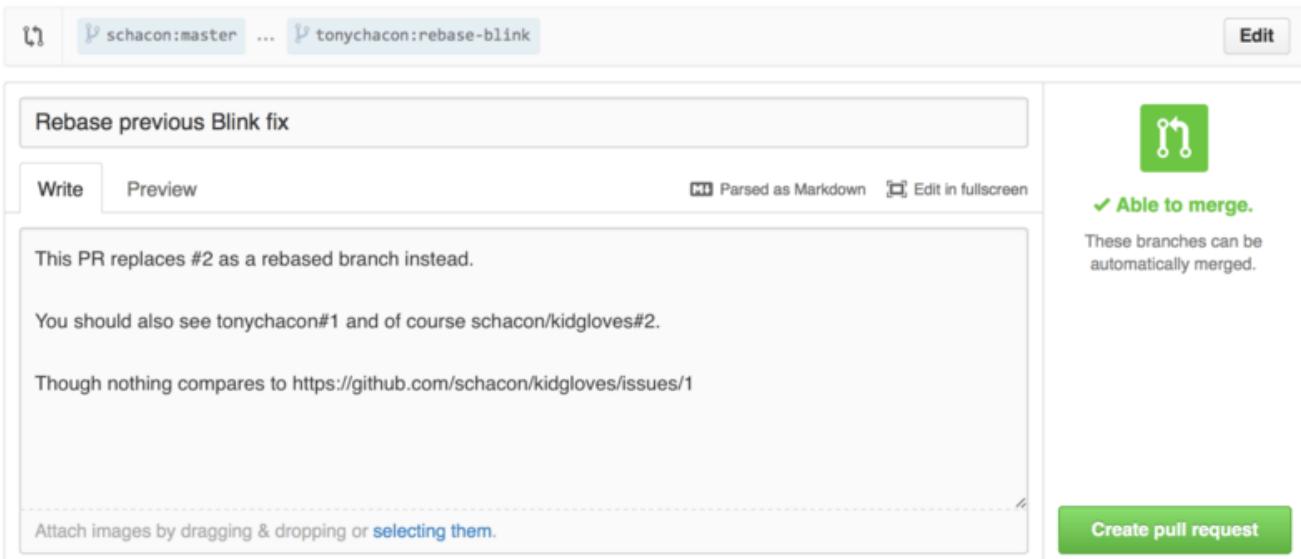
如果你一定要對分支做變基，你還是可以這樣做，不過強烈建議你不要強制對已經開了 Pull Request 分支做推送。如果其他人已經 pull 下來而且做了些變動，你會遇到所有在 [使用衍和的危險](#) 描述的問題。相對的，應該要把變基過的分支推送到 GitHub 上的新分支，然後建立一個參考至舊的 PR 的新 Pull Request，之後關閉原本的 PR。

參考

你的下一個問題可能是「我要怎麼對舊的 Pull Request 做參考連結？」。這有非常非常多的方法可以對其他東西做參考連結，幾乎所有你在 GitHub 上能撰寫訊息的地方都可以做到。

先從怎麼在 Pull Request 或是議題互相做參考開始吧。所有的 Pull Request 在專案裡都會被賦與一個獨一無二的編號。舉例來說你不能同時擁有 Pull Request #3 和議題 #3。如果你要在 Pull Request 裡參考其他的 Pull Request 和議題，你只要在評論或描述打下 `#<num>` 即可。你也可以指定參考不在同個專案裡的專案；如果是在同一個倉儲的 fork 裡你可以用 `username#<num>` 指定，或是 `username/repo#<num>` 來指定別人的其他倉儲裡的專案。

來看個範例吧。假設我們們在上個範例選擇使用變基處理分支，並為此開了新的 Pull Request，之後我們想要在新的 PR 裡放個參考連結到舊的。而且我們也想要參考一個在這個倉儲的 fork 裡的議題，還有一個在完全不同的專案裡的議題。我們的描述就可以用 [Pull Request 裡的跨倉儲參考](#) 裡的寫法。



圖表 98. Pull Request 裡的跨倉儲參考

當我們送出這個 pull request，我們可以看到內容被渲染成 在 Pull Request 中被渲染後的跨倉儲參考。裡的型式。

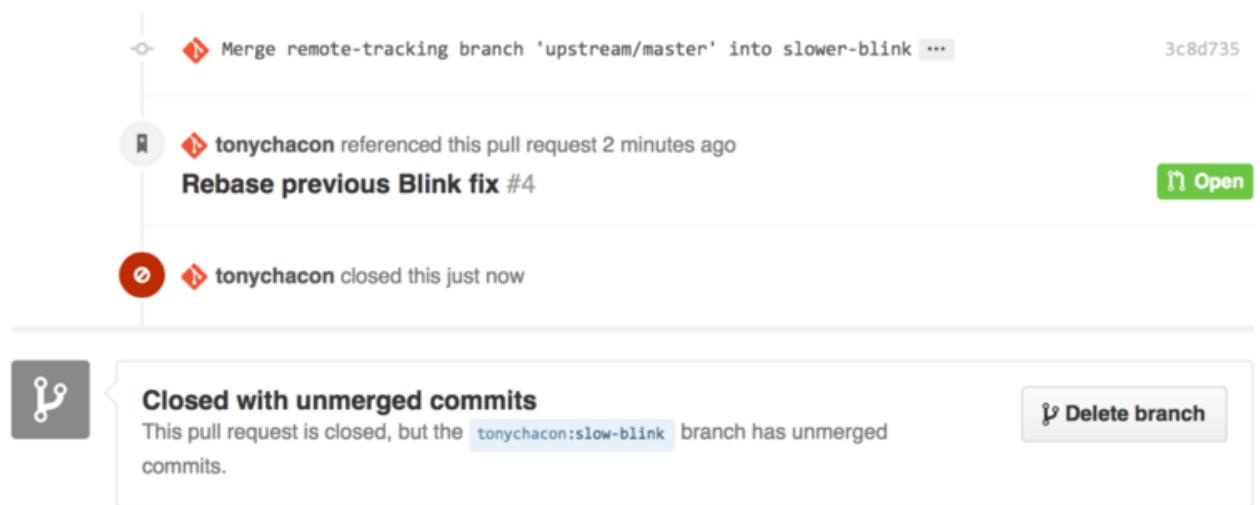
Rebase previous Blink fix #4

A screenshot of a GitHub pull request page for a specific commit. It shows a green 'Open' button, the author 'tonychacon', and the target branch 'schacon:master'. Below this, there are tabs for 'Conversation' (0), 'Commits' (2), and 'Files changed' (1). A comment from 'tonychacon' is shown, containing the same three paragraphs as the previous screenshot. Below the comment, a commit history is visible, showing 'tonychacon' adding some commits 4 hours ago, with two commits listed: 'three seconds is better' (commit hash: afe904a) and 'remove trailing whitespace' (commit hash: a5a7751).

圖表 99. 在 Pull Request 中被渲染後的跨倉儲參考。

可以注意到那完整的 GitHub 網址被簡化了，只留下必需的資訊。

如果 Tony 現在去關閉原本的 Pull Request，當我們在新的 PR 標記它時會得知這件事情，因為 GitHub 會自動在 PR 的時間線上對這件事做反向追蹤。這意味著所有造訪舊的 PR 頁面的人會知道這個 PR 已經被一個新的 PR 取代了，並且能簡單的透過連結造訪新的 PR 頁面。這連結看起來就是 在 Pull Request 中被渲染後的跨倉儲參考。這樣。



圖表 100. 在 Pull Request 中被渲染後的跨倉儲參考。

除了議題編號之外，你也可以用 SHA-1 對一個提交做參考。你必須完整的標出 40 字元的 SHA-1，然後 GitHub 在評論裡看到那個 SHA-1 時就會產生該提交的超連結。而且你也可以用和議題一樣的方式，對其他 fork 甚至是其他倉儲的提交做參考。

Markdown

連結其他議題，對於你在 GitHub 大多數的文字方塊裡能做的有趣事情而言，只是個開始。在議題、PR 描述、評論、程式碼評論，以及其他更多的地方，你都可以使用「GitHub 風格的 Markdown」。Markdown 能以純文字方式編輯，但能渲染出豐富的內容。

看看 [一個顯示出 Markdwon 撰寫型式和渲染結果的範例](#) 的範例來看看文字和評論能怎樣撰寫，並接著以 Markdwon 的方式渲染。

A screenshot of a GitHub issue page titled "A Markdown Example". The left pane shows the raw Markdown code and its rendered preview. The preview includes a list of items, a blockquote, and an image reference. The right pane shows a comment from tonychacon with a detailed response about a problem with a blink code. It includes a large "git" logo at the bottom.

圖表 101. 一個顯示出 Markdwon 撰寫型式和渲染結果的範例

GitHub Flavored Markdown

GitHub 風格的 Markdown 增加了許多你在基本 Markdown 語法做不到的事。這些在你要建立有用的 Pull Request、議題評論、描述時，會非常的有用。

工作清單

第一個專屬於 GitHub 的 Markdown 功能，特別是在 Pull Request 上，就是工作清單。工作清單就是一系列對應到你想要完成的事情的核取方塊。把這放在議題或是 Pull Request 裡時，通常表明了你想要完成的事項。

你可以建立這樣的工作清單：

- [X] 撰寫程式碼
- [] 撰寫所有的測試項
- [] 為程式碼做文件

如果我們在 Pull Request 的描述裡或是議題裡加入這個，我們就能看到他被渲染成像 [Markdwon 評論裡渲染後的工作清單](#)。這樣。



圖表 102. [Markdwon 評論裡渲染後的工作清單](#)。

這個功能在 Pull Request 裏面，常被用來聲明在合併之前，你想要在這個分支裡完成的事情。最酷的地方就是你只要點下核取方塊就能更新你的評論—你不需要為了標記工作完成而得修改 Markdown。

除此之外，GitHub 還會把議題和 Pull Request 裡面所有的工作清單整理起來，把它們作為後設資料顯示在 Pull Request 的清單頁面。舉例來說，如果你的 Pull Request 裡面有工作清單，你可以在所有 Pull Request 的總覽頁面上看到進度。這讓人們得以把一個 Pull Request 分解成數個小工作，同時也便於其他人追蹤這個分支上的進度。你可以在 [在 Pull Request 清單裡的工作清單統整](#)。看到關於這個功能的範例。

A screenshot of a GitHub pull request list. At the top, there are filters: '2 Open' and '1 Closed'. To the right are dropdown menus for 'Author', 'Labels', 'Milestones', 'Assignee', and 'Sort'. Below the filters, there are two pull requests listed:

- #4 Change blink time to four seconds - opened 4 hours ago by tonychacon - 2 of 3
- #2 Three seconds is better - opened 7 hours ago by tonychacon - 3

Each pull request has a green icon followed by its title and details.

圖表 103. 在 Pull Request 清單裡的工作清單統整。

當你在實作一個功能的開始就開了 Pull Request，並使用工作清單追蹤進度時，這個功能會驚人的好用。

程式碼摘錄

你也可以在評論裡摘錄某段程式碼。當你想要展示某段還沒提交到分支的變更時，這會非常的有用。這在展示無法正常運作或是這個 Pull Request 可以實作的的程式碼時也會用到。

你摘錄的程式碼需要用反引號「包」起來。

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```

```

如果你在上面那段代碼的 java 的位置放上其他的程式語言名稱，GitHub 也會試著做語法上色。如果以我們上面的範例來說，上面那段代碼最後會被渲染成 [被渲染過的嵌入程式碼片斷](#) 的樣子。



圖表 104. 被渲染過的嵌入程式碼片斷

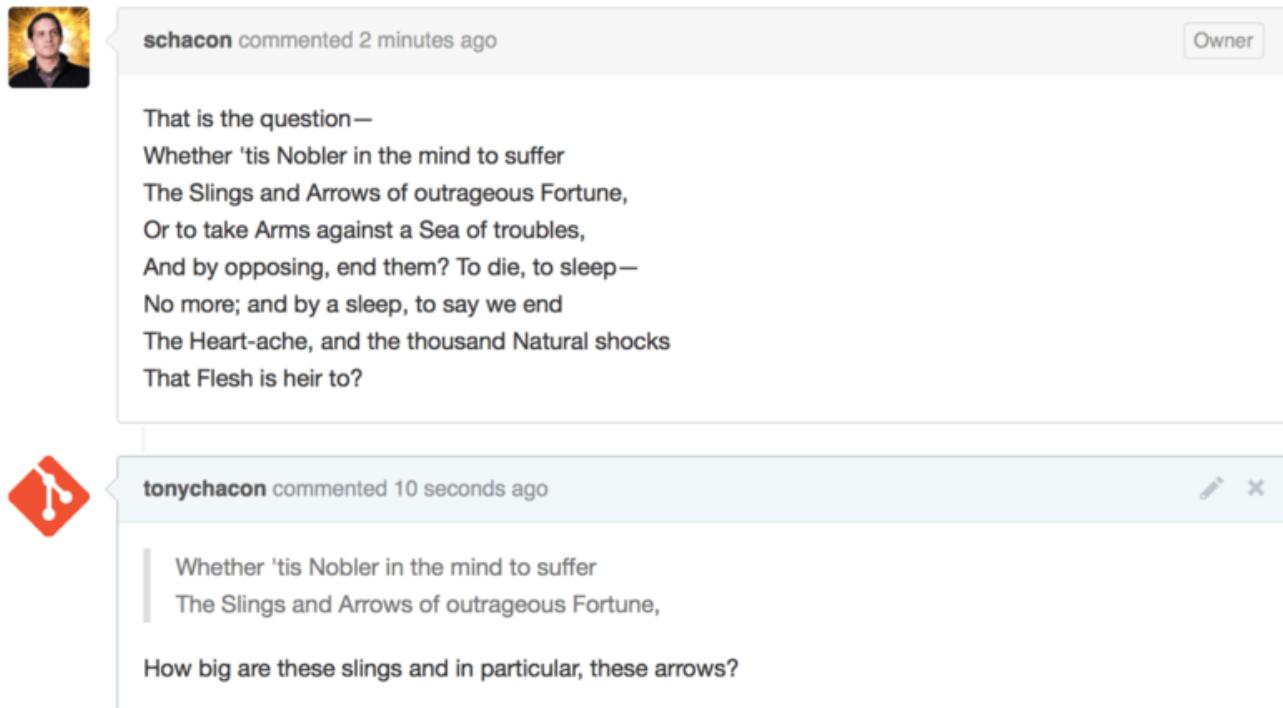
引文

如果你要對一長段評論的一部份做回應，你只要複製你需要的片斷，然後在前面加上 `>` 即可。事實上，因為這個功能因為太實用也太常用到，所以有一個專用的快捷鍵可用。如果你把評論你要回應的文字反白起來，並按下 `r` 鍵，那段文字就會被引文到評論欄裡供你使用。

引文的部份看起來就像這樣：

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,
How big are these slings and in particular, these arrows?
```

經過渲染之後評論就會變成 [渲染過的引文範例](#) 這個樣子。



The screenshot shows two GitHub comment cards. The top card is from user schacon, commented 2 minutes ago, containing a quote from Shakespeare's Hamlet:

That is the question—
 Whether 'tis Nobler in the mind to suffer
 The Slings and Arrows of outrageous Fortune,
 Or to take Arms against a Sea of troubles,
 And by opposing, end them? To die, to sleep—
 No more; and by a sleep, to say we end
 The Heart-ache, and the thousand Natural shocks
 That Flesh is heir to?

The bottom card is from user tonychacon, commented 10 seconds ago, with a partial quote:

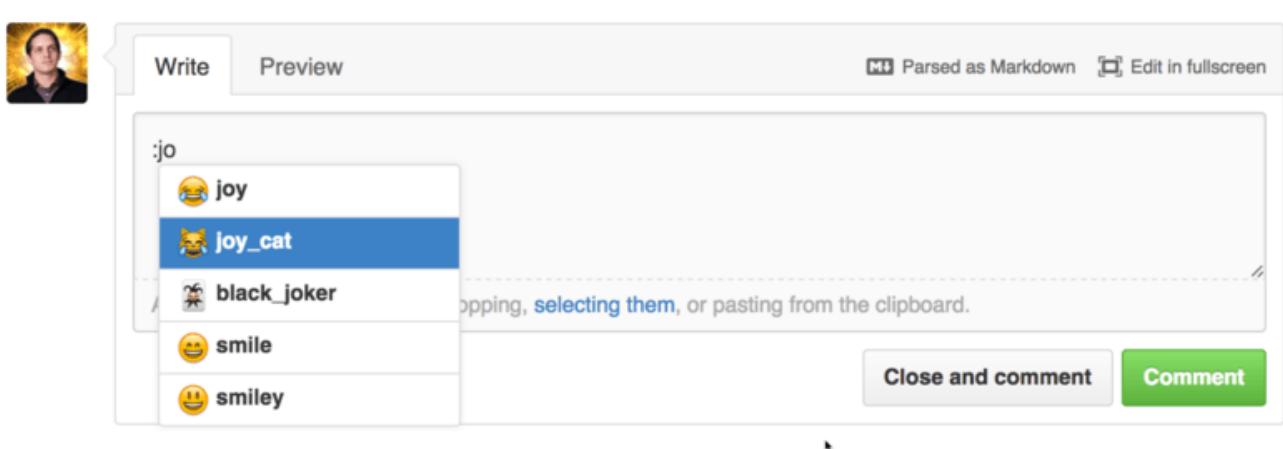
Whether 'tis Nobler in the mind to suffer
 The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?

圖表 105. 渲染過的引文範例

表情符號

最後就是你可以在評論裡使用表情符號。這個東西很常出現在許多 GitHub 議題和 Pull Request 的評論裏面。GitHub 上甚至有表情符號工具。如果你在評論裡用了 : 當作開頭，自動完成會協助你找出你想要的表情。



The screenshot shows the GitHub emoji picker interface. A user has typed ':jo' into the input field. A dropdown menu lists several options starting with 'jo':

- joy
- joy_cat** (highlighted)
- black_joker
- smile
- smiley

Below the dropdown, there is a note: 'Type another character to keep popping, selecting them, or pasting from the clipboard.' At the bottom right are 'Close and comment' and 'Comment' buttons.

圖表 106. 表情符號的自動完成提示

你可以在評論的任何地方打出 :<name>: 來使用表情符號。舉例來說，你可以寫出這樣的東西：

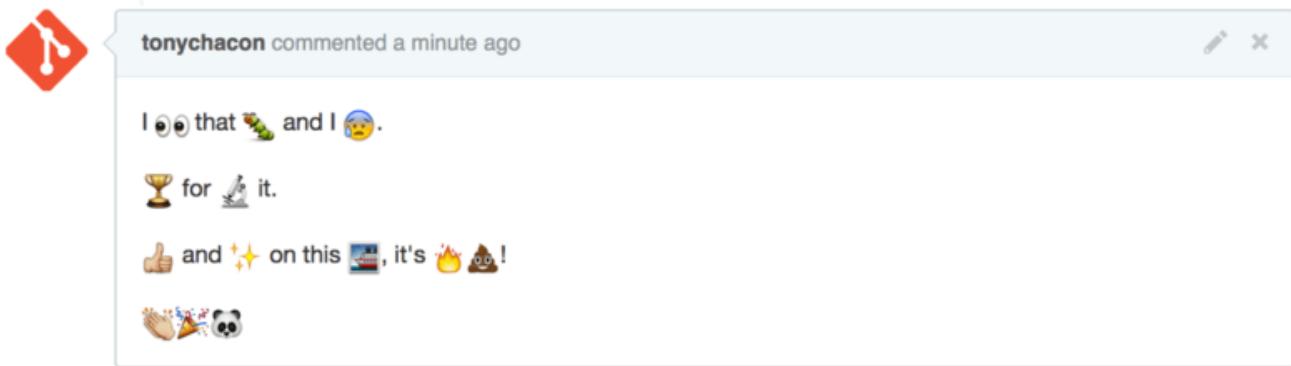
```
I :eyes: that :bug: and I :cold_sweat:.  

:trophy: for :microscope: it.  

:+1: and :sparkles: on this :ship:, it's :fire::poop:!  

:clap::tada::panda_face:
```

經過渲染之後會變成這樣 使用大量表情符號的評論：



圖表 107. 使用大量表情符號的評論

雖然不能說它是個非常實用的功能，但它能在這種不方便表達情緒的媒介裡，加入了由趣味和心情構成的元素。

筆記

事實上現在有不少的網路服務可以在上面使用表符字元。這邊有張非常好用的大抄可以讓你很快到找到能表達你現在情緒的符號：

<http://www.emoji-cheat-sheet.com>

圖片

技術上來說，雖然這並不是 GitHub 風格的 Markdown，但是還是非常的實用。如果不想用 Markdown 圖片語法這種很難知道是什麼圖片的方法之外，GitHub 允許你以把圖片拖曳至文字方塊的方式來嵌入圖片。

This is the wrong version of Git for the website:

Git.png

Attach images by dragging & dropping or selecting them.

This is the wrong version of Git for the website:

![git](https://cloud.githubusercontent.com/assets/7874698/4481741/7b87b8fe-49a2-11e4-817d-8023b752b750.png)

Attach images by dragging & dropping or selecting them.

圖表 108. 以拖曳的方式來上傳並自動嵌入圖片

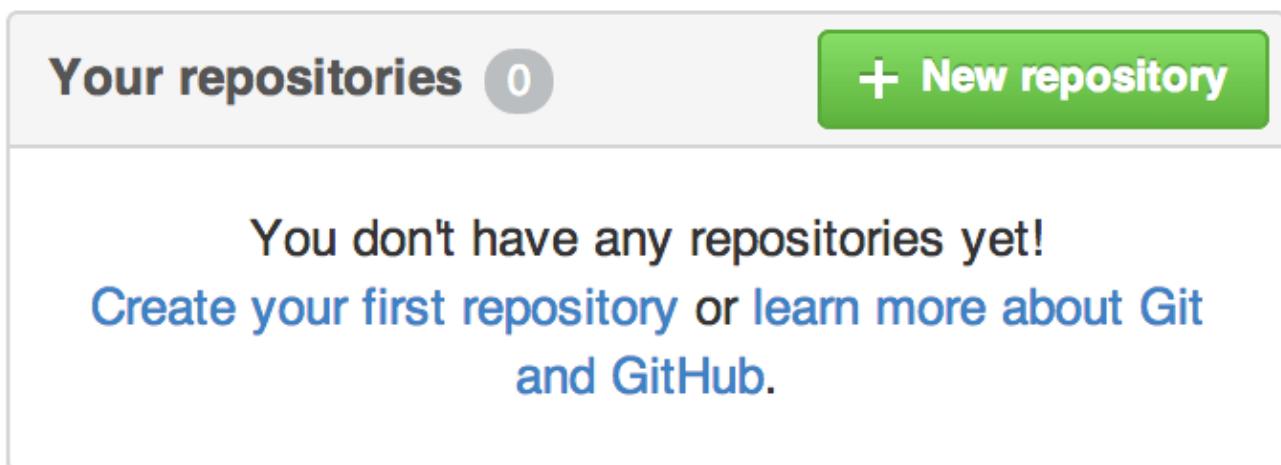
如果你回到 Pull Request 裡的跨倉儲參考，你會在文字區塊上看到一個小小的「Parsed as Markdown」提示。點一下那個提示，他就會提供你包含所有在 GitHub 上可以用 Markdown 做的事的小抄。

維護專案

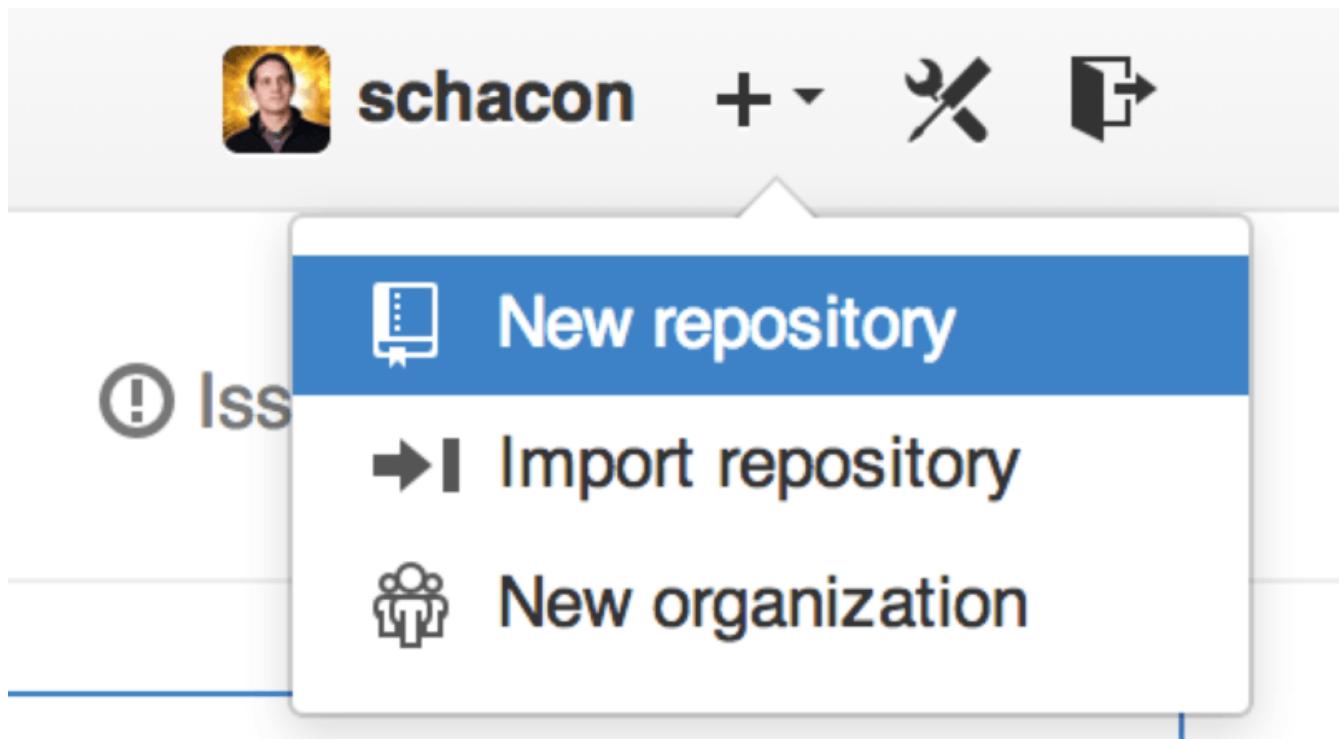
現在我們可以舒適自在地對一個專案做出貢獻了，所以我們來看看另一個面向：建立、維護以及管理一個專案。

建立一個新倉儲

來建立一個拿來分享我們的原始碼的倉儲吧。首先點擊主控面板右邊的「New Repository」；或是點擊頂端工具列裡面使用者名稱旁邊的 + 按鈕，如 「New repository」 下拉式選單. 所示。



圖表 109. 「Your Repositories」 區塊



圖表 110. 「New repository」 下拉式選單.

這會把你帶到「new repository」表單的所在頁面：

Owner Repository name

PUBLIC  / iOSApp ✓

Great repository names are short and memorable. Need inspiration? How about [drunken-dubstep](#).

Description (optional)

iOS project for our mobile group

 **Public**
Anyone can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with a README
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** | ⓘ

Create repository

圖表 111. 「new repository」表單

你只需要提供專案名稱，因為剩餘的欄位是完全選擇性的。現在，你只要點下「Create Repository」鈕，然後碰地一聲——你就擁有了一个在 GitHub 上名為 `<user>/<project_name>` 的全新倉儲了。

因為你這個倉儲還沒有任何原始碼在裡面，GitHub 會展示一份關於如何建立一個全新的 Git 倉儲或是連結一個舊有 Git 專案的指引。我們在這邊不會對這部份多做描述，如果你需要回憶一下，去看看 [Git 基礎](#) 吧。

現在你的專案被託管在 GitHub 上了，你可以把網址給任何你想要分享專案的人。所有人都可以透過 https://github.com/<user>/<project_name> 以 HTTP 方式存取，或是透過 git@github.com:<user>/<project_name> 以 SSH 方式存取。Git 可以透過上述兩種途徑來推送及擷取資料，但所有操作都會透過對其連結的使用的驗證資訊來做存取控管。

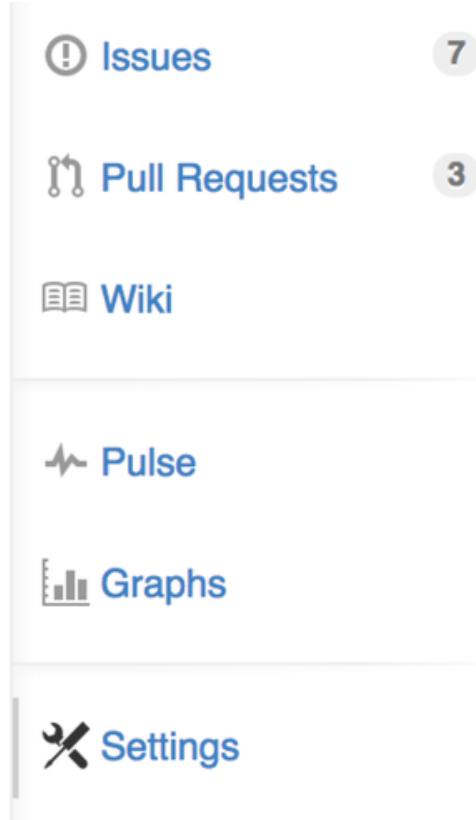
筆記

通常公開專案會傾向於分享基於 HTTP 的網址，因為這樣沒有 GitHub 帳戶的使用者也能夠對其存取來拓製專案。如果你給了 SSH 版本的網址，使用者必須建立一個帳戶並加入 SSH key 才能存取專案。而且 HTTP 網址就是他們會貼在瀏覽器裡來瀏覽專案的網址。

增加協作者

你必須要把和你合作的人加入「協作者」，這樣他們才能對專案提交變更。如果 Ben、Jeff 和 Louise 都有 GitHub 帳戶，而且你想要給他們推送變更的權限，你可以把他們加到你的專案。把他們加入專案後，他們可以對專案「推送」變更，這意味著他們有這專案及專案的 Git 倉儲的讀寫權限。

點擊右側欄最下面的「Settings」連結。



圖表 112. 倉儲設定連結。

接著選擇右邊選單的「Collaborators」。然後在文字方塊裡輸入使用者名稱，按下「Add collaborator」。你可以一直重複這個步驟來賦予所有你想要的人存取權限。如果你要收回權限，只要點一下那個使用者右手邊的「X」即可。

| Options | Collaborators | Full access to the repository |
|---|---|-------------------------------|
| Collaborators |  Ben Straub
ben | X |
| Webhooks & Services |  Jeff King
peff | X |
| Deploy keys |  Louise Corrigan
LouiseCorrigan | X |
| | <input type="text" value="Type a username"/> Add collaborator | |

圖表 113. 倉儲協作者

管理 Pull Requests

現在你擁有一個裡面有些原始碼的專案，同時也可能會有些擁有推送權限的協作者。然後我們來看看當你收到 Pull Request 時要如何處理吧。

Pull Requests 可能是來自某個 fork 裡的分支，或是同個倉儲裡的某個分支。兩者之間只差在，來自其他 fork 的 PR 通常你沒有對他們分支的推送權限，他們也沒有；而內部 PR 就是雙方都能存取分支。

關於這些東西的範例，我們就在這邊假設你是「tonychacon」而且你建立了一個名叫「fade」的 Arduino 原始碼專案吧。

電郵通知

有人對你的原始碼做了些變更，然後發給你一個 Pull Request。這時你應該會收到像 [對於新的 Pull Request 的電郵通知](#)。這樣的郵件。

The screenshot shows an email from Scott Chacon at notifications@github.com. The subject is "[fade] Wait longer to see the dimming effect better (#1)". The email body contains the following text:

One needs to wait another 10 ms to properly see the fade.

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

<https://github.com/tonychacon/fade/pull/1>

Commit Summary

- wait longer to see the dimming effect better

File Changes

- M [fade.ino](#) (2)

Patch Links:

- <https://github.com/tonychacon/fade/pull/1.patch>
- <https://github.com/tonychacon/fade/pull/1.diff>

—
Reply to this email directly or [view it on GitHub](#).

圖表 114. 對於新的 Pull Request 的電郵通知。

這電郵裡面有些值得注意的東西。他會給你一個簡易的差異狀態——一個在這 Pull Request 之中被變更的檔案清單，以及變動量。內附一個 GitHub PR 連結。同時也會給你一些可以從指令列操作的網址。

你可能會注意到這行指令 `git pull <url> patch-1`，這行指令是可以在不用增加遠端的情況下合併一個遠端分支的簡易方式。我們曾在 [切換到遠端分支](#) 簡短的提過。你可以去建立並切換至主題分支，然後執行這條指令以合併 Pull Request 中的變更。

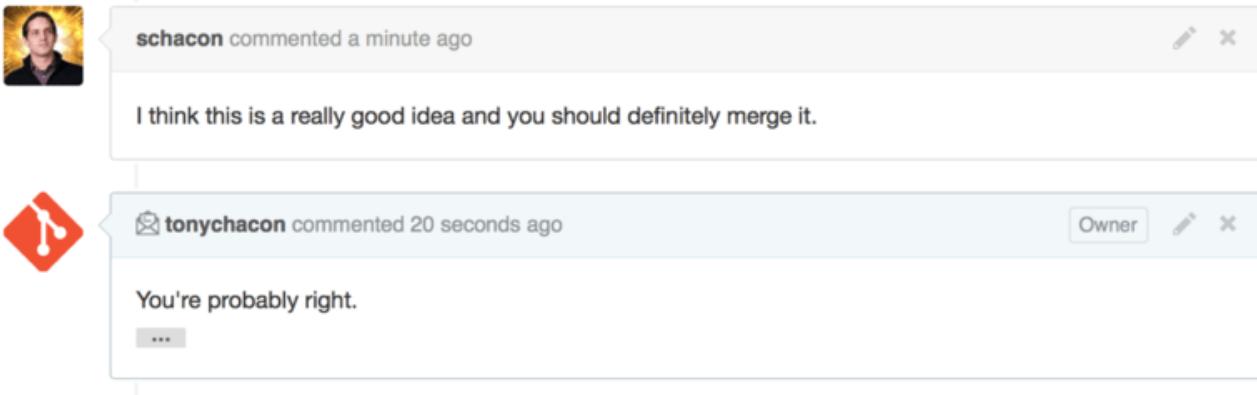
其他有趣的網址就是 `.diff` 和 `.patch`，你或許猜的到，他們分別提供 Pull Request 的統合差異和系列補綴。你可以以下述方式來做技術性的合併：

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

在 Pull Request 裡合作

就如我們在 [GitHub 流程](#) 提過的，你現在可以和建立 Pull Request 的人對談了。你可以針對某幾行原始碼提出評論、對一整個提交做評論或是對一整個 Pull Request 做評論，而且你在其中每個部分都可以使用 GitHub 風格的 Markdown。

每當有人在 Pull Request 裡留下評論時，你都會收到一封電郵通知，這樣你就能掌握整個 Pull Request 的動態。每次通知都會附有連結連往 Pull Request 有活動的部分，同時你也可以直接回覆郵件以在 Pull Request 討論串中評論。



圖表 115. 包含在討論串中的電郵回覆。

當你覺得 PR 裡面的原始碼已經可以合併的時候，你可手動拉取，然後在本地端合併；或是用 `git pull <url> <branch>` 這個稍早看過的語法；也可以把那個 fork 加入成遠端之一之後再擷取並合併。

如果這只是個簡單的合併，你可以按下 GitHub 網站上的「Merge」按鈕來解決。這會做一個「非快進」的合併——即便是可以快進，仍會建立合併提交。這意味著，不論你在何種情況按下「Merge」，都會建立一個合併提交。如果你點了提示連結，GitHub 就會提供你所有的資訊，就像你在 [Merge 按鈕和手動合併 Pull Request 的指引](#) 看到的一樣。

A screenshot of the GitHub pull request interface showing the 'Merge pull request' button. Below it, there's a section titled 'Merging via command line' with instructions and command-line examples. The interface includes tabs for HTTP, Git, and Patch, and a clipboard icon.

圖表 116. Merge 按鈕和手動合併 Pull Request 的指引。

如果你決定不要合併這個 Pull Request，你只需要關閉這個 Pull Request 即可，同時建立這個 Pull Request 的人也會收到通知。

Pull Request 參照

如果你要處理 **非常多** 的 Pull Request 而且不想加入一堆的遠端或是一直做只會用到一次的拉取，關於這點 GitHub 提供了一個好用的小技巧給你用。這是個有點進階的技巧，所以我們會在 [The Refspec](#) 提到更多的細節，不過還是非常的有用。

事實上 GitHub 會把倉儲的 Pull Request 當成伺服器上的假分支。預設情況下你不會在拓製的時候取得它們，但他們還是以隱藏的狀態存在著而且你可以用非常簡單的方式取得它們。

To demonstrate this, we’re going to use a low-level command (often referred to as a “plumbing” command, which we’ll read about more in [Plumbing and Porcelain](#)) called `ls-remote`. This command is generally not used in day-to-day Git operations but it’s useful to show us what references are present on the server. 為了展示這個事實，我們要使用一個比較低階的指令（通常會被稱作「底層」指令，關於這點我們會在 [Plumbing and Porcelain](#) 再做詳細描述）——`ls-remote`。這指令通常不會在日常的 Git 操作使用，但在展現伺服器上的所有參照是非常有用的。

如果對我們之前的「blink」倉儲使用這條指令，我們會得到在伺服器上這個倉儲裡所有的分支、標籤和其他各種參照的清單。

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d      HEAD
10d539600d86723087810ec636870a504f4fee4d      refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e      refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3      refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfb2665adec1      refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d      refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a      refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c      refs/pull/4/merge
```

當然，如果你在你的倉儲裡使用 `git ls-remote origin` 或是對其他任一個想確認的遠端使用，你會得到一個與這類似的結果。

如果你的倉儲是在 GitHub 上，而且有開放中的 Pull Request；你會看到一些以 `refs/pull/` 開頭的參照。它們基本上也是分支，但是因為他們不是以 `refs/heads` 起頭，所以你一般來說不會在拓製或擷取時從伺服器下載到他們——擷取的過程會忽略它們。

每個 Pull Request 都會有兩個參照——以 `/head` 結尾的是對應到目前 Pull Request 分支的最後一個提交。所以，如果有人在我們的倉儲開了一個 Pull Request，而且他的分支名稱叫做 `bug-fix` 還有它指向 `a5a775` 這個提交，在之後我們的倉儲裡並不會出現 `bug-fix` 這個分支，但我們會出現指向 `a5a775` 的 `pull/<pr#>/head`。這意味著我們可以非常簡單的直接拉取所有 Pull Request 分支，而非加一大堆的遠端來解決。

現在，你可以直接做些事——好比說擷取參考。

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch                  refs/pull/958/head -> FETCH_HEAD
```

這告訴了 Git：「連結到名為 `origin` 的遠端，並下載名為 `refs/pull/958/head` 的參考。」Git 很樂意的照做了，之後下載了所有建立這個參考的必要資訊，之後就在 `.git/FETCH_HEAD` 裡放了一個指向你所需要的提交的指針。你可以接著在你想要測試的分支裡執行 `git merge FETCH_HEAD`，不過這個合併提交的訊息可能會有點奇怪。不過，如果你需要審閱「一大堆」的 Pull Request，這會顯得相當的枯燥乏味。

也是有辦法可以擷取「全部」的 Pull Request 的，而且可以在你每次連接到遠端時更新。用你習慣的編輯器打開 `.git/config`，並且找到關於遠端 `origin` 的部份。通常會長的像這樣：

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2
  fetch = +refs/heads/*:refs/remotes/origin/*
```

以 `fetch =` 開頭的那行是個「參照規格」。這是個本地端的 `.git` 資料夾裏面的名稱對應到遠端的方法。這一段告訴 Git：「在遠端 `refs/heads` 之下的東西，要保存在本地倉儲的 `refs/remotes/origin` 之下。」你可以編輯這個段落以加入其他的參照規格：

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

最後一行告訴 Git：「所有類似 `refs/pull/123/head` 的參照都會以 `refs/remotes/origin/pr/123` 的型式儲存在本地端。」所以現在存檔，然後執行 `git fetch`：

```
$ git fetch
# ...
* [new ref]          refs/pull/1/head -> origin/pr/1
* [new ref]          refs/pull/2/head -> origin/pr/2
* [new ref]          refs/pull/4/head -> origin/pr/4
# ...
```

現在所有遠端的 Pull Request 都會以類似追蹤分支型式的參照出現在本端；他們是唯讀的，並且會在你做擷取的時候更新。這讓在本地端測試 pull request 裡的程式碼變成超級簡單的事：

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

你們其中某些比較眼尖的人可能已經注意到參照規格中遠端部份的尾巴的 `head`。這同時也是 GitHub 端一個名為 `refs/pull/#/merge` 的參照，這代表著當你點下網站上的「Merge」鈕時會產生的提交。這讓你甚至可以在按下按鈕前測試合併結果。

對應到 Pull Request 的 Pull Request

你不只可以對主要分支或是 `master` 分支建立 Pull Request，你也可以對這整個網絡裡的任意一個分支做同樣的事。事實上，你甚至可以對另外一個 Pull Request 做 Pull Request。

如果你發現一個 Pull Request 正在往良好的方向發展，而你也想加點依賴於它的變更、或是你只是不確定這變更是否是個好主意、甚至只是因為你沒有對目的分支推送的權限時，你都可以直接對它開 PR。

當你要開啟一個 Pull Request 時，頁面最上方會有個區塊可以讓你選擇要請求擁有者在哪個分支上拉取變更，以及要從哪個分支拉取。如果你按下右手邊的「Edit」鈕，你不只能指定分支，也可以指定 fork。

The screenshot shows a GitHub pull request comparison between 'schacon:master' and 'tonychacon:patch-2'. At the top, there are buttons for 'Create pull request' and 'Discuss and review the changes in this comparison with others.' Below this, it shows '2 commits', '1 file changed', '0 commit comments', and '2 contributors'. The commit list includes a commit from schacon and one from tonychacon. A modal window titled 'Choose a base branch' is open, showing 'master' as the selected option, with 'patch-1' also listed.

圖表 117. 手動指定 Pull Request 的目標 fork 和分支

在這邊你可以相當簡單的方式決定你要把你的新分支合併到另一個 Pull Request 或是這專案底下的其他 fork。

提及他人和通知

GitHub 也有非常好的通知系統，讓徵詢特定人或團隊的意見變成一件易如反掌的事。

在任意一個專案裡，當你一輸入 @ 這個字元後，自動完成就會提供給你所有這個計畫的協作者和貢獻者的顯示名稱及使用者名稱。

The screenshot shows a GitHub comment input field. The user has typed '@' and a dropdown menu appears, listing suggestions for users: 'ben Ben Straub', 'peff Jeff King', 'jlehm...', and 'LouiseCorrigan Louise Corrigan'. There are buttons for 'Write' and 'Preview' at the top, and 'Parsed as Markdown' and 'Edit in fullscreen' on the right. At the bottom, there are 'Close and comment' and 'Comment' buttons.

圖表 118. 輸入 @ 來提到其他人

你也可以提及某個並沒有出現在這個下拉式清單的使用者，不過自動完成通常會更快把牠抓出來。

當你發佈了一個有提到其他使用者的評論時，那些被標記的人會收到通知。這意味著這會是一個非常有效率的把人拉入討論的方式，而非讓那些人主動追蹤討論進度。在 GitHub 上人們很常把同團隊或是同公司的人拉近討論裡，藉以審閱 Pull Request 或是議題。

如果有人被標記，會被自動訂閱提到他們的 Pull Request 或是議題，之後就會收到所有相關的動態。你也會自動訂閱所有你建立的東西、你觀注的版本庫或是你曾經發表過評論的東西。如果你不想繼續收到通知，你可以按下頁面裡的「Unsubscribe」鈕來停止收到後續的更新。

Notifications

🔈 × Unsubscribe

You're receiving notifications because you commented.

圖表 119. 取消訂閱 PullRequest 或是議題。

通知總覽頁面

當我們在這邊提到關於 GitHub 的「通知」時，這是指 GitHub 試著讓你跟上新發生的事件的方式，而且你也有許多方式可以自訂。如果進到設定頁面的「Notification Center」分頁，你會看到一些可以使用的選項。

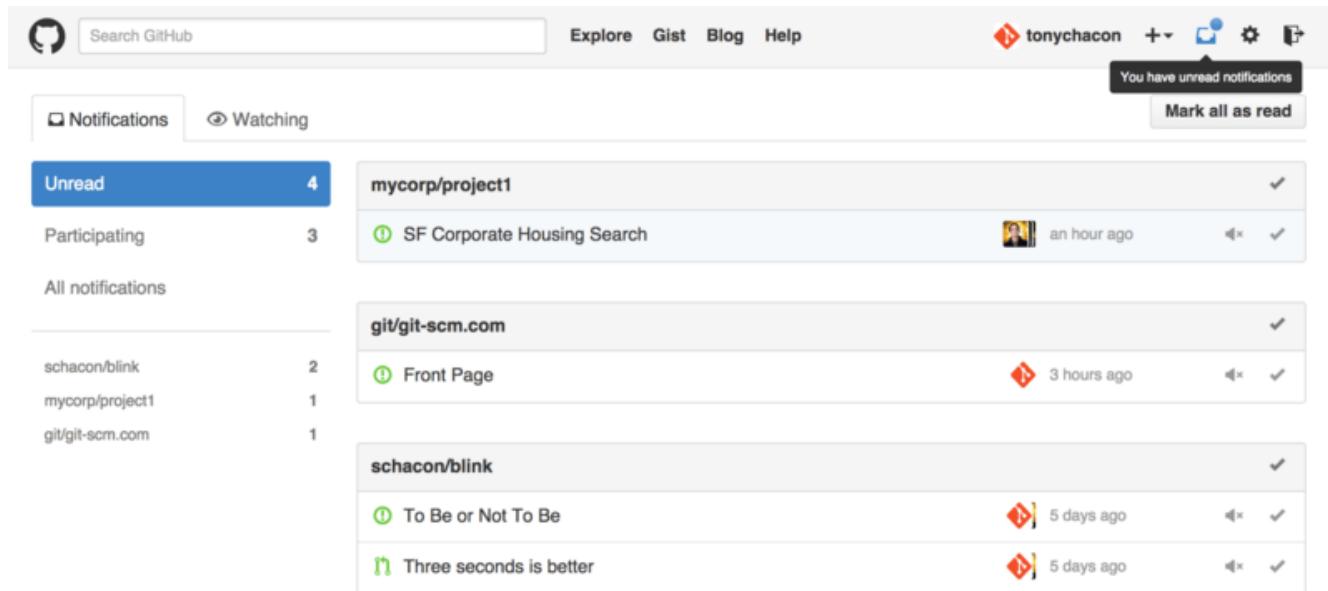
The screenshot shows the GitHub user profile for tonychacon. On the left is a sidebar with links: Profile, Account settings, Emails, **Notification center** (which is selected and highlighted in orange), Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main content area is titled 'How you receive notifications'. It has two sections: 'Participating' and 'Watching'. Under 'Participating', it says 'When you participate in a conversation or someone brings you in with an @mention.' with checkboxes for Email and Web, both of which are checked. Under 'Watching', it says 'Updates to any repositories or threads you're watching.' with checkboxes for Email and Web, both of which are checked. Below this is a section titled 'Notification email' with a 'Primary email address' field containing 'tchacon@example.com' and a 'Save' button. At the bottom is a section titled 'Custom routing' with the note 'You can send notifications to different **verified** email addresses depending on the organization that owns the repository.'

圖表 120. 通知中心選項。

在這邊你有兩種取得通知的方式可以選擇--「電子郵件」和「網頁」，你可以決定當你在參與事情或是接收你關注的版本庫的動態時，你要如何透過這兩個管道接受通知。

網頁通知

網頁通知僅存在於 GitHub 上，而你也只能在 GitHub 上檢視。如果你有選用這個選項，當你在收到通知的時候，你會在頁面上方的通知圖示上看到一個藍點，就如 [通知中心](#) 所示。



The screenshot shows the GitHub Notifications page. At the top, there's a search bar, navigation links for Explore, Gist, Blog, Help, and a user profile for tonychacon. A notification badge indicates 'You have unread notifications'. Below the header, there are two tabs: 'Notifications' (selected) and 'Watching'. The main area is divided into sections by repository:

- Unread**: Shows 4 notifications. One from 'mycorp/project1' about 'SF Corporate Housing Search' posted an hour ago. Another from 'git/git-scm.com' about the 'Front Page' posted 3 hours ago.
- Participating**: Shows 3 notifications. One from 'schacon/blink' about 'To Be or Not To Be' posted 5 days ago. Another from 'git/git-scm.com' about 'Three seconds is better' posted 5 days ago.
- All notifications**: Shows 1 notification from 'mycorp/project1'.

Each notification item includes a small icon, the repository name, the specific event, the time it was posted, and a checkbox for marking it as read.

圖表 121. 通知中心

如果你點一下那個圖示，你會看到以專案分類的通知清單。你也可以點擊-左側列表裡的專案名稱來過濾出關於個專案的通知。你也以按下任一項通知右側的勾選圖示來確認接收，或是點下專案名稱旁的勾選圖示來接收所有關於這個專案的通知。勾選圖示旁邊也有個靜音按鈕，當你點下那個圖示，代表你將不會收到那項東西的後續通知。

這些工具對於管理大量的通知來說是非常的方便。許多的 GitHub 的進階使用者會直接關掉所有的電郵通知，然後在這頁面上管理所有的通知。

電郵通知

電郵通知是 GitHub 上另一個接受通知的方式。如果你開啟這個選項，之後每當你收到通知，你就會收到一封電郵。我們可以在 [以電子郵件型式寄送的評論](#) 和 [對於新的 Pull Request 的電郵通知](#) 有範例。這廂電子郵件會自動討論串化，如果你在使用討論串式的電郵客戶端，這會是個好事情。

GitHub 會再寄給你的通知郵件的標頭里嵌入相當多的後設資訊，這樣你在建立自訂過濾條件時可以更得心應手。

舉例來說，如果展開在 [對於新的 Pull Request 的電郵通知](#)。寄給 Tony 的郵件的真實標頭，我們會看到這樣被寄出的訊息。

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...
X-GitHub-Recipient-Address: tchacon@example.com
```

這裡面有不少有趣的東西。如果你想針對特定專案甚至是 Pull Request 的電郵做特別標示或是轉寄時，「Message-ID」這項以 `<user>/<project>/<type>/<id>` 的格式提供了所有資訊。如果這是個議題，`<type>` 這部分就會是「issue」而非「pull」。

而而你如果有個能夠解析 `List-Post` 和 `List-Unsubscribe` 這兩個欄位的電郵客戶端的話，你就能直接發表意見到清單裡，或是直接「取消訂閱」這個串，就像你直接在那 Pull Request 或是議題頁面上做的一樣。

值得一提的是，如果你同時啟用網頁和電郵通知，當你使用允許內嵌圖片的郵件客戶端開啟通知郵件時，網頁上的通知也會被標示為已讀。

特殊檔案

當有些特殊檔案出現在你的版本庫時，GitHub 會注意到它們。

README

第一個就是 `README` 檔，它可以用任意一種可以被 GitHub 便是成文章的格式寫成。舉例來說，它可以是 `README`、`README.md`、`README.asciidoc`、等等。如果 GitHub 在你的原始碼裡發現 `README` 這個檔案時，他會被渲染再在專案首頁。

許多團隊用這檔案來整理所有這專案的相關資訊，來讓第一次接觸這個專案或版本庫的人了解這個專案。通常會包含這些東西：

- 這個專案的目的
- 如何設定及安裝
- 舉個使用或執行的例子
- 這個專案是基於哪種授權發布的
- 如何對這專案做貢獻

因為 GitHub 會渲染這個檔案，所以你可以嵌入圖片或連結來讓他更容易理解。

CONTRIBUTING

另個 GitHub 會辨識的特殊檔案就是 `CONTRIBUTING`。如果你有個帶著任意種附檔名的 `CONTRIBUTING`，GitHub 會把它顯示成 在有 `CONTRIBUTING` 這個檔案時建立 Pull Request 紿那些想建立 Pull Request 的使用者看。

Please review the [guidelines for contributing](#) to this repository.

The screenshot shows the GitHub interface for creating a pull request. At the top, there's a message: "Please review the [guidelines for contributing](#) to this repository." Below this is a title input field labeled "Title". Underneath the title is a toolbar with "Write" and "Preview" buttons, and status indicators "Parsed as Markdown" and "Edit in fullscreen". A large text area is labeled "Leave a comment". To the right of the text area is a note: "We can't automatically merge these branches." followed by the text "Don't worry, you can still create the pull request." At the bottom right is a green "Create pull request" button. A note at the bottom of the text area says "Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard."

圖表 122. 在有 *CONTRIBUTING* 這個檔案時建立 Pull Request

這檔案就是你可以指定發給你這專案的 Pull Request 裡面可以包含哪些以及不能包含哪些東西。這樣大家或許就會在建立 Pull Request 前看看這些原則了。

專案管理

你在一個專案上通常沒有什麼管理性的事能做，不過這邊有幾個你或許有興趣的專案。

變更主分支

如果你希望使用其他不是「master」的分支來作為其他人建立 Pull Request 的預設目標分支，你可以在版本庫的設定頁面的「Options」分頁裡設定。

The screenshot shows the GitHub repository settings page under the "Options" tab. On the left is a sidebar with "Collaborators", "Webhooks & Services", and "Deploy keys". The main area is titled "Settings" and contains fields for "Repository name" (set to "fade") and "Default branch". A dropdown menu is open over the "Default branch" field, showing three options: "master" (which is checked), "development", and "issue-53".

圖表 123. 變更一個專案的預設分支

只要在下拉式清單裡選擇你要的分支，就可以簡單的變更所有主要動作的預設分支，包含其他人在拓製版本庫時預設簽出的分支。

版本庫移轉

如果你想要把一個專案轉移給其他在 GitHub 上的使用者或是組織時，你可以在倉儲設定裡同樣在「Options」分頁的底部找到「Transfer ownership」來達成這件事。

Danger Zone

Make this repository private
Please [upgrade your plan](#) to make this repository private.

Transfer ownership
Transfer this repo to another user or to an organization where you have admin rights. [Transfer](#)

Delete this repository
Once you delete a repository, there is no going back. Please be certain. [Delete this repository](#)

圖表 124. 把一個專案轉移給其他的 GitHub 使用者或是組織

這當你要放棄一個專案而有人要接手時很有幫助，或是在你的專案日漸茁壯，你想要把它移到一個組織裡時也有用。

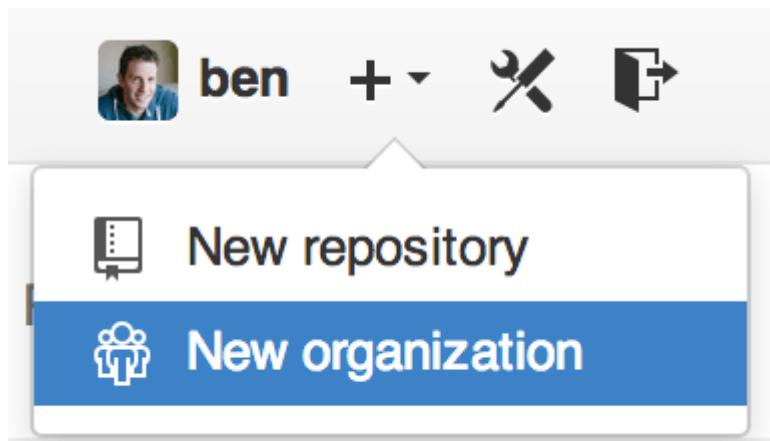
這個動作不只會把所有關注和標記星號的使用者搬到其他地方，同時也會在舊頁面建立連向新頁面的重導向連結。這也會重導向所有來自 Git 的擷取和拓製動作，不只是網頁請求而已。

Managing an organization

In addition to single-user accounts, GitHub has what are called Organizations. Like personal accounts, Organizational accounts have a namespace where all their projects exist, but many other things are different. These accounts represent a group of people with shared ownership of projects, and there are many tools to manage subgroups of those people. Normally these accounts are used for Open Source groups (such as “perl” or “rails”) or companies (such as “google” or “twitter”).

Organization Basics

An organization is pretty easy to create; just click on the “+” icon at the top-right of any GitHub page, and select “New organization” from the menu.



圖表 125. The “New organization” menu item.

First you’ll need to name your organization and provide an email address for a main point of contact for the group. Then you can invite other users to be co-owners of the account if you want to.

Follow these steps and you'll soon be the owner of a brand-new organization. Like personal accounts, organizations are free if everything you plan to store there will be open source.

As an owner in an organization, when you fork a repository, you'll have the choice of forking it to your organization's namespace. When you create new repositories you can create them either under your personal account or under any of the organizations that you are an owner in. You also automatically "watch" any new repository created under these organizations.

Just like in [你的頭像](#), you can upload an avatar for your organization to personalize it a bit. Also just like personal accounts, you have a landing page for the organization that lists all of your repositories and can be viewed by other people.

Now let's cover some of the things that are a bit different with an organizational account.

Teams

Organizations are associated with individual people by way of teams, which are simply a grouping of individual user accounts and repositories within the organization and what kind of access those people have in those repositories.

For example, say your company has three repositories: `frontend`, `backend`, and `deployscripts`. You'd want your HTML/CSS/JavaScript developers to have access to `frontend` and maybe `backend`, and your Operations people to have access to `backend` and `deployscripts`. Teams make this easy, without having to manage the collaborators for every individual repository.

The Organization page shows you a simple dashboard of all the repositories, users and teams that are under this organization.

The screenshot shows the GitHub organization page for 'chaconcorp'. On the left, there's a sidebar with a purple logo and a search bar. Below it are three repository cards: 'deployscripts' (scripts for deployment), 'backend' (Backend Code), and 'frontend' (Frontend Code). Each card has a star rating of 0, 0 reviews, and was updated 16 hours ago. To the right, there are two sections: 'People' (listing three members: dragonchacon, schacon, and tonychacon) and 'Teams' (listing three teams: Owners, Frontend Developers, and Ops). Each team section includes a 'Create new team' button.

圖表 126. The Organization page.

To manage your Teams, you can click on the Teams sidebar on the right hand side of the page in [The Organization page](#). This will bring you to a page you can use to add members to the team, add repositories to the team or manage the settings and access control levels for the team. Each team can have read only, read/write or administrative access to the repositories. You can change that level by clicking the “Settings” button in [The Team page](#).

The screenshot shows the GitHub Team page for 'Frontend Developers'. It features a sidebar with team statistics (2 members, 2 repositories) and buttons for 'Leave' and 'Settings'. The main area shows two team members: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon), each with a profile picture and a 'Remove' button. A note at the bottom states: "This team grants **Admin** access: members can read from, push to, and add collaborators to the team's repositories." Navigation tabs include 'Members' (selected), 'Repositories', and a search bar for inviting users.

圖表 127. The Team page.

When you invite someone to a team, they will get an email letting them know they've been invited.

Additionally, team **@mentions** (such as **@acmecorp/frontend**) work much the same as they do with individual users, except that **all** members of the team are then subscribed to the thread. This is useful

if you want the attention from someone on a team, but you don't know exactly who to ask.

A user can belong to any number of teams, so don't limit yourself to only access-control teams. Special-interest teams like [ux](#), [css](#), or [refactoring](#) are useful for certain kinds of questions, and others like [legal](#) and [colorblind](#) for an entirely different kind.

Audit Log

Organizations also give owners access to all the information about what went on under the organization. You can go to the *Audit Log* tab and see what events have happened at an organization level, who did them and where in the world they were done.

The screenshot shows the GitHub Audit Log interface. At the top, there's a navigation bar with the organization logo 'chaconcorp', links for 'People' (3), 'Teams' (3), and the active 'Audit log' tab. Below the navigation is a world map with red dots indicating event locations. The main area is titled 'Recent events' and lists ten entries:

| Event | User | Location | Type | Time |
|---|------------------------------|----------|-------------------------------------|----------------|
| added themselves to the chaconcorp/ops team | dragonchacon | | member | 32 minutes ago |
| added themselves to the chaconcorp/ops team | schacon | | member | 33 minutes ago |
| invited dragonchacon to the chaconcorp organization | tonychacon | | member | 16 hours ago |
| invited schacon to the chaconcorp organization | tonychacon | France | org.invite_member | 16 hours ago |
| gave chaconcorp/ops access to chaconcorp/backend | tonychacon | France | team.add_repository | 16 hours ago |
| gave chaconcorp/frontend-developers access to chaconcorp/backend | tonychacon | France | team.add_repository | 16 hours ago |
| gave chaconcorp/frontend-developers access to chaconcorp/frontend | tonychacon | France | team.add_repository | 16 hours ago |
| created the repository chaconcorp/deployscripts | tonychacon | France | repo.create | 16 hours ago |
| created the repository chaconcorp/backend | tonychacon | France | repo.create | 16 hours ago |

On the right side of the table, there's a 'Filters' dropdown menu with options: 'Yesterday's activity', 'Organization membership', 'Team management' (which is selected), 'Repository management', 'Billing updates', and 'Hook activity'. There's also a 'Search...' input field.

圖表 128. The Audit log.

You can also filter down to specific types of events, specific places or specific people.

Scripting GitHub

So now we've covered all of the major features and workflows of GitHub, but any large group or project will have customizations they may want to make or external services they may want to integrate.

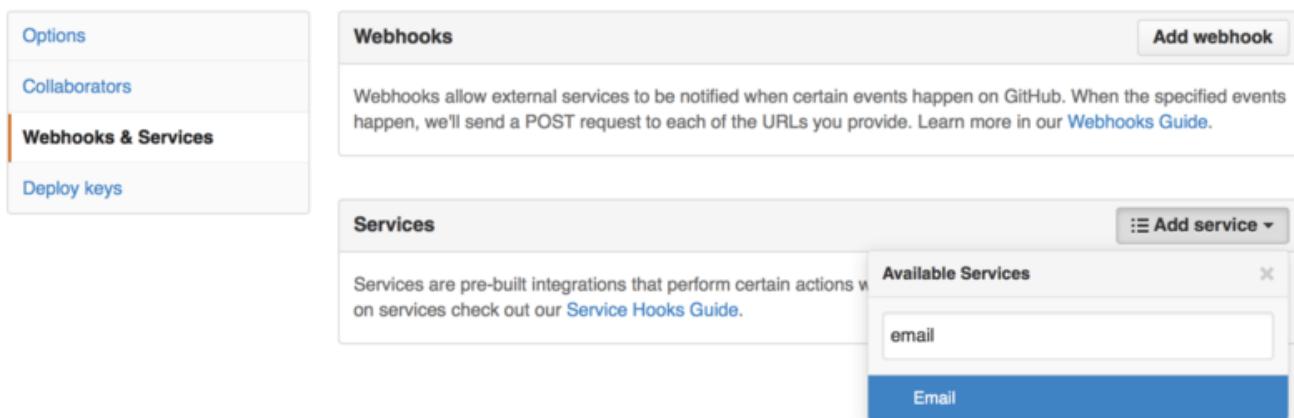
Luckily for us, GitHub is really quite hackable in many ways. In this section we'll cover how to use the GitHub hooks system and its API to make GitHub work how we want it to.

Hooks

The Hooks and Services section of GitHub repository administration is the easiest way to have GitHub interact with external systems.

Services

First we'll take a look at Services. Both the Hooks and Services integrations can be found in the Settings section of your repository, where we previously looked at adding Collaborators and changing the default branch of your project. Under the "Webhooks and Services" tab you will see something like [Services and Hooks configuration section..](#)



圖表 129. *Services and Hooks configuration section.*

There are dozens of services you can choose from, most of them integrations into other commercial and open source systems. Most of them are for Continuous Integration services, bug and issue trackers, chat room systems and documentation systems. We'll walk through setting up a very simple one, the Email hook. If you choose "email" from the "Add Service" dropdown, you'll get a configuration screen like [Email service configuration..](#)

Services / Add Email

Install Notes

- address whitespace separated email addresses (at most two)
- secret fills out the Approved header to automatically approve the message in a read-only or moderated mailing list.
- `send_from_author` uses the commit author email address in the From address of the email.

Address

Secret

Send from author

Active
We will run this service when an event is triggered.

Add service

圖表 130. Email service configuration.

In this case, if we hit the “Add service” button, the email address we specified will get an email every time someone pushes to the repository. Services can listen for lots of different types of events, but most only listen for push events and then do something with that data.

If there is a system you are using that you would like to integrate with GitHub, you should check here to see if there is an existing service integration available. For example, if you’re using Jenkins to run tests on your codebase, you can enable the Jenkins builtin service integration to kick off a test run every time someone pushes to your repository.

Hooks

If you need something more specific or you want to integrate with a service or site that is not included in this list, you can instead use the more generic hooks system. GitHub repository hooks are pretty simple. You specify a URL and GitHub will post an HTTP payload to that URL on any event you want.

Generally the way this works is you can setup a small web service to listen for a GitHub hook payload and then do something with the data when it is received.

To enable a hook, you click the “Add webhook” button in [Services and Hooks configuration section..](#) This will bring you to a page that looks like [Web hook configuration..](#)

Options

Collaborators

Webhooks & Services

Deploy keys

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

Content type

Secret

Which events would you like to trigger this webhook?

Just the push event.

Send me **everything**.

Let me select individual events.

Active
We will deliver event details when this hook is triggered.

Add webhook

圖表 131. Web hook configuration.

The configuration for a web hook is pretty simple. In most cases you simply enter a URL and a secret key and hit “Add webhook”. There are a few options for which events you want GitHub to send you a payload for—the default is to only get a payload for the **push** event, when someone pushes new code to any branch of your repository.

Let’s see a small example of a web service you may set up to handle a web hook. We’ll use the Ruby web framework Sinatra since it’s fairly concise and you should be able to easily see what we’re doing.

Let’s say we want to get an email if a specific person pushes to a specific branch of our project modifying a specific file. We could fairly easily do that with code like this:

```

require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
    branch == 'ref/heads/special-branch' &&
    files.include?('special-file.txt')

    Mail.deliver do
      from      'tchacon@example.com'
      to        'tchacon@example.com'
      subject   'Scott Changed the File'
      body     "ALARM"
    end
  end
end

```

Here we’re taking the JSON payload that GitHub delivers us and looking up who pushed it, what branch they pushed to and what files were touched in all the commits that were pushed. Then we check that against our criteria and send an email if it matches.

In order to develop and test something like this, you have a nice developer console in the same screen where you set the hook up. You can see the last few deliveries that GitHub has tried to make for that webhook. For each hook you can dig down into when it was delivered, if it was successful and the body and headers for both the request and the response. This makes it incredibly easy to test and debug your hooks.

Recent Deliveries

| | | | |
|--------------------------------------|--------------------------------------|---------------------|-----|
| ⚠ | 4aeae280-4e38-11e4-9bac-c130e992644b | 2014-10-07 17:40:41 | ... |
| ✓ | aff20880-4e37-11e4-9089-35319435e08b | 2014-10-07 17:36:21 | ... |
| ✓ | 90f37680-4e37-11e4-9508-227d13b2ccfc | 2014-10-07 17:35:29 | ... |

Request Response 200

🕒 Completed in 0.61 seconds. ↻ Redeliver

Headers

```
Request URL: https://hooks.example.com/payload
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-GitHub-Event: push
```

Payload

```
{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bfffaf827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bfffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
    }
  ]
}
```

图表 132. Web hook debugging information.

The other great feature of this is that you can redeliver any of the payloads to test your service easily.

For more information on how to write webhooks and all the different event types you can listen for, go to the GitHub Developer documentation at <https://developer.github.com/webhooks/>

The GitHub API

Services and hooks give you a way to receive push notifications about events that happen on your repositories, but what if you need more information about these events? What if you need to automate something like adding collaborators or labeling issues?

This is where the GitHub API comes in handy. GitHub has tons of API endpoints for doing nearly anything you can do on the website in an automated fashion. In this section we'll learn how to

authenticate and connect to the API, how to comment on an issue and how to change the status of a Pull Request through the API.

Basic Usage

The most basic thing you can do is a simple GET request on an endpoint that doesn't require authentication. This could be a user or read-only information on an open source project. For example, if we want to know more about a user named "schacon", we can run something like this:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
# ...
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

There are tons of endpoints like this to get information about organizations, projects, issues, commits—just about anything you can publicly see on GitHub. You can even use the API to render arbitrary Markdown or find a [.gitignore](#) template.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
```

Commenting on an Issue

However, if you want to do an action on the website such as comment on an Issue or Pull Request or if you want to view or interact with private content, you'll need to authenticate.

There are several ways to authenticate. You can use basic authentication with just your username and password, but generally it's a better idea to use a personal access token. You can generate this from the “Applications” tab of your settings page.

The screenshot shows the GitHub user interface for managing applications. On the left, a sidebar lists profile, account settings, emails, notification center, billing, SSH keys, security, and applications. The 'Applications' section is selected, indicated by an orange border. Below the sidebar, there are two main sections: 'Developer applications' and 'Personal access tokens'. The 'Developer applications' section contains a link to register a new application and a note about generating OAuth tokens. The 'Personal access tokens' section includes a 'Generate new token' button and a note about using tokens for scripts or testing, with a link to generate a token. The 'Authorized applications' section notes that no applications are authorized. The 'GitHub applications' section lists 'GitHub Team' with a 'Last used on Oct 6, 2014' timestamp and a 'Revoke' button.

圖表 133. Generate your access token from the “Applications” tab of your settings page.

It will ask you which scopes you want for this token and a description. Make sure to use a good description so you feel comfortable removing the token when your script or application is no longer used.

GitHub will only show you the token once, so be sure to copy it. You can now use this to authenticate in your script instead of using a username and password. This is nice because you can limit the scope of what you want to do and the token is revocable.

This also has the added advantage of increasing your rate limit. Without authenticating, you will be limited to 60 requests per hour. If you authenticate you can make up to 5,000 requests per hour.

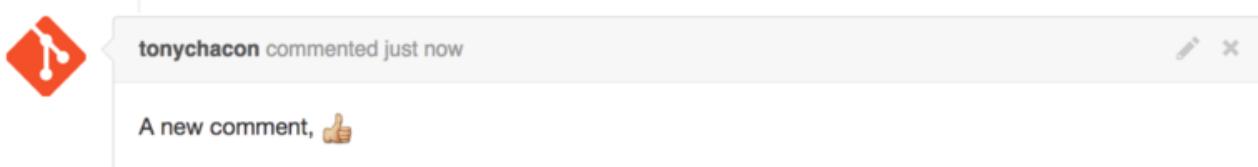
So let's use it to make a comment on one of our issues. Let's say we want to leave a comment on a specific issue, Issue #6. To do so we have to do an HTTP POST request to `repos/<user>/<repo>/issues/<num>/comments` with the token we just generated as an Authorization header.

```

$ curl -H "Content-Type: application/json" \
      -H "Authorization: token TOKEN" \
      --data '{"body":"A new comment, :+1:"}' \
      https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}

```

Now if you go to that issue, you can see the comment that we just successfully posted as in [A comment posted from the GitHub API..](#)



圖表 134. *A comment posted from the GitHub API.*

You can use the API to do just about anything you can do on the website—creating and setting milestones, assigning people to Issues and Pull Requests, creating and changing labels, accessing commit data, creating new commits and branches, opening, closing or merging Pull Requests, creating and editing teams, commenting on lines of code in a Pull Request, searching the site and on and on.

Changing the Status of a Pull Request

There is one final example we'll look at since it's really useful if you're working with Pull Requests. Each commit can have one or more statuses associated with it and there is an API to add and query that status.

Most of the Continuous Integration and testing services make use of this API to react to pushes by testing the code that was pushed, and then report back if that commit has passed all the tests. You could also use this to check if the commit message is properly formatted, if the submitter followed all your contribution guidelines, if the commit was validly signed—any number of things.

Let's say you set up a webhook on your repository that hits a small web service that checks for a **Signed-off-by** string in the commit message.

```

require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url =
"https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"      => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" })
  end
end

```

Hopefully this is fairly simple to follow. In this web hook handler we look through each commit that was just pushed, we look for the string *Signed-off-by* in the commit message and finally we POST via HTTP to the [`/repos/<user>/<repo>/statuses/<commit_sha>`](#) API endpoint with the status.

In this case you can send a state (*success, failure, error*), a description of what happened, a target URL the user can go to for more information and a “context” in case there are multiple statuses for a single commit. For example, a testing service may provide a status and a validation service like this may also provide a status—~~the “context” field is how they’re differentiated.~~

If someone opens a new Pull Request on GitHub and this hook is set up, you may see something like

Commit status via the API..

The screenshot shows a GitHub pull request interface. At the top, a comment from user 'schacon' is displayed, stating 'Removing whitespace in the files.' Below this, another comment from 'schacon' shows two commits: one marked as 'properly signed off' with a green checkmark and commit hash '9f40fd5', and another marked as 'forgot to sign off' with a red cross and commit hash 'ee7aa38'. A note below the commits says 'Add more commits by pushing to the `remove-whitespace` branch on `tonychacon/fade`'. A yellow warning box at the bottom left indicates a failure with the message 'Failed — No signoff found.' It also includes a 'Merge with caution!' note and a link to merge the pull request.

圖表 135. Commit status via the API.

You can now see a little green check mark next to the commit that has a “Signed-off-by” string in the message and a red cross through the one where the author forgot to sign off. You can also see that the Pull Request takes the status of the last commit on the branch and warns you if it is a failure. This is really useful if you’re using this API for test results so you don’t accidentally merge something where the last commit is failing tests.

Octokit

Though we’ve been doing nearly everything through `curl` and simple HTTP requests in these examples, several open-source libraries exist that make this API available in a more idiomatic way. At the time of this writing, the supported languages include Go, Objective-C, Ruby, and .NET. Check out <http://github.com/octokit> for more information on these, as they handle much of the HTTP for you.

Hopefully these tools can help you customize and modify GitHub to work better for your specific workflows. For complete documentation on the entire API as well as guides for common tasks, check out <https://developer.github.com>.

總結

現在你是個 GitHub 使用者了。你現在知道如何建立帳號、管理組織、建立及推送變更到倉儲、參與別人的專案或是接受別人對你專案的變更。在下個章節，你會學到更加強力的工具及技巧來應對複雜的處境，這會讓你變成真的 Git 專家。

Git Tools

By now, you've learned most of the day-to-day commands and workflows that you need to manage or maintain a Git repository for your source code control. You've accomplished the basic tasks of tracking and committing files, and you've harnessed the power of the staging area and lightweight topic branching and merging.

Now you'll explore a number of very powerful things that Git can do that you may not necessarily use on a day-to-day basis but that you may need at some point.

Revision Selection

Git allows you to specify specific commits or a range of commits in several ways. They aren't necessarily obvious but are helpful to know.

Single Revisions

You can obviously refer to a commit by the SHA-1 hash that it's given, but there are more human-friendly ways to refer to commits as well. This section outlines the various ways you can refer to a single commit.

Short SHA-1

Git is smart enough to figure out what commit you meant to type if you provide the first few characters, as long as your partial SHA-1 is at least four characters long and unambiguous – that is, only one object in the current repository begins with that partial SHA-1.

For example, to see a specific commit, suppose you run a `git log` command and identify the commit where you added certain functionality:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

In this case, choose `1c002dd...`. If you `git show` that commit, the following commands are equivalent (assuming the shorter versions are unambiguous):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b  
$ git show 1c002dd4b536e7479f  
$ git show 1c002d
```

Git can figure out a short, unique abbreviation for your SHA-1 values. If you pass `--abbrev-commit` to the `git log` command, the output will use shorter values but keep them unique; it defaults to using seven characters but makes them longer if necessary to keep the SHA-1 unambiguous:

```
$ git log --abbrev-commit --pretty=oneline  
ca82a6d changed the version number  
085bb3b removed unnecessary test code  
a11bef0 first commit
```

Generally, eight to ten characters are more than enough to be unique within a project.

As an example, the Linux kernel, which is a pretty large project with over 450k commits and 3.6 million objects, has no two objects whose SHA-1s overlap more than the first 11 characters.

A SHORT NOTE ABOUT SHA-1

A lot of people become concerned at some point that they will, by random happenstance, have two objects in their repository that hash to the same SHA-1 value. What then?

If you do happen to commit an object that hashes to the same SHA-1 value as a previous object in your repository, Git will see the previous object already in your Git database and assume it was already written. If you try to check out that object again at some point, you'll always get the data of the first object.

However, you should be aware of how ridiculously unlikely this scenario is. The SHA-1 digest is 20 bytes or 160 bits. The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is $p = (n(n-1)/2) * (1/2^{160})$). 2^{80} is 1.2×10^{24} or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.

Here's an example to give you an idea of what it would take to get a SHA-1 collision. If all 6.5 billion humans on Earth were programming, and every second, each one was producing code that was the equivalent of the entire Linux kernel history (3.6 million Git objects) and pushing it into one enormous Git repository, it would take roughly 2 years until that repository contained enough objects to have a 50% probability of a single SHA-1 object collision. A higher probability exists that every member of your programming team will be attacked and killed by wolves in unrelated incidents on the same night.

筆記

Branch References

The most straightforward way to specify a commit requires that it has a branch reference pointed at it. Then, you can use a branch name in any Git command that expects a commit object or SHA-1 value. For instance, if you want to show the last commit object on a branch, the following commands are equivalent, assuming that the `topic1` branch points to `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949  
$ git show topic1
```

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called `rev-parse`. You can see [Git Internals](#) for more information about plumbing tools; basically, `rev-parse` exists for lower-level operations and isn't designed to be used in day-to-day operations. However, it can be helpful sometimes when you need to see what's really going on. Here you can run `rev-parse` on your branch.

```
$ git rev-parse topic1  
ca82a6dff817ec66f44342007202690a93763949
```

RefLog Shortnames

One of the things Git does in the background while you're working away is keep a “reflog” – a log of where your HEAD and branch references have been for the last few months.

You can see your reflog by using `git reflog`:

```
$ git reflog  
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated  
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.  
1c002dd HEAD@{2}: commit: added some blame and merge stuff  
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD  
95df984 HEAD@{4}: commit: # This is a combination of two commits.  
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD  
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Every time your branch tip is updated for any reason, Git stores that information for you in this temporary history. And you can specify older commits with this data, as well. If you want to see the fifth prior value of the HEAD of your repository, you can use the `@{n}` reference that you see in the reflog output:

```
$ git show HEAD@{5}
```

You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to see where your `master` branch was yesterday, you can type

```
$ git show master@{yesterday}
```

That shows you where the branch tip was yesterday. This technique only works for data that's still in your reflog, so you can't use it to look for commits older than a few months.

To see reflog information formatted like the `git log` output, you can run `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

        fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

It's important to note that the reflog information is strictly local – it's a log of what you've done in your repository. The references won't be the same on someone else's copy of the repository; and right after you initially clone a repository, you'll have an empty reflog, as no activity has occurred yet in your repository. Running `git show HEAD@{2.months.ago}` will work only if you cloned the project at least two months ago – if you cloned it five minutes ago, you'll get no results.

Ancestry References

The other main way to specify a commit is via its ancestry. If you place a `^` at the end of a reference, Git resolves it to mean the parent of that commit. Suppose you look at the history of your project:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\ 
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Then, you can see the previous commit by specifying `HEAD^`, which means “the parent of HEAD” :

```
$ git show HEAD^  
commit d921970aadf03b3cf0e71becdaab3147ba71cdef  
Merge: 1c002dd... 35cfb2b...  
Author: Scott Chacon <schacon@gmail.com>  
Date: Thu Dec 11 15:08:43 2008 -0800  
  
        Merge commit 'phedders/rdocs'
```

You can also specify a number after the `^` – for example, `d921970^2` means “the second parent of `d921970`.” This syntax is only useful for merge commits, which have more than one parent. The first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in:

```
$ git show d921970^  
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b  
Author: Scott Chacon <schacon@gmail.com>  
Date: Thu Dec 11 14:58:32 2008 -0800  
  
        added some blame and merge stuff  
  
$ git show d921970^2  
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548  
Author: Paul Hedderly <paul+git@mjr.org>  
Date: Wed Dec 10 22:22:03 2008 +0000  
  
        Some rdoc changes
```

The other main ancestry specification is the `~`. This also refers to the first parent, so `HEAD~` and `HEAD^` are equivalent. The difference becomes apparent when you specify a number. `HEAD~2` means “the first parent of the first parent,” or “the grandparent” – it traverses the first parents the number of times you specify. For example, in the history listed earlier, `HEAD~3` would be

```
$ git show HEAD~3  
commit 1c3618887afb5fbcbeca25b7c013f4e2114448b8d  
Author: Tom Preston-Werner <tom@mojombo.com>  
Date: Fri Nov 7 13:47:59 2008 -0500  
  
        ignore *.gem
```

This can also be written `HEAD^{~^}`, which again is the first parent of the first parent of the first parent:

```
$ git show HEAD^^^  
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d  
Author: Tom Preston-Werner <tom@mojombo.com>  
Date: Fri Nov 7 13:47:59 2008 -0500  
  
ignore *.gem
```

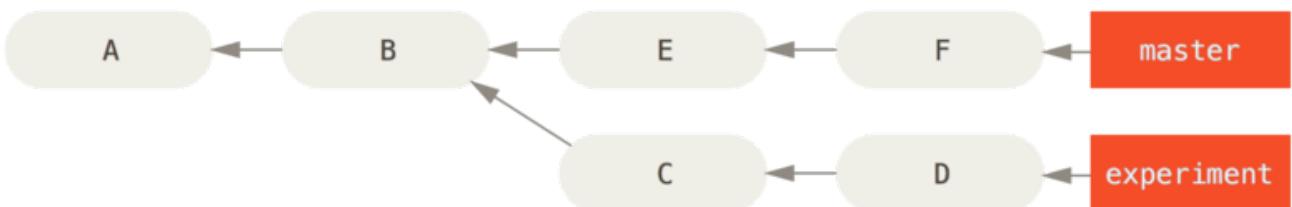
You can also combine these syntaxes – you can get the second parent of the previous reference (assuming it was a merge commit) by using `HEAD~3^2`, and so on.

Commit Ranges

Now that you can specify individual commits, let's see how to specify ranges of commits. This is particularly useful for managing your branches – if you have a lot of branches, you can use range specifications to answer questions such as, “What work is on this branch that I haven’t yet merged into my main branch?”

Double Dot

The most common range specification is the double-dot syntax. This basically asks Git to resolve a range of commits that are reachable from one commit but aren't reachable from another. For example, say you have a commit history that looks like [Example history for range selection..](#)



图表 136. Example history for range selection.

You want to see what is in your experiment branch that hasn't yet been merged into your master branch. You can ask Git to show you a log of just those commits with `master..experiment` – that means “all commits reachable by experiment that aren't reachable by master.” For the sake of brevity and clarity in these examples, I'll use the letters of the commit objects from the diagram in place of the actual log output in the order that they would display:

```
$ git log master..experiment  
D  
C
```

If, on the other hand, you want to see the opposite – all commits in `master` that aren't in `experiment` – you can reverse the branch names. `experiment..master` shows you everything in `master` not reachable from `experiment`:

```
$ git log experiment..master  
F  
E
```

This is useful if you want to keep the `experiment` branch up to date and preview what you’re about to merge in. Another very frequent use of this syntax is to see what you’re about to push to a remote:

```
$ git log origin/master..HEAD
```

This command shows you any commits in your current branch that aren’t in the `master` branch on your `origin` remote. If you run a `git push` and your current branch is tracking `origin/master`, the commits listed by `git log origin/master..HEAD` are the commits that will be transferred to the server. You can also leave off one side of the syntax to have Git assume HEAD. For example, you can get the same results as in the previous example by typing `git log origin/master..` – Git substitutes HEAD if one side is missing.

Multiple Points

The double-dot syntax is useful as a shorthand; but perhaps you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of several branches that aren’t in the branch you’re currently on. Git allows you to do this by using either the `^` character or `--not` before any reference from which you don’t want to see reachable commits. Thus these three commands are equivalent:

```
$ git log refA..refB  
$ git log ^refA refB  
$ git log refB --not refA
```

This is nice because with this syntax you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from `refA` or `refB` but not from `refC`, you can type one of these:

```
$ git log refA refB ^refC  
$ git log refA refB --not refC
```

This makes for a very powerful revision query system that should help you figure out what is in your branches.

Triple Dot

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by either of two references but not by both of them. Look back at the example commit history in [Example history for range selection..](#). If you want to see what is in `master` or `experiment` but not any common references, you can run

```
$ git log master...experiment  
F  
E  
D  
C
```

Again, this gives you normal `log` output but shows you only the commit information for those four commits, appearing in the traditional commit date ordering.

A common switch to use with the `log` command in this case is `--left-right`, which shows you which side of the range each commit is in. This helps make the data more useful:

```
$ git log --left-right master...experiment  
< F  
< E  
> D  
> C
```

With these tools, you can much more easily let Git know what commit or commits you want to inspect.

Interactive Staging

Git comes with a couple of scripts that make some command-line tasks easier. Here, you'll look at a few interactive commands that can help you easily craft your commits to include only certain combinations and parts of files. These tools are very helpful if you modify a bunch of files and then decide that you want those changes to be in several focused commits rather than one big messy commit. This way, you can make sure your commits are logically separate changesets and can be easily reviewed by the developers working with you. If you run `git add` with the `-i` or `--interactive` option, Git goes into an interactive shell mode, displaying something like this:

```
$ git add -i  
      staged      unstaged path  
1: unchanged      +0/-1 TODO  
2: unchanged      +1/-1 index.html  
3: unchanged      +5/-1 lib/simplegit.rb  
  
*** Commands ***  
 1: status       2: update       3: revert       4: add untracked  
 5: patch       6: diff          7: quit         8: help  
What now>
```

You can see that this command shows you a much different view of your staging area – basically the same information you get with `git status` but a bit more succinct and informative. It lists the changes you've staged on the left and unstaged changes on the right.

After this comes a Commands section. Here you can do a number of things, including staging files, unstaging files, staging parts of files, adding untracked files, and seeing diffs of what has been staged.

Staging and Unstaging Files

If you type **2** or **u** at the **What now>** prompt, the script prompts you for which files you want to stage:

```
What now> 2
      staged      unstaged path
 1:  unchanged      +0/-1 TODO
 2:  unchanged      +1/-1 index.html
 3:  unchanged      +5/-1 lib/simplegit.rb
Update>>
```

To stage the TODO and index.html files, you can type the numbers:

```
Update>> 1,2
      staged      unstaged path
* 1:  unchanged      +0/-1 TODO
* 2:  unchanged      +1/-1 index.html
 3:  unchanged      +5/-1 lib/simplegit.rb
Update>>
```

The ***** next to each file means the file is selected to be staged. If you press Enter after typing nothing at the **Update>>** prompt, Git takes anything selected and stages it for you:

```
Update>>
updated 2 paths

*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit        8: help
What now> 1
      staged      unstaged path
 1:      +0/-1      nothing TODO
 2:      +1/-1      nothing index.html
 3:  unchanged      +5/-1 lib/simplegit.rb
```

Now you can see that the TODO and index.html files are staged and the simplegit.rb file is still unstaged. If you want to unstage the TODO file at this point, you use the **3** or **r** (for revert) option:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 3
          staged      unstaged path
1:          +0/-1      nothing TODO
2:          +1/-1      nothing index.html
3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> 1
          staged      unstaged path
* 1:          +0/-1      nothing TODO
2:          +1/-1      nothing index.html
3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path

```

Looking at your Git status again, you can see that you've unstaged the TODO file:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 1
          staged      unstaged path
1:      unchanged      +0/-1 TODO
2:          +1/-1      nothing index.html
3:      unchanged      +5/-1 lib/simplegit.rb

```

To see the diff of what you've staged, you can use the **6** or **d** (for diff) command. It shows you a list of your staged files, and you can select the ones for which you would like to see the staged diff. This is much like specifying **git diff --cached** on the command line:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch       6: diff        7: quit        8: help
What now> 6
          staged      unstaged path
1:           +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

With these basic commands, you can use the interactive add mode to deal with your staging area a little more easily.

Staging Patches

It's also possible for Git to stage certain parts of files and not the rest. For example, if you make two changes to your simplegit.rb file and want to stage one of them and not the other, doing so is very easy in Git. From the interactive prompt, type **5** or **p** (for patch). Git will ask you which files you would like to partially stage; then, for each section of the selected files, it will display hunks of the file diff and ask if you would like to stage them, one by one:

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
  end

  def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
  end

  def blame(path)
Stage this hunk [y,n,a,d/,j,J,g,e,?]?

```

You have a lot of options at this point. Typing **?** shows a list of what you can do:

```
Stage this hunk [y,n,a,d,/,,j,J,g,e,?] ? 
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Generally, you'll type `y` or `n` if you want to stage each hunk, but staging all of them in certain files or skipping a hunk decision until later can be helpful too. If you stage one part of the file and leave another part unstaged, your status output will look like this:

```
What now> 1
      staged      unstaged path
1:   unchanged      +0/-1 TODO
2:          +1/-1    nothing index.html
3:          +1/-1    +4/-0 lib/simplegit.rb
```

The status of the `simplegit.rb` file is interesting. It shows you that a couple of lines are staged and a couple are unstaged. You've partially staged this file. At this point, you can exit the interactive adding script and run `git commit` to commit the partially staged files.

You also don't need to be in interactive add mode to do the partial-file staging – you can start the same script by using `git add -p` or `git add --patch` on the command line.

Furthermore, you can use patch mode for partially resetting files with the `reset --patch` command, for checking out parts of files with the `checkout --patch` command and for stashing parts of files with the `stash save --patch` command. We'll go into more details on each of these as we get to more advanced usages of these commands.

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Stashing Your Work

To demonstrate, you'll go into your project and start working on a couple of files and possibly stage one of the changes. If you run `git status`, you can see your dirty state:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

    modified:   lib/simplegit.rb
```

Now you want to switch branches, but you don't want to commit what you've been working on yet; so you'll stash the changes. To push a new stash onto your stack, run `git stash` or `git stash save`:

```
$ git stash
Saved working directory and index state \
"WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Your working directory is clean:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

At this point, you can easily switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, two stashes were done previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. If you want to apply one of the older stashes, you can specify it by naming it, like this: `git stash apply stash@{2}`. If you don't specify a stash, Git

assumes the most recent stash and tries to apply it:

```
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

    modified:   index.html
    modified:   lib/simplegit.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

You can see that Git re-modifies the files you reverted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from; but having a clean working directory and applying it on the same branch aren't necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash – Git gives you merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but the file you staged before wasn't restaged. To do that, you must run the `git stash apply` command with a `--index` option to tell the command to try to reapply the staged changes. If you had run that instead, you'd have gotten back to your original position:

```
$ git stash apply --index
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

    modified:   lib/simplegit.rb
```

The apply option only tries to apply the stashed work – you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the `--keep-index` option to the `stash save` command. This tells Git to not stash anything that you've already staged with the `git add` command.

This can be really helpful if you've made a number of changes but want to only commit some of them and then come back to the rest of the changes at a later time.

```
$ git status -s
M index.html
M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the
index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M index.html
```

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will only store files that are already in the index. If you specify `--include-untracked` or `-u`, Git will also stash any untracked files you have created.

```
$ git status -s
M index.html
M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the
index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Finally, if you specify the `--patch` flag, Git will not stash everything that is modified but will instead

prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
    return `#{git_cmd} 2>&1`.chomp
  end
end

+
+  def show(treeish = 'master')
+    command("git show #{treeish}")
+  end

end
test
Stash this hunk [y,n,q,a,d,/ ,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the
index file
```

Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you've since modified, you'll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run `git stash branch`, which creates a new branch for you, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

```
$ git stash branch testchanges
M  index.html
M  lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

Cleaning your Working Directory

Finally, you may not want to stash some work or files in your working directory, but simply get rid of them. The `git clean` command will do this for you.

Some common reasons for this might be to remove cruft that has been generated by merges or external tools or to remove build artifacts in order to run a clean build.

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with `git clean`. To remove all the untracked files in your working directory, you can run `git clean -f -d`, which removes any files and also any subdirectories that become empty as a result. The `-f` means *force* or "really do this".

If you ever want to see what it would do, you can run the command with the `-n` option, which means "do a dry run and tell me what you *would* have removed".

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

By default, the `git clean` command will only remove untracked files that are not ignored. Any file that matches a pattern in your `.gitignore` or other ignore files will not be removed. If you want to remove those files too, such as to remove all `.o` files generated from a build so you can do a fully clean

build, you can add a `-x` to the clean command.

```
$ git status -s
M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

If you don't know what the `git clean` command is going to do, always run it with a `-n` first to double check before changing the `-n` to a `-f` and doing it for real. The other way you can be careful about the process is to run it with the `-i` or "interactive" flag.

This will run the clean command in an interactive mode.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
  1: clean          2: filter by pattern   3: select by numbers
  4: ask each       5: quit
  6: help
What now>
```

This way you can step through each file individually or specify patterns for deletion interactively.

Signing Your Work

Git is cryptographically secure, but it's not foolproof. If you're taking work from others on the internet and want to verify that commits are actually from a trusted source, Git has a few ways to sign and verify work using GPG.

GPG Introduction

First of all, if you want to sign anything you need to get GPG configured and your personal key installed.

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg  
-----  
pub 2048R/0A46826A 2014-06-04  
uid Scott Chacon (Git signing key) <schacon@gmail.com>  
sub 2048R/874529A9 2014-06-04
```

If you don't have a key installed, you can generate one with `gpg --gen-key`.

```
gpg --gen-key
```

Once you have a private key to sign with, you can configure Git to use it for signing things by setting the `user.signingkey` config setting.

```
git config --global user.signingkey 0A46826A
```

Now Git will use your key by default to sign tags and commits if you want.

Signing Tags

If you have a GPG private key setup, you can now use it to sign new tags. All you have to do is use `-s` instead of `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
```

```
You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

If you run `git show` on that tag, you can see your GPG signature attached to it:

```

$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQ1AAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kC3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfcimnSDeSvzCpWAHl7h8Wj6hhqePmLm91AYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1Pb1GfHR4XAhu0qN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

changed the version number

```

Verifying Tags

To verify a signed tag, you use `git tag -v [tag-name]`. This command uses GPG to verify the signature. You need the signer's public key in your keyring for this to work properly:

```

$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID
F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                               aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311
9B9A

```

If you don't have the signer's public key, you get something like this instead:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID  
F3119B9A  
gpg: Can't check signature: public key not found  
error: could not verify the tag 'v1.4.2.1'
```

Signing Commits

In more recent versions of Git (v1.7.9 and above), you can now also sign individual commits. If you're interested in signing commits directly instead of just the tags, all you need to do is add a `-S` to your `git commit` command.

```
$ git commit -a -S -m 'signed commit'  
  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"  
2048-bit RSA key, ID 0A46826A, created 2014-06-04  
  
[master 5c3386c] signed commit  
 4 files changed, 4 insertions(+), 24 deletions(-)  
 rewrite Rakefile (100%)  
 create mode 100644 lib/git.rb
```

To see and verify these signatures, there is also a `--show-signature` option to `git log`.

```
$ git log --show-signature -1  
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2  
gpg: Signature made Wed Jun 4 19:49:17 2014 PDT using RSA key ID  
0A46826A  
gpg: Good signature from "Scott Chacon (Git signing key)  
<schacon@gmail.com>"  
Author: Scott Chacon <schacon@gmail.com>  
Date:   Wed Jun 4 19:49:17 2014 -0700  
  
        signed commit
```

Additionally, you can configure `git log` to check any signatures it finds and list them in its output with the `%G?` format.

```
$ git log --pretty="format:%h %G? %aN  %s"  
5c3386c G Scott Chacon  signed commit  
ca82a6d N Scott Chacon  changed the version number  
085bb3b N Scott Chacon  removed unnecessary test code  
a11bef0 N Scott Chacon  first commit
```

Here we can see that only the latest commit is signed and valid and the previous commits are not.

In Git 1.8.3 and later, "git merge" and "git pull" can be told to inspect and reject when merging a commit that does not carry a trusted GPG signature with the `--verify-signatures` command.

If you use this option when merging a branch and it contains commits that are not signed and valid, the merge will not work.

```
$ git merge --verify-signatures non-verify  
fatal: Commit ab06180 does not have a GPG signature.
```

If the merge contains only valid signed commits, the merge command will show you all the signatures it has checked and then move forward with the merge.

```
$ git merge --verify-signatures signed-branch  
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing  
key) <schacon@gmail.com>  
Updating 5c3386c..13ad65e  
Fast-forward  
 README | 2 ++  
 1 file changed, 2 insertions(+)
```

You can also use the `-S` option with the `git merge` command itself to sign the resulting merge commit itself. The following example both verifies that every commit in the branch to be merged is signed and furthermore signs the resulting merge commit.

```
$ git merge --verify-signatures -S signed-branch  
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing  
key) <schacon@gmail.com>  
  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"  
2048-bit RSA key, ID 0A46826A, created 2014-06-04  
  
Merge made by the 'recursive' strategy.  
 README | 2 ++  
 1 file changed, 2 insertions(+)
```

Everyone Must Sign

Signing tags and commits is great, but if you decide to use this in your normal workflow, you'll have to make sure that everyone on your team understands how to do so. If you don't, you'll end up spending a lot of time helping people figure out how to rewrite their commits with signed versions. Make sure you understand GPG and the benefits of signing things before adopting this as part of your standard workflow.

Searching

With just about any size codebase, you'll often need to find where a function is called or defined, or find the history of a method. Git provides a couple of useful tools for looking through the code and

commits stored in its database quickly and easily. We'll go through a few of them.

Git Grep

Git ships with a command called `grep` that allows you to easily search through any committed tree or the working directory for a string or regular expression. For these examples, we'll look through the Git source code itself.

By default, it will look through the files in your working directory. You can pass `-n` to print out the line numbers where Git has found matches.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:    return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct
tm *result)
compat/gmtime.c:16:    ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm
*result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm
*result);
date.c:429:            if (gmtime_r(&now, &now_tm))
date.c:492:            if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm
*);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

There are a number of interesting options you can provide the `grep` command.

For instance, instead of the previous call, you can have Git summarize the output by just showing you which files matched and how many matches there were in each file with the `--count` option:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

If you want to see what method or function it thinks it has found a match in, you can pass `-p`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const
char *date, char *end, struct tm *tm)
date.c:            if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int
*offset, int *tm_gmt)
date.c:            if (gmtime_r(&time, tm)) {
```

So here we can see that `gmtime_r` is called in the `match_multi_number` and `match_digit` functions in the `date.c` file.

You can also look for complex combinations of strings with the `--and` flag, which makes sure that multiple matches are in the same line. For instance, let's look for any lines that define a constant with either the strings "LINK" or "BUF_MAX" in them in the Git codebase in an older 1.8.0 version.

Here we'll also use the `--break` and `--heading` options which help split up the output into a more readable format.

```
$ git grep --break --heading \
-n -e '#define' --and \(-e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m) (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATIONUSES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

The `git grep` command has a few advantages over normal searching commands like `grep` and `ack`. The first is that it's really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

Git Log Searching

Perhaps you're looking not for **where** a term exists, but **when** it existed or was introduced. The `git log` command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If we want to find out for example when the `ZLIB_BUF_MAX` constant was originally introduced, we can tell Git to only show us the commits that either added or removed that string with the `-S` option.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

If we look at the diff of those commits we can see that in [ef49a7a](#) the constant was introduced and in [e01503b](#) it was modified.

If you need to be more specific, you can provide a regular expression to search for with the `-G` option.

Line Log Search

Another fairly advanced log search that is insanely useful is the line history search. This is a fairly recent addition and not very well known, but it can be really helpful. It is called with the `-L` option to `git log` and will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function `git_deflate_bound` in the `zlib.c` file, we could run `git log -L :git_deflate_bound:zlib.c`. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
{
-    return deflateBound(strm, size);
+    return deflateBound(&strm->z, size);
}

commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+    return deflateBound(strm, size);
+}
+
```

If Git can't figure out how to match a function or method in your programming language, you can also provide it a regex. For example, this would have done the same thing: `git log -L '/unsigned long git_deflate_bound/ ,/^}/:zlib.c`. You could also give it a range of lines or a single line number and you'll get the same sort of output.

Rewriting History

Many times, when working with Git, you may want to revise your commit history for some reason. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn't mean to be working on something yet with the stash command, and you can rewrite commits that already happened so they look like they happened in a different way. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely – all before you share your work with others.

In this section, you'll cover how to accomplish these very useful tasks so that you can make your commit history look the way you want before you share it with others.

Changing the Last Commit

Changing your last commit is probably the most common rewriting of history that you'll do. You'll often want to do two basic things to your last commit: change the commit message, or change the snapshot you just recorded by adding, changing and removing files.

If you only want to modify your last commit message, it's very simple:

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

That drops you into your text editor, which has your last commit message in it, ready for you to modify the message. When you save and close the editor, the editor writes a new commit containing that message and makes it your new last commit.

If you've committed and then you want to change the snapshot you committed by adding or changing files, possibly because you forgot to add a newly created file when you originally committed, the process works basically the same way. You stage the changes you want by editing a file and running `git add` on it or `git rm` to a tracked file, and the subsequent `git commit --amend` takes your current staging area and makes it the snapshot for the new commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It's like a very small rebase – don't amend your last commit if you've already pushed it.

Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the `-i` option to `git rebase`. You must indicate how far back you want to rewrite commits by telling the command

which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to `git rebase -i` the parent of the last commit you want to edit, which is `HEAD~2^` or `HEAD~3`. It may be easier to remember the `~3` because you’re trying to edit the last three commits; but keep in mind that you’re actually designating four commits ago, the parent of the last commit you want to edit:

```
$ git rebase -i HEAD~3
```

Remember again that this is a rebasing command – every commit included in the range `HEAD~3..HEAD` will be rewritten, whether you change the message or not. Don’t include any commit you’ve already pushed to a central server – doing so will confuse other developers by providing an alternate version of the same change.

Running this command gives you a list of commits in your text editor that looks something like this:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

It’s important to note that these commits are listed in the opposite order than you normally see them using the `log` command. If you run a `log`, you see something like this:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Notice the reverse order. The interactive rebase gives you a script that it’s going to run. It will start at

the commit you specify on the command line (`HEAD~3`) and replay the changes introduced in each of these commits from top to bottom. It lists the oldest at the top, rather than the newest, because that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word ‘pick’ to the word ‘edit’ for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

These instructions tell you exactly what to do. Type

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

Change the commit message, and exit the editor. Then, run

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

This command will apply the other two commits automatically, and then you’re done. If you change pick to edit on more lines, you can repeat these steps for each commit you change to edit. Each time, Git will stop, let you amend the commit, and continue when you’re finished.

Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the “added cat-file” commit and change the order in which the other two commits are introduced, you can change the rebase script from this

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

to this:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies **310154e** and then **f7f3f6d**, and then stops. You effectively change the order of those commits and remove the “added cat-file” commit completely.

Squashing Commits

It’s also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

```
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

If, instead of “pick” or “edit”, you specify “squash”, Git applies both that change and the change directly before it and makes you merge the commit messages together. So, if you want to make a single commit from these three commits, you make the script look like this:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

```
# This is a combination of 3 commits.  
# The first commit's message is:  
changed my name a bit  
  
# This is the 2nd commit message:  
  
updated README formatting and added blame  
  
# This is the 3rd commit message:  
  
added cat-file
```

When you save that, you have a single commit that introduces the changes of all three previous commits.

Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of “updated README formatting and added blame”, you want to split it into two commits: “updated README formatting” for the first, and “added blame” for the second. You can do that in the `rebase -i` script by changing the instruction on the commit you want to split to “edit” :

```
pick f7f3f6d changed my name a bit  
edit 310154e updated README formatting and added blame  
pick a5f4a0d added cat-file
```

Then, when the script drops you to the command line, you reset that commit, take the changes that have been reset, and create multiple commits out of them. When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (**f7f3f6d**), applies the second (**310154e**), and drops you to the console. There, you can do a mixed reset of that commit with `git reset HEAD^`, which effectively undoes that commit and leaves the modified files unstaged.

Now you can stage and commit files until you have several commits, and run `git rebase --continue` when you’re done:

```
$ git reset HEAD^  
$ git add README  
$ git commit -m 'updated README formatting'  
$ git add lib/simplegit.rb  
$ git commit -m 'added blame'  
$ git rebase --continue
```

Git applies the last commit (**a5f4a0d**) in the script, and your history looks like this:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Once again, this changes the SHA-1s of all the commits in your list, so make sure no commit shows up in that list that you've already pushed to a shared repository.

The Nuclear Option: filter-branch

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way – for instance, changing your email address globally or removing a file from every commit. The command is `filter-branch`, and it can rewrite huge swaths of your history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses so you can get an idea of some of the things it's capable of.

Removing a File from Every Commit

This occurs fairly commonly. Someone accidentally commits a huge binary file with a thoughtless `git add .`, and you want to remove it everywhere. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `filter-branch` is the tool you probably want to use to scrub your entire history. To remove a file named `passwords.txt` from your entire history, you can use the `--tree-filter` option to `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called `passwords.txt` from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like `git filter-branch --tree-filter 'rm -f *~' HEAD`.

You'll be able to watch Git rewriting trees and commits and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your master branch after you've determined the outcome is what you really want. To run `filter-branch` on all your branches, you can pass `--all` to the command.

Making a Subdirectory the New Root

Suppose you've done an import from another source control system and have subdirectories that make no sense (`trunk`, `tags`, and so on). If you want to make the `trunk` subdirectory be the new project root for every commit, `filter-branch` can help you do that, too:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory each time. Git will also automatically remove commits that did not affect the subdirectory.

Changing Email Addresses Globally

Another common case is that you forgot to run `git config` to set your name and email address before you started working, or perhaps you want to open-source a project at work and change all your work email addresses to your personal address. In any case, you can change email addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the email addresses that are yours, so you use `--commit-filter`:

```
$ git filter-branch --commit-filter '
  if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
  then
    GIT_AUTHOR_NAME="Scott Chacon";
    GIT_AUTHOR_EMAIL="schacon@example.com";
    git commit-tree "$@";
  else
    git commit-tree "$@";
  fi' HEAD
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA-1 in your history, not just those that have the matching email address.

Reset Demystified

Before moving on to more specialized tools, let's talk about `reset` and `checkout`. These commands are two of the most confusing parts of Git when you first encounter them. They do so many things, that it seems hopeless to actually understand them and employ them properly. For this, we recommend a simple metaphor.

The Three Trees

An easier way to think about `reset` and `checkout` is through the mental frame of Git being a content manager of three different trees. By “tree” here we really mean “collection of files”, not specifically the data structure. (There are a few cases where the index doesn’t exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

| Tree | Role |
|-------|-----------------------------------|
| HEAD | Last commit snapshot, next parent |
| Index | Proposed next commit snapshot |

| Tree | Role |
|-------------------|---------|
| Working Directory | Sandbox |

The HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It's generally simplest to think of HEAD as the **snapshot of your last commit**.

In fact, it's pretty easy to see what that snapshot looks like. Here is an example of getting the actual directory listing and SHA-1 checksums for each file in the HEAD snapshot:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

The `cat-file` and `ls-tree` commands are “plumbing” commands that are used for lower level things and not really used in day-to-day work, but they help us see what's going on here.

The Index

The Index is your **proposed next commit**. We've also been referring to this concept as Git's “Staging Area” as this is what Git looks at when you run `git commit`.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and `git commit` converts that into the tree for a new commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Again, here we're using `ls-files`, which is more of a behind the scenes command that shows you what your index currently looks like.

The index is not technically a tree structure – it's actually implemented as a flattened manifest – but for our purposes it's close enough.

The Working Directory

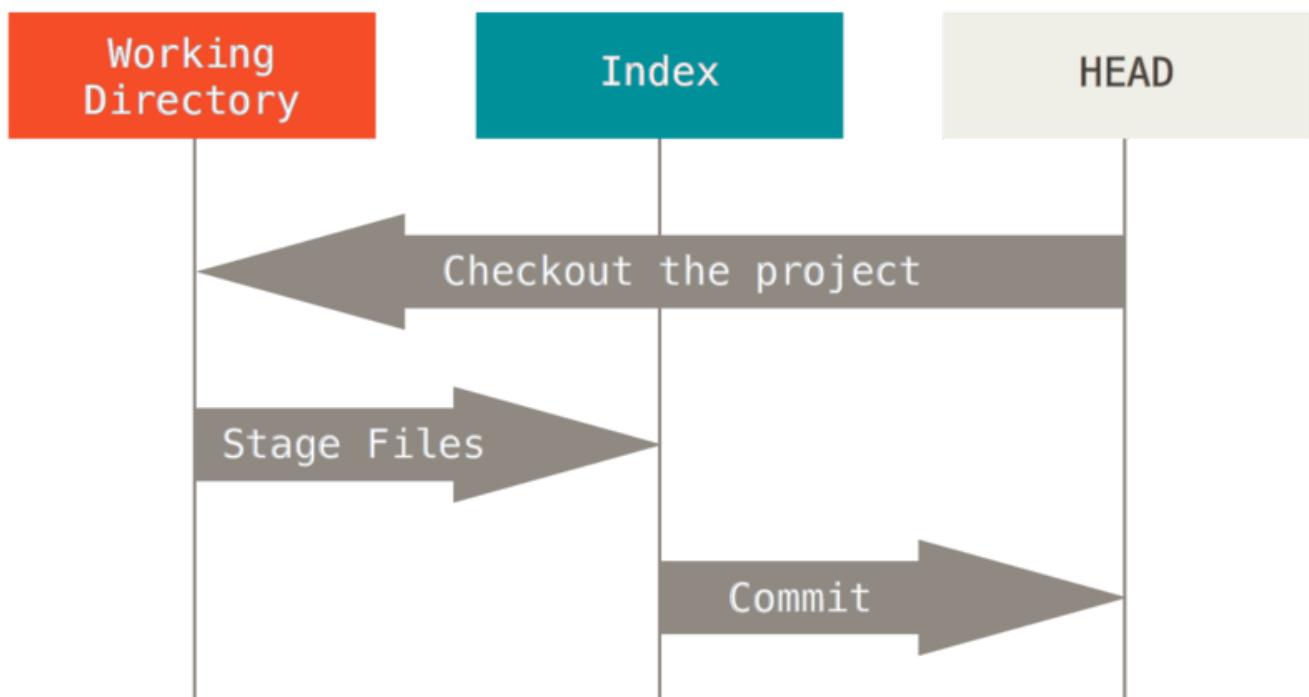
Finally, you have your working directory. The other two trees store their content in an efficient but inconvenient manner, inside the `.git` folder. The Working Directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the Working Directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

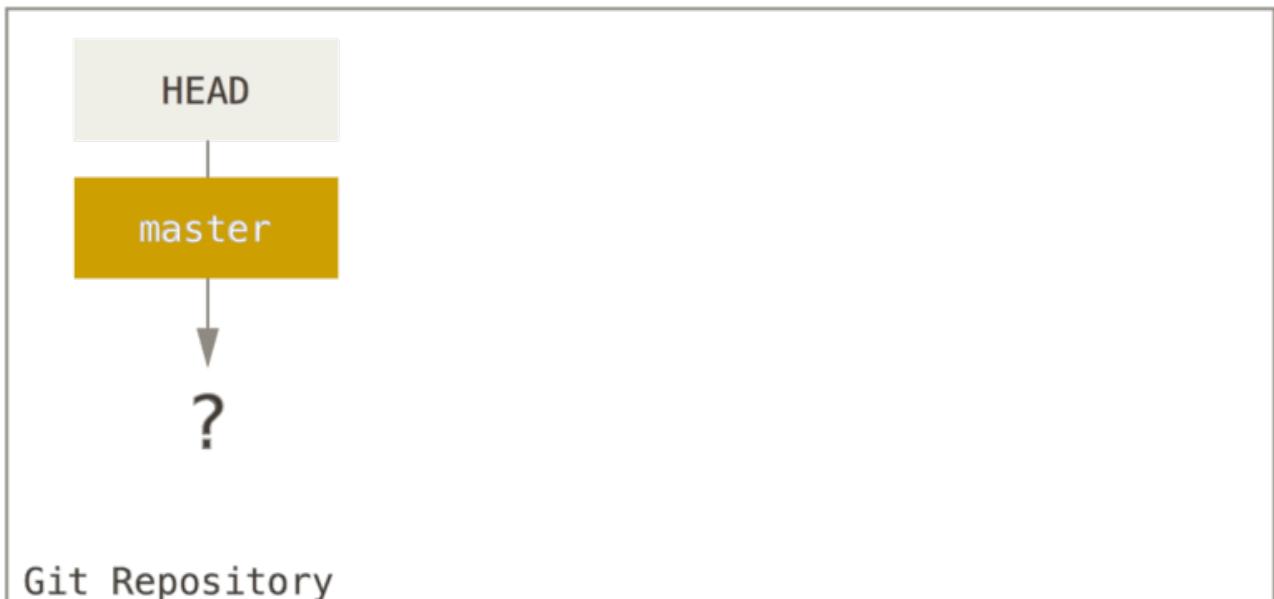
1 directory, 3 files
```

The Workflow

Git's main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.



Let's visualize this process: say you go into a new directory with a single file in it. We'll call this **v1** of the file, and we'll indicate it in blue. Now we run `git init`, which will create a Git repository with a HEAD reference which points to an unborn branch (`master` doesn't exist yet).



HEAD

master

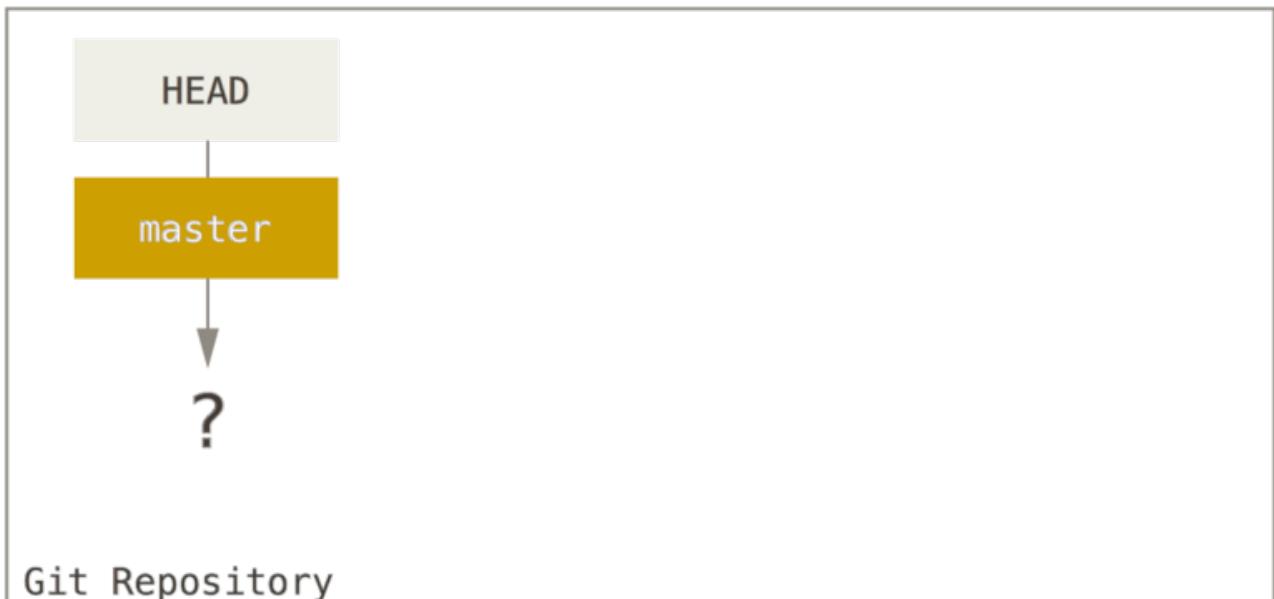
?

Git Repository

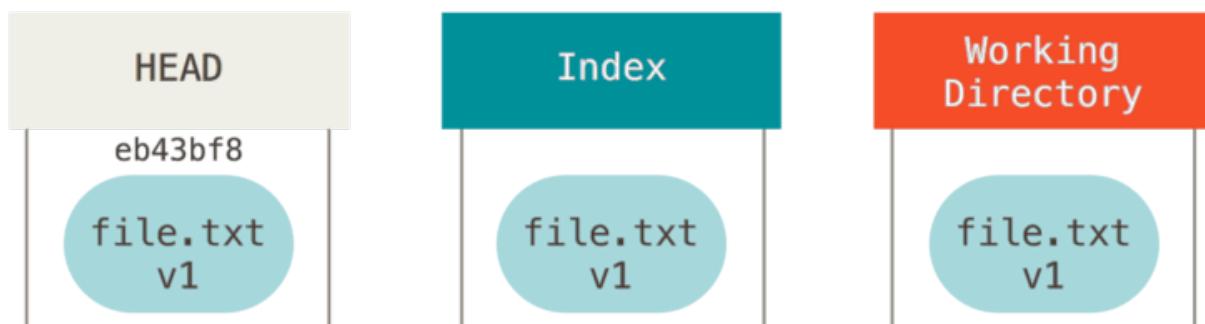
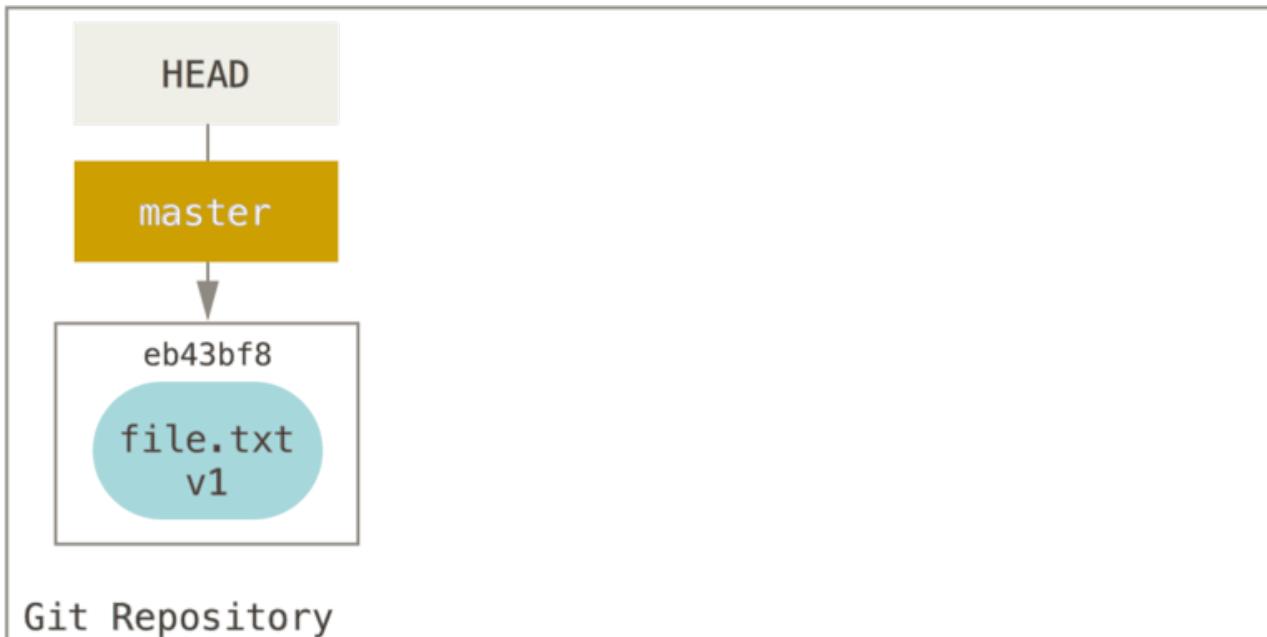


At this point, only the Working Directory tree has any content.

Now we want to commit this file, so we use `git add` to take content in the Working Directory and copy it to the Index.



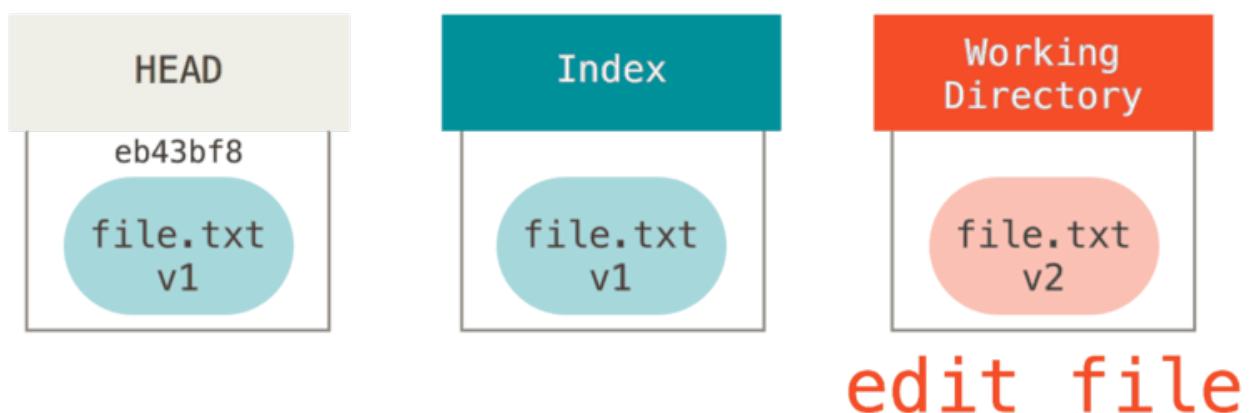
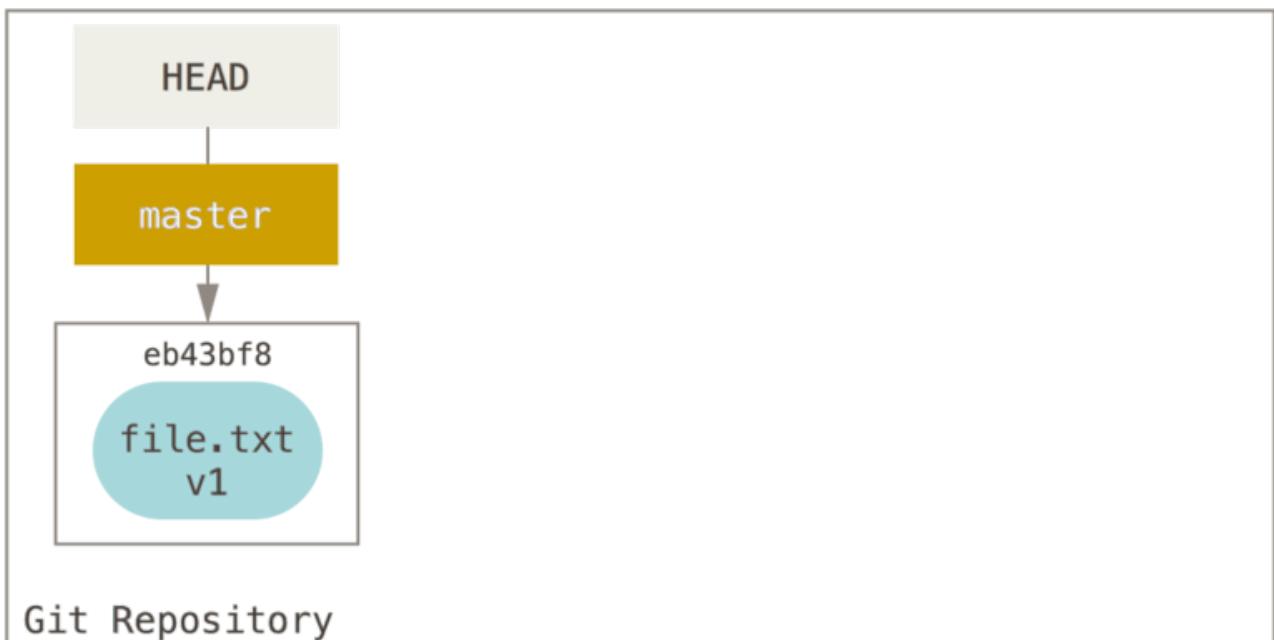
Then we run `git commit`, which takes the contents of the Index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates `master` to point to that commit.



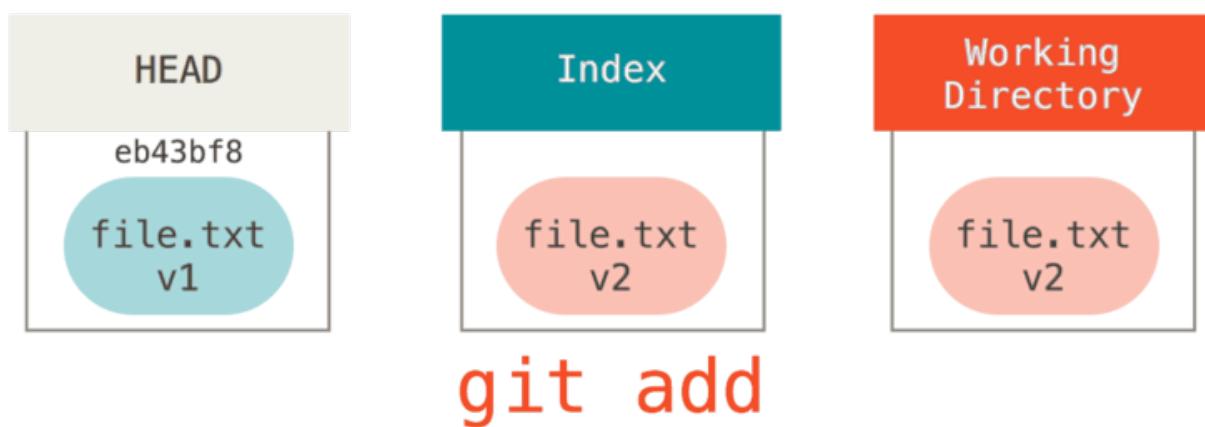
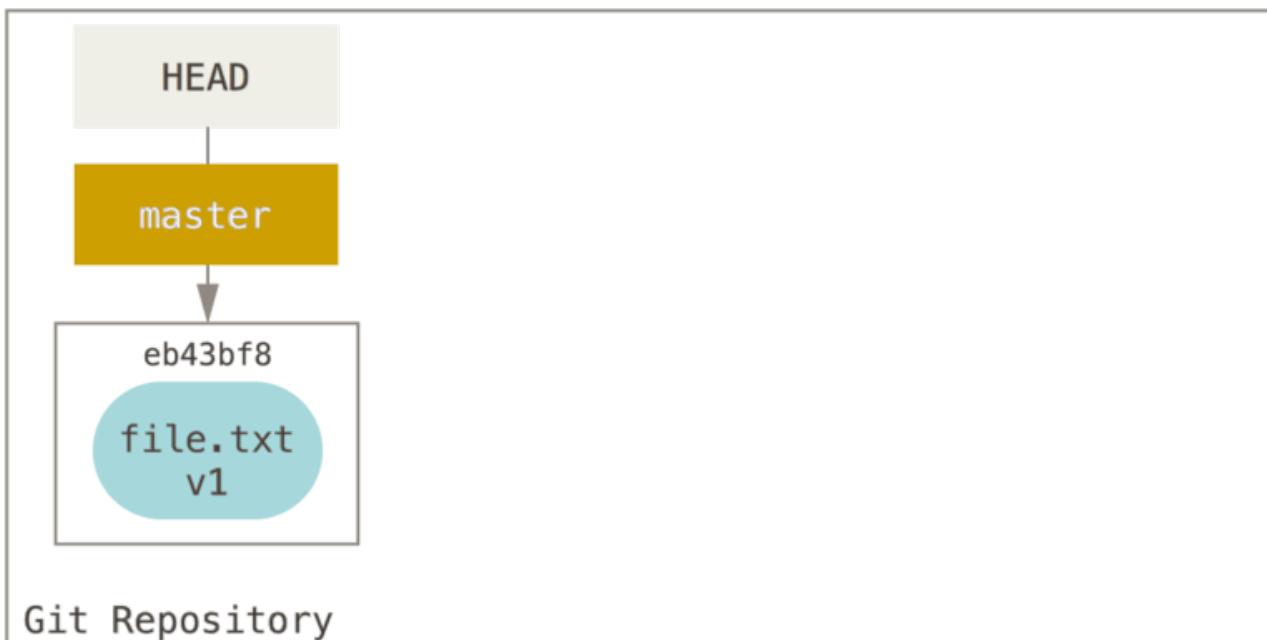
git commit

If we run `git status`, we'll see no changes, because all three trees are the same.

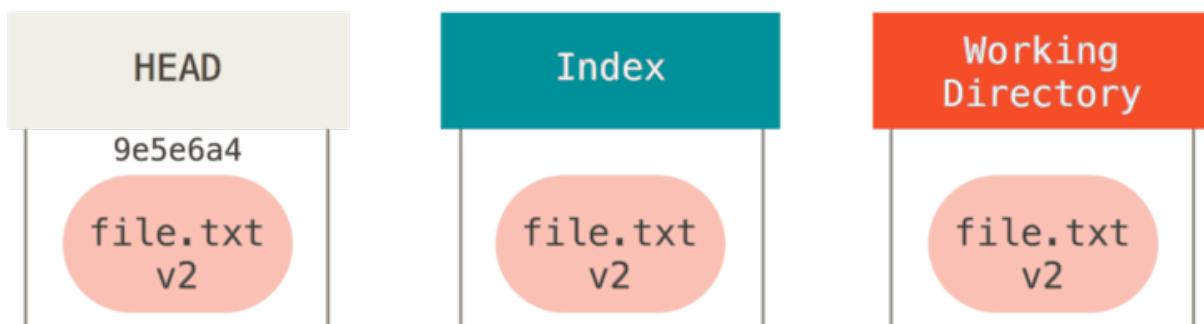
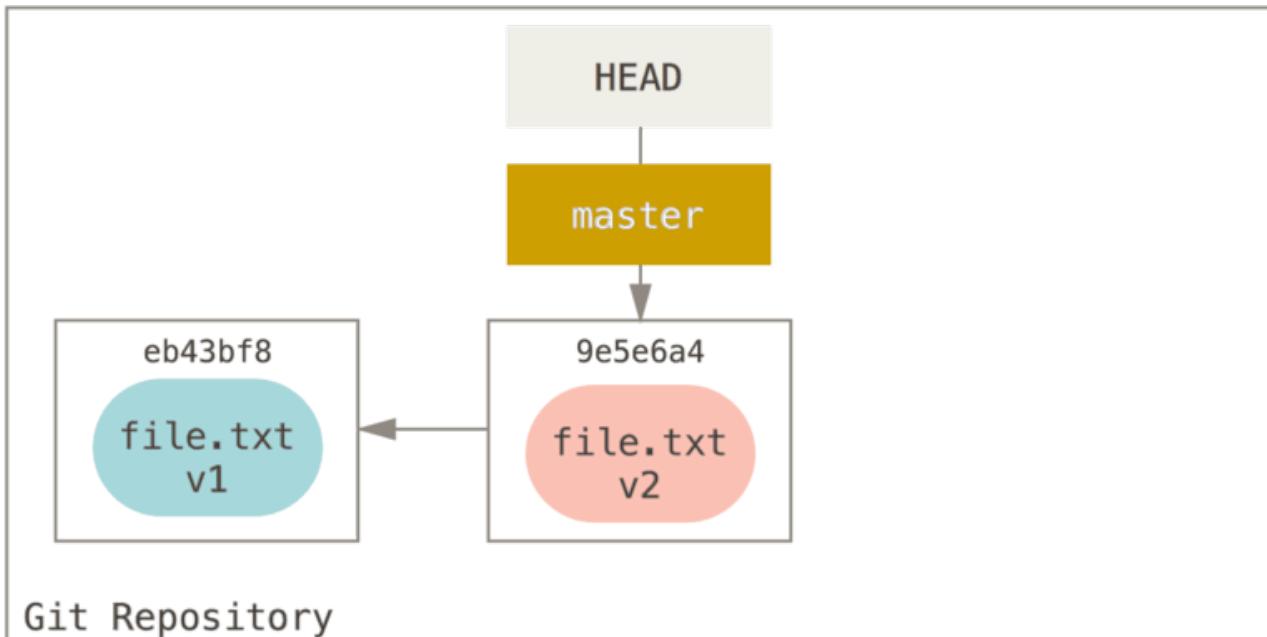
Now we want to make a change to that file and commit it. We'll go through the same process; first we change the file in our working directory. Let's call this **v2** of the file, and indicate it in red.



If we run `git status` right now, we'll see the file in red as "Changes not staged for commit," because that entry differs between the Index and the Working Directory. Next we run `git add` on it to stage it into our Index.



At this point if we run `git status` we will see the file in green under “Changes to be committed” because the Index and HEAD differ – that is, our proposed next commit is now different from our last commit. Finally, we run `git commit` to finalize the commit.



git commit

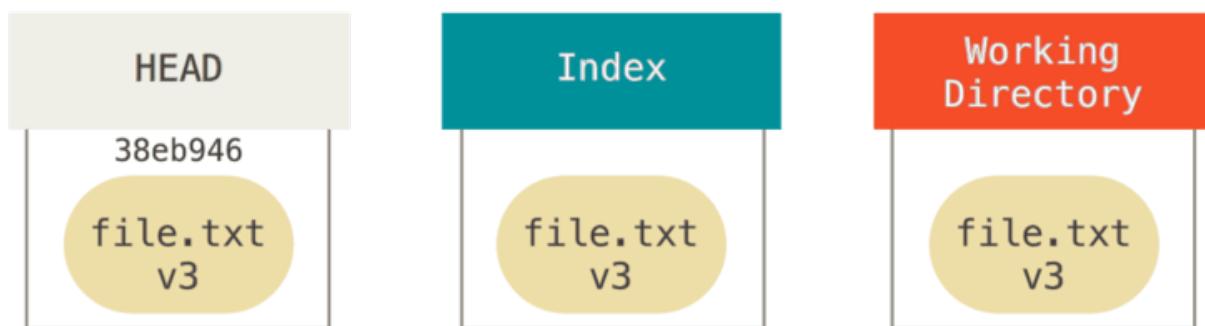
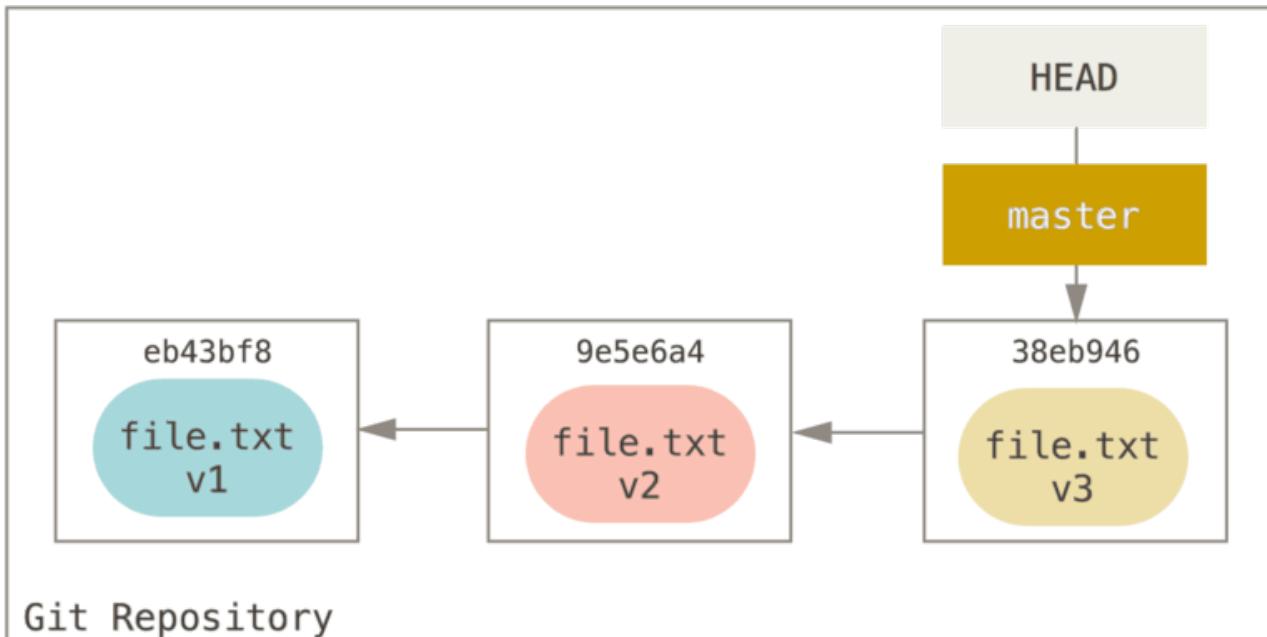
Now `git status` will give us no output, because all three trees are the same again.

Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **Index** with the snapshot of that commit, then copies the contents of the **Index** into your **Working Directory**.

The Role of Reset

The `reset` command makes more sense when viewed in this context.

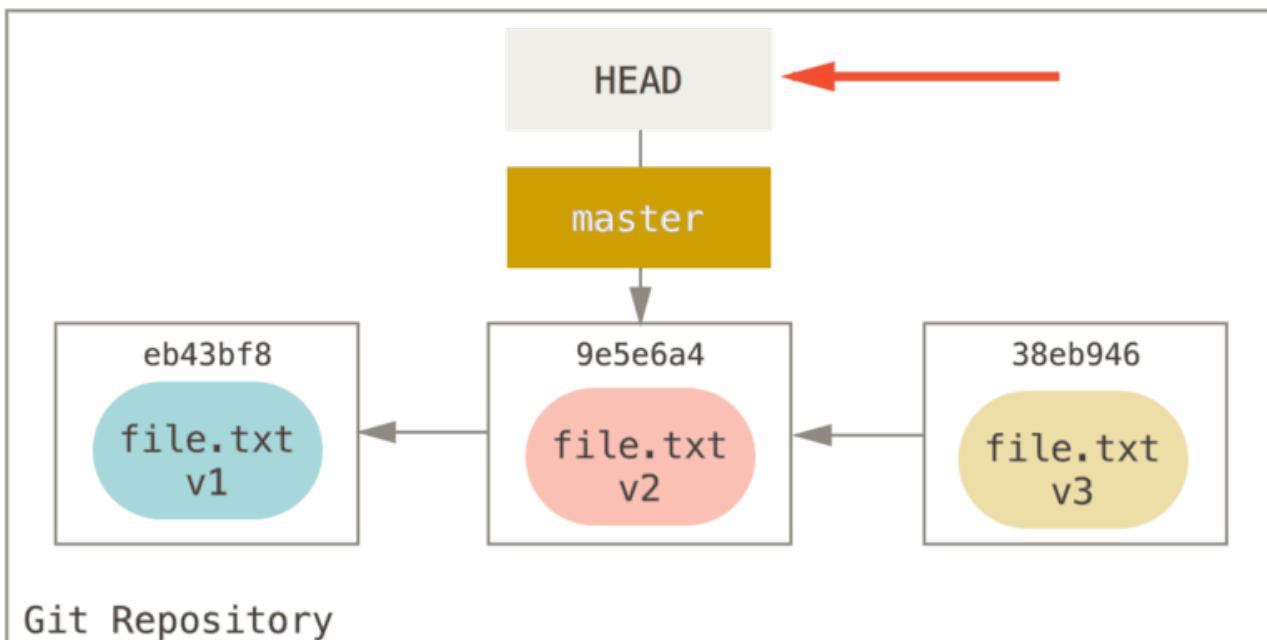
For the purposes of these examples, let's say that we've modified `file.txt` again and committed it a third time. So now our history looks like this:



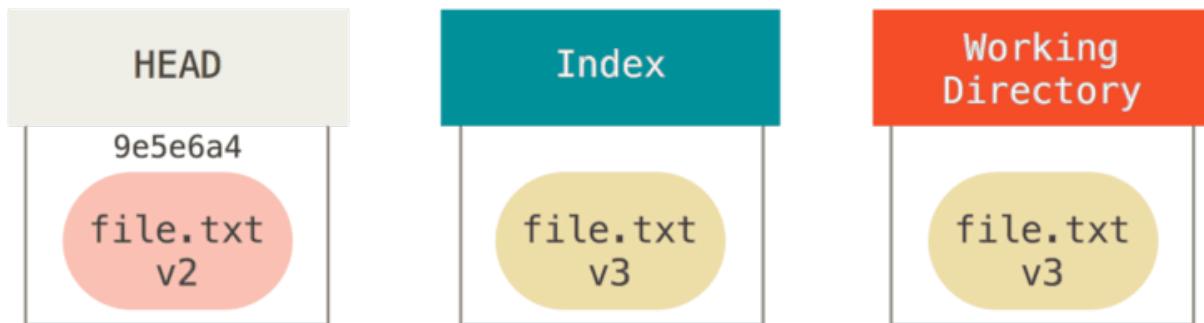
Let's now walk through exactly what `reset` does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

Step 1: Move HEAD

The first thing `reset` will do is move what HEAD points to. This isn't the same as changing HEAD itself (which is what `checkout` does); `reset` moves the branch that HEAD is pointing to. This means if HEAD is set to the `master` branch (i.e. you're currently on the `master` branch), running `git reset 9e5e6a4` will start by making `master` point to `9e5e6a4`.



Git Repository



git reset --soft HEAD~

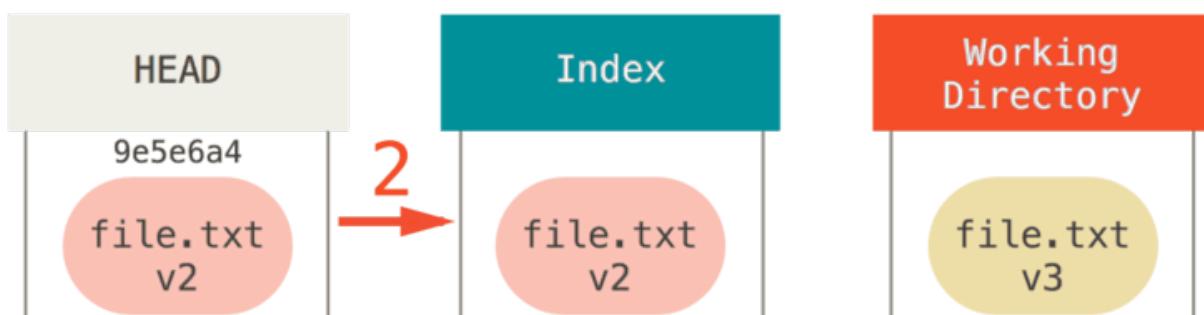
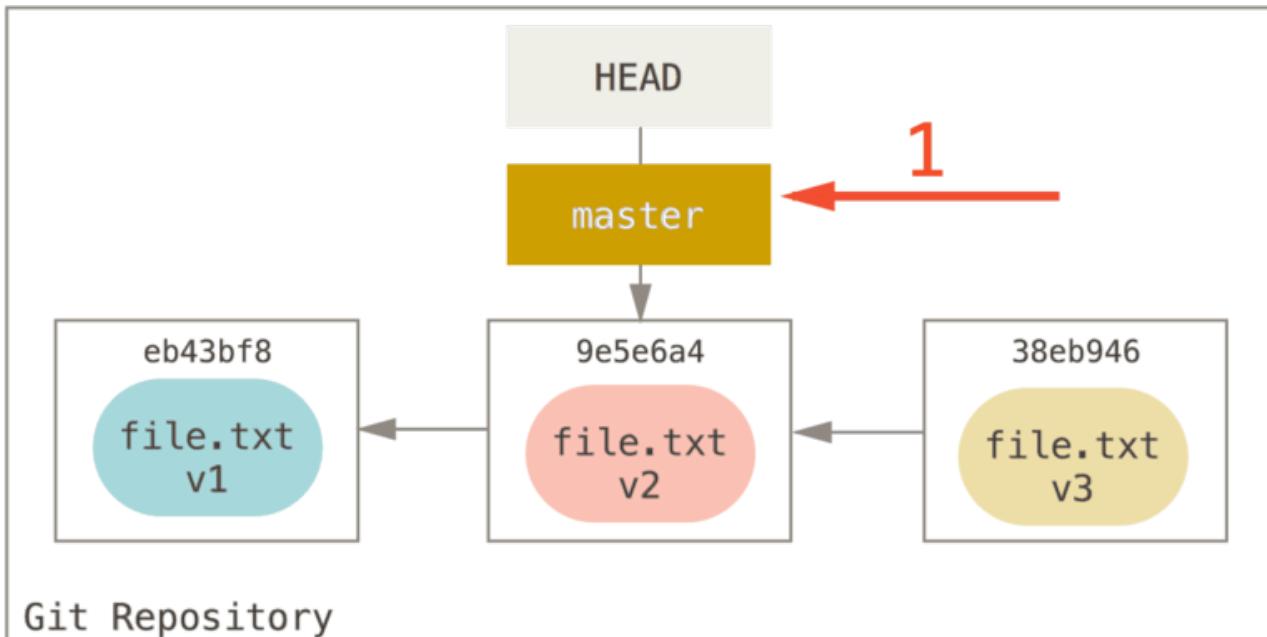
No matter what form of `reset` with a commit you invoke, this is the first thing it will always try to do. With `reset --soft`, it will simply stop there.

Now take a second to look at that diagram and realize what happened: it essentially undid the last `git commit` command. When you run `git commit`, Git creates a new commit and moves the branch that `HEAD` points to up to it. When you `reset` back to `HEAD~` (the parent of `HEAD`), you are moving the branch back to where it was, without changing the Index or Working Directory. You could now update the Index and run `git commit` again to accomplish what `git commit --amend` would have done (see [Changing the Last Commit](#)).

Step 2: Updating the Index (--mixed)

Note that if you run `git status` now you'll see in green the difference between the Index and what the new `HEAD` is.

The next thing `reset` will do is to update the Index with the contents of whatever snapshot `HEAD` now points to.



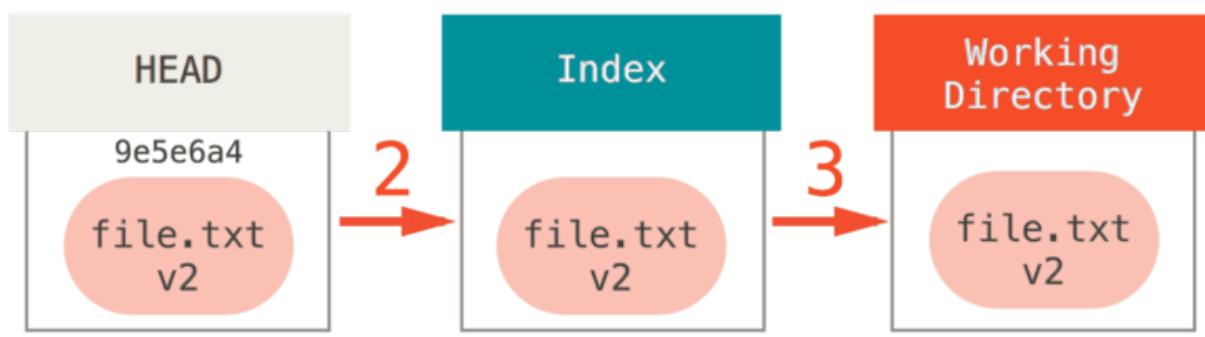
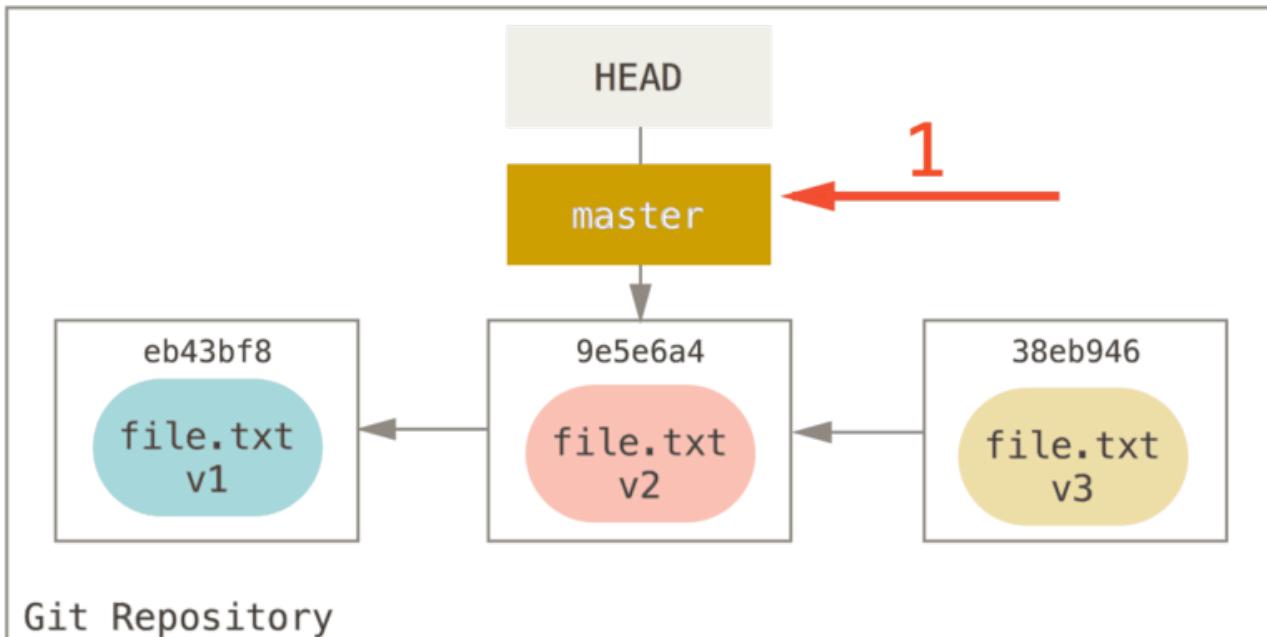
git reset [--mixed] HEAD~

If you specify the `--mixed` option, `reset` will stop at this point. This is also the default, so if you specify no option at all (just `git reset HEAD~` in this case), this is where the command will stop.

Now take another second to look at that diagram and realize what happened: it still undid your last `commit`, but also *unstaged* everything. You rolled back to before you ran all your `git add` and `git commit` commands.

Step 3: Updating the Working Directory (--hard)

The third thing that `reset` will do is to make the Working Directory look like the Index. If you use the `--hard` option, it will continue to this stage.



git reset --hard HEAD~

So let's think about what just happened. You undid your last commit, the `git add` and `git commit` commands, **and** all the work you did in your working directory.

It's important to note that this flag (`--hard`) is the only way to make the `reset` command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of `reset` can be pretty easily undone, but the `--hard` option cannot, since it forcibly overwrites files in the Working Directory. In this particular case, we still have the **v3** version of our file in a commit in our Git DB, and we could get it back by looking at our `reflog`, but if we had not committed it, Git still would have overwritten the file and it would be unrecoverable.

Recap

The `reset` command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch HEAD points to (*stop here if --soft*)
2. Make the Index look like HEAD (*stop here unless --hard*)
3. Make the Working Directory look like the Index

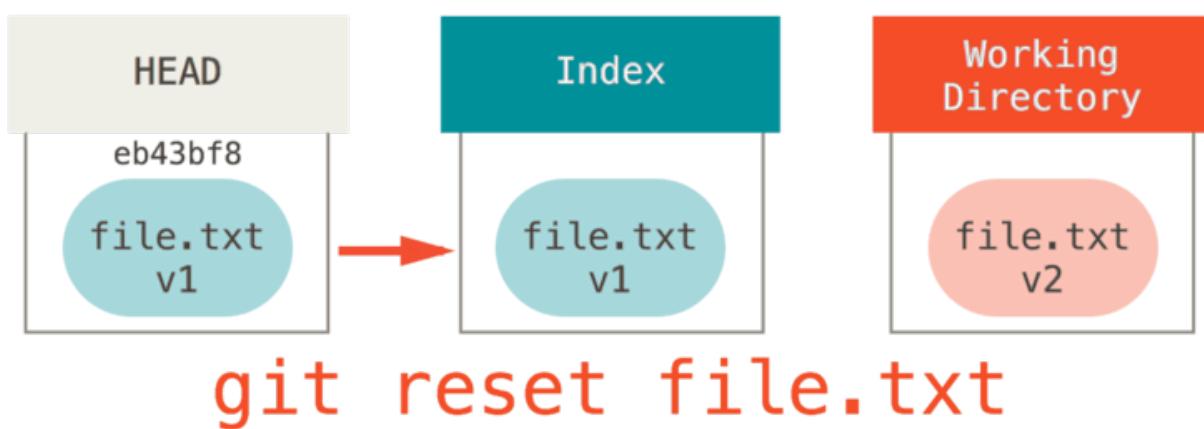
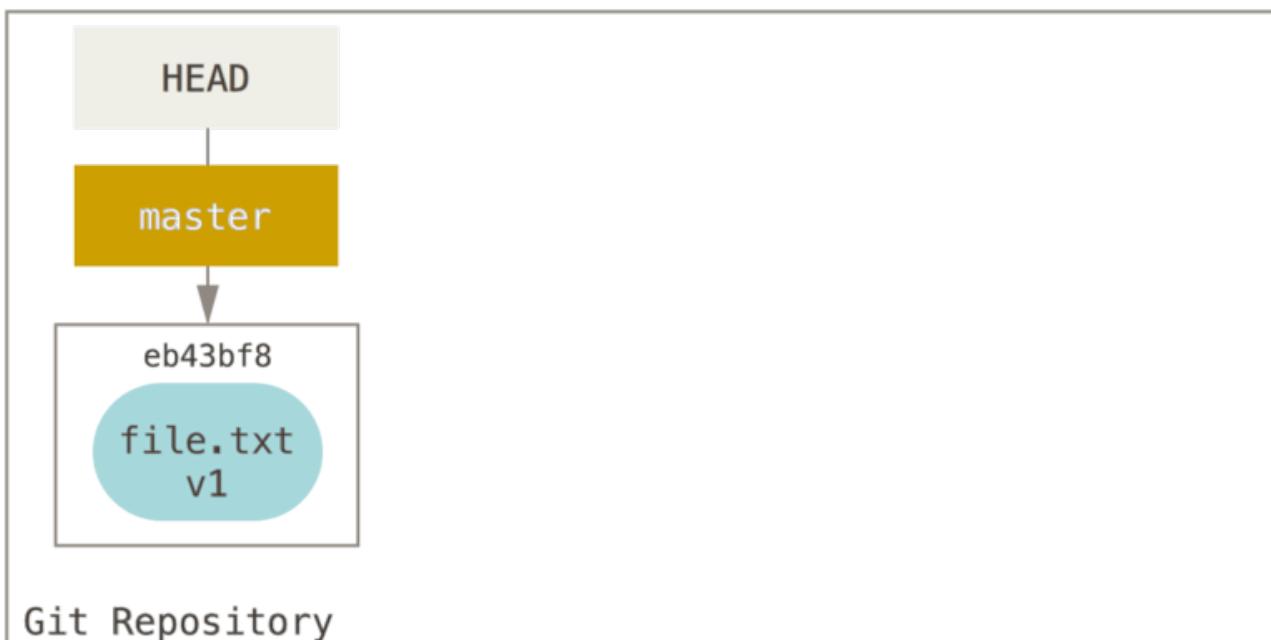
Reset With a Path

That covers the behavior of `reset` in its basic form, but you can also provide it with a path to act upon. If you specify a path, `reset` will skip step 1, and limit the remainder of its actions to a specific file or set of files. This actually sort of makes sense – HEAD is just a pointer, and you can't point to part of one commit and part of another. But the Index and Working directory *can* be partially updated, so `reset` proceeds with steps 2 and 3.

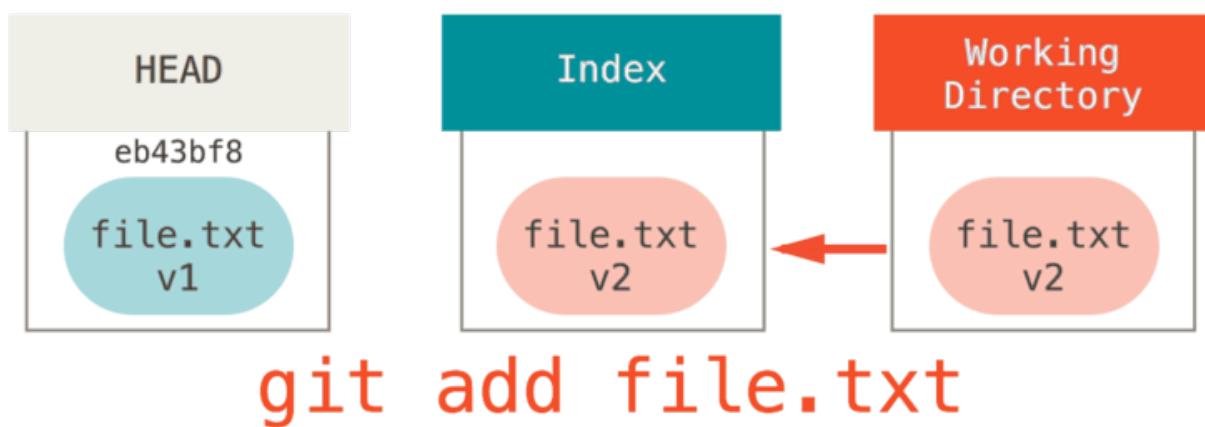
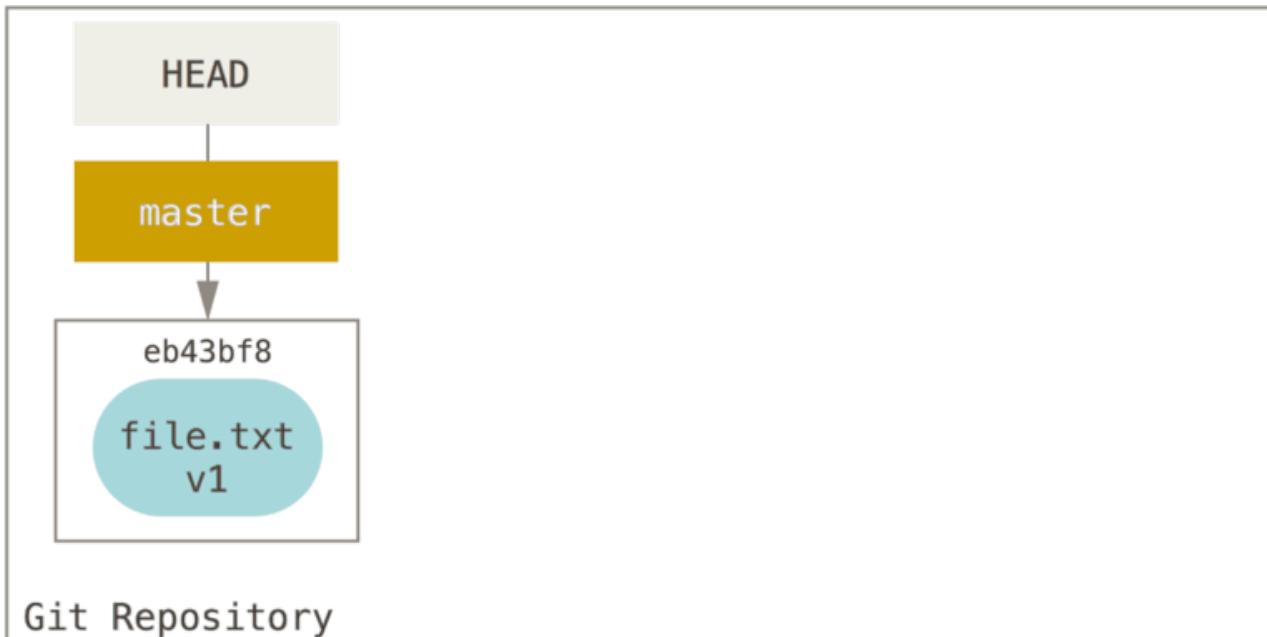
So, assume we run `git reset file.txt`. This form (since you did not specify a commit SHA-1 or branch, and you didn't specify `--soft` or `--hard`) is shorthand for `git reset --mixed HEAD file.txt`, which will:

1. Move the branch HEAD points to (*skipped*)
2. Make the Index look like HEAD (*stop here*)

So it essentially just copies `file.txt` from HEAD to the Index.

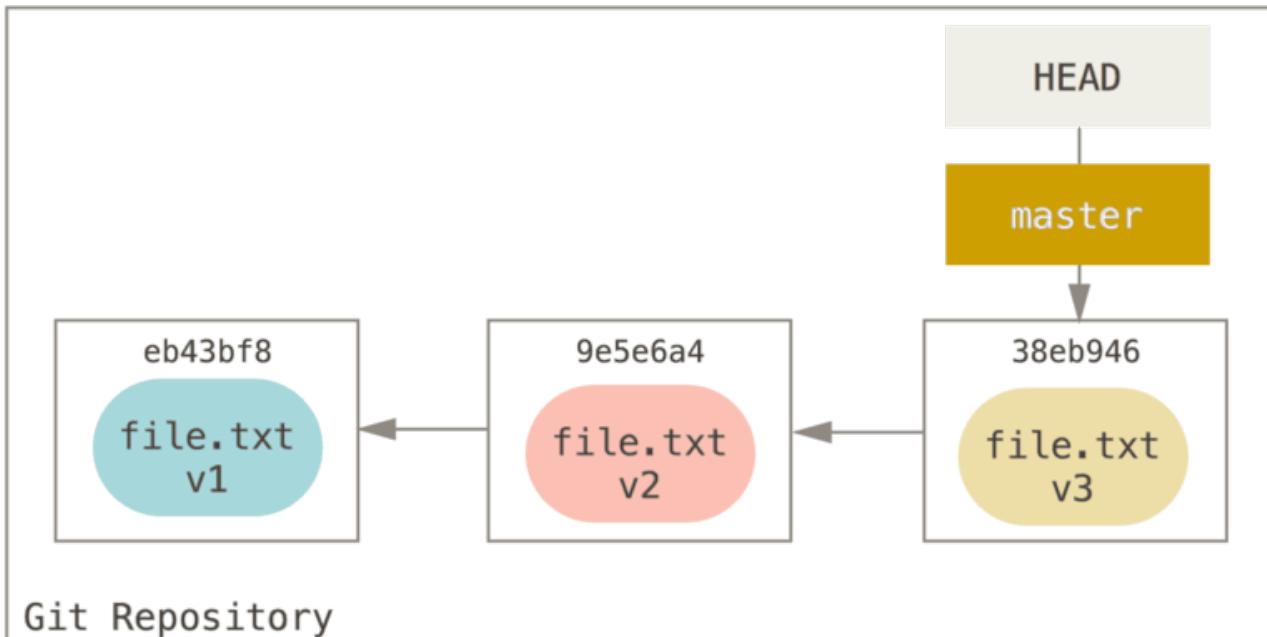


This has the practical effect of *unstaging* the file. If we look at the diagram for that command and think about what `git add` does, they are exact opposites.



This is why the output of the `git status` command suggests that you run this to unstage a file. (See [將已預存的檔案移出預存區](#) for more on this.)

We could just as easily not let Git assume we meant “pull the data from HEAD” by specifying a specific commit to pull that file version from. We would just run something like `git reset eb43bf file.txt`.



git reset eb43 -- file.txt

This effectively does the same thing as if we had reverted the content of the file to **v1** in the Working Directory, ran `git add` on it, then reverted it back to **v3** again (without actually going through all those steps). If we run `git commit` now, it will record a change that reverts that file back to **v1**, even though we never actually had it in our Working Directory again.

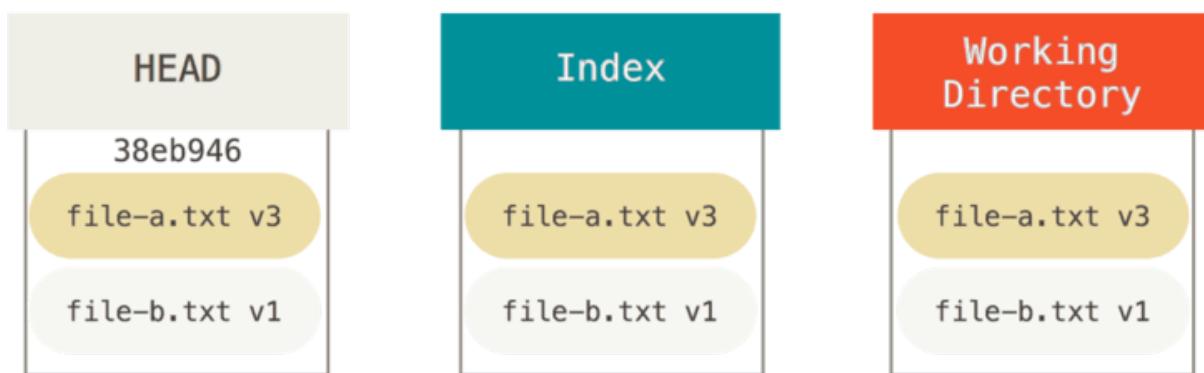
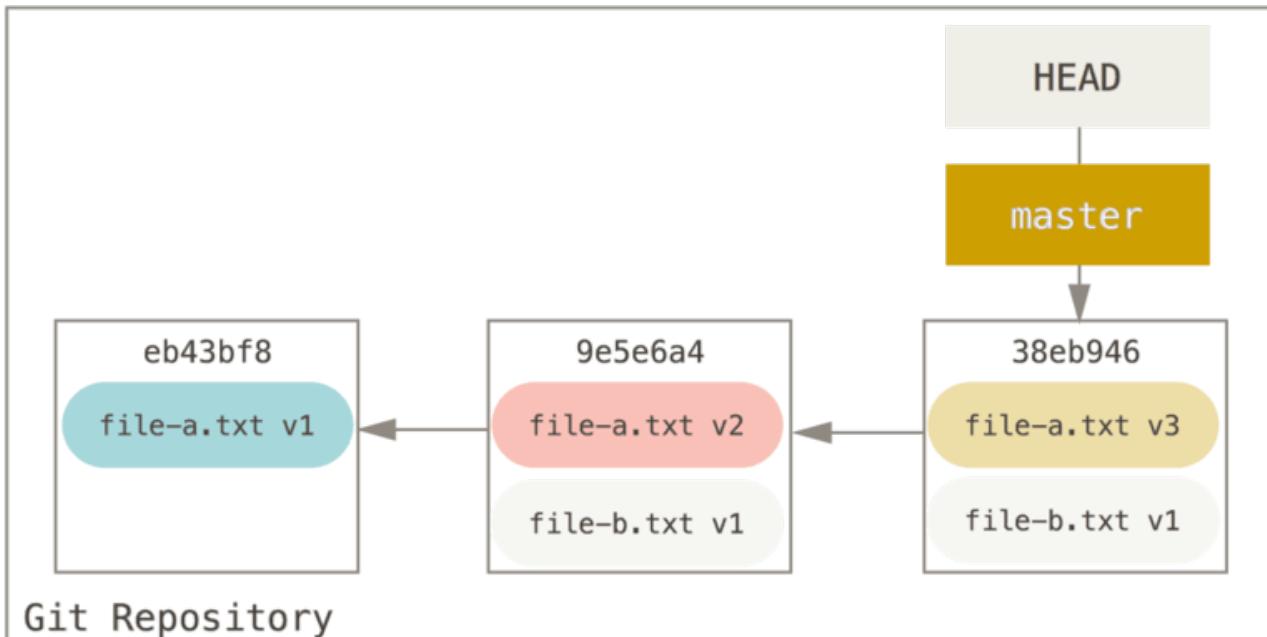
It's also interesting to note that like `git add`, the `reset` command will accept a `--patch` option to unstage content on a hunk-by-hunk basis. So you can selectively unstage or revert content.

Squashing

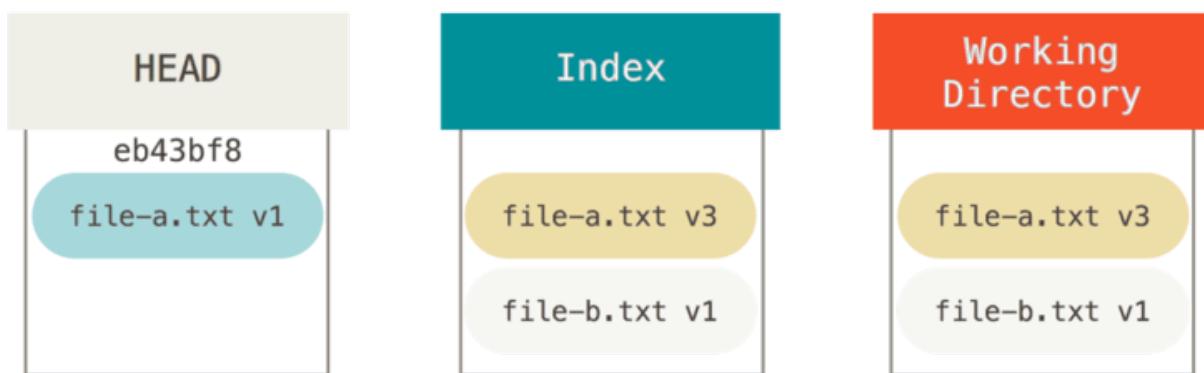
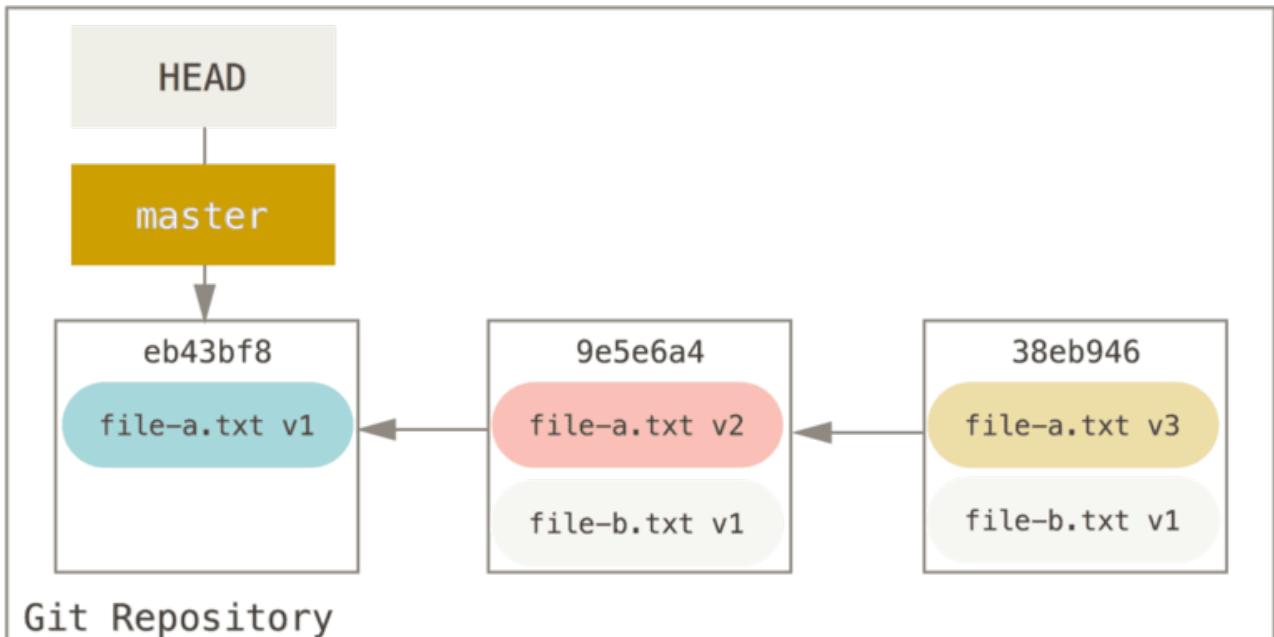
Let's look at how to do something interesting with this newfound power – squashing commits.

Say you have a series of commits with messages like “oops.”, “WIP” and “forgot this file”. You can use `reset` to quickly and easily squash them into a single commit that makes you look really smart. ([Squashing Commits](#) shows another way to do this, but in this example it's simpler to use `reset`.)

Let's say you have a project where the first commit has one file, the second commit added a new file and changed the first, and the third commit changed the first file again. The second commit was a work in progress and you want to squash it down.

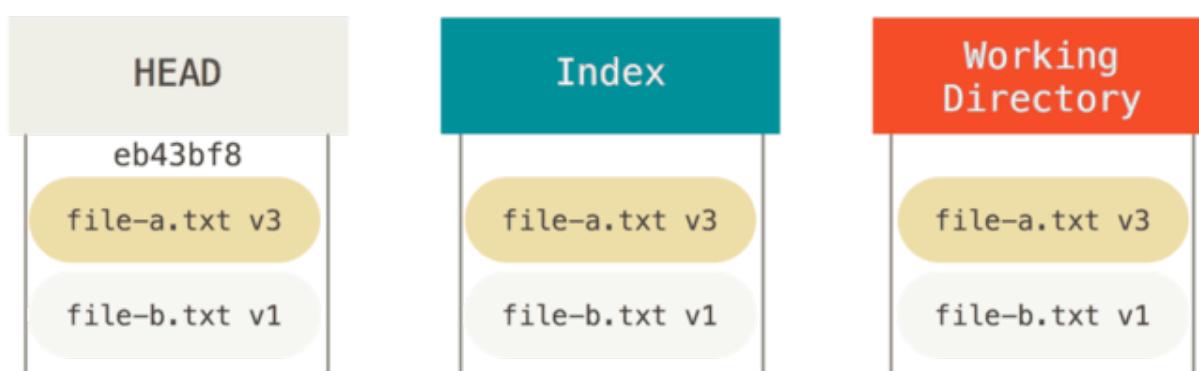
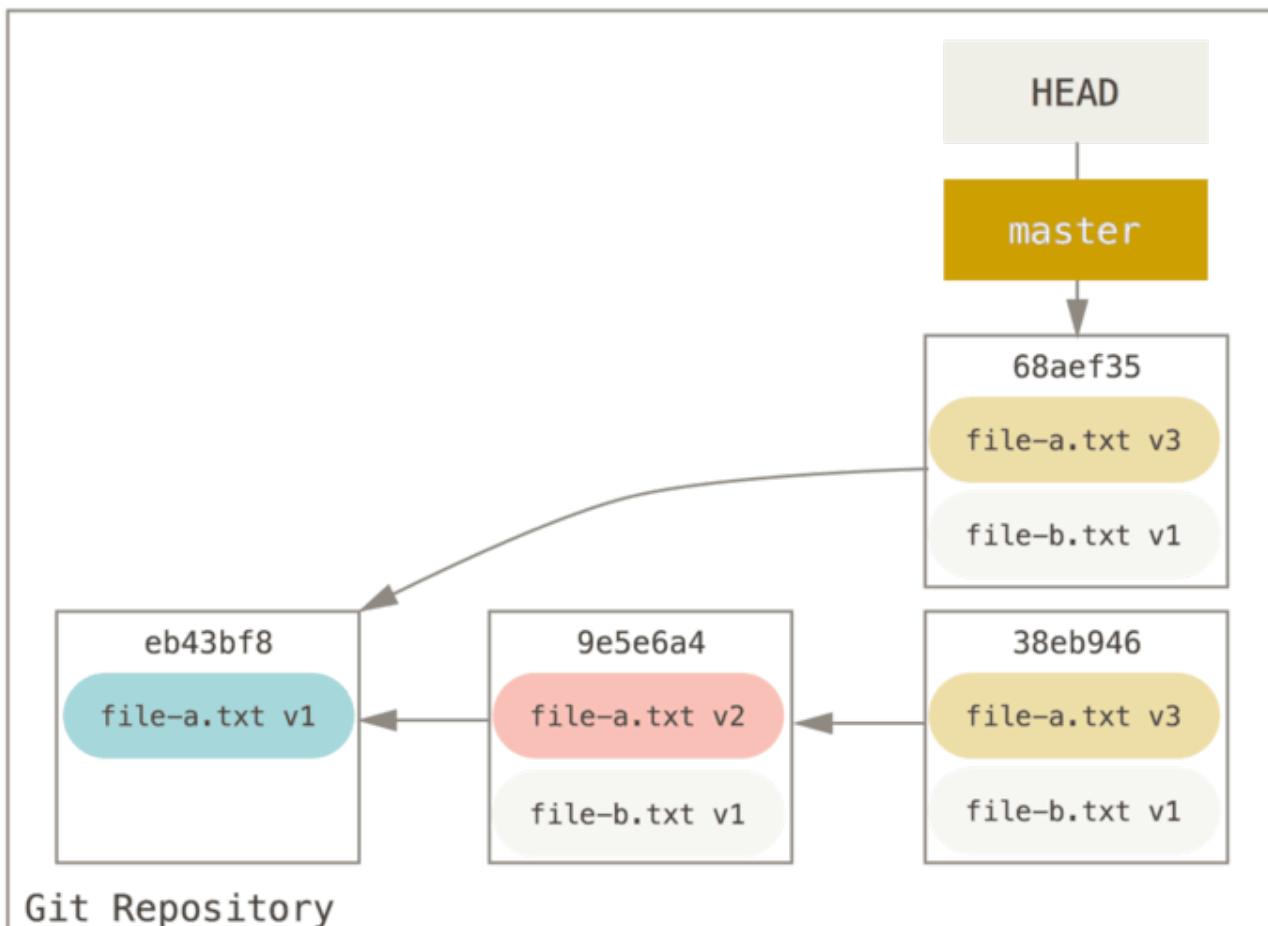


You can run `git reset --soft HEAD~2` to move the HEAD branch back to an older commit (the first commit you want to keep):



git reset --soft HEAD~2

And then simply run `git commit` again:



git commit

Now you can see that your reachable history, the history you would push, now looks like you had one commit with `file-a.txt` v1, then a second that both modified `file-a.txt` to v3 and added `file-b.txt`. The commit with the v2 version of the file is no longer in the history.

Check It Out

Finally, you may wonder what the difference between `checkout` and `reset` is. Like `reset`, `checkout` manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

Without Paths

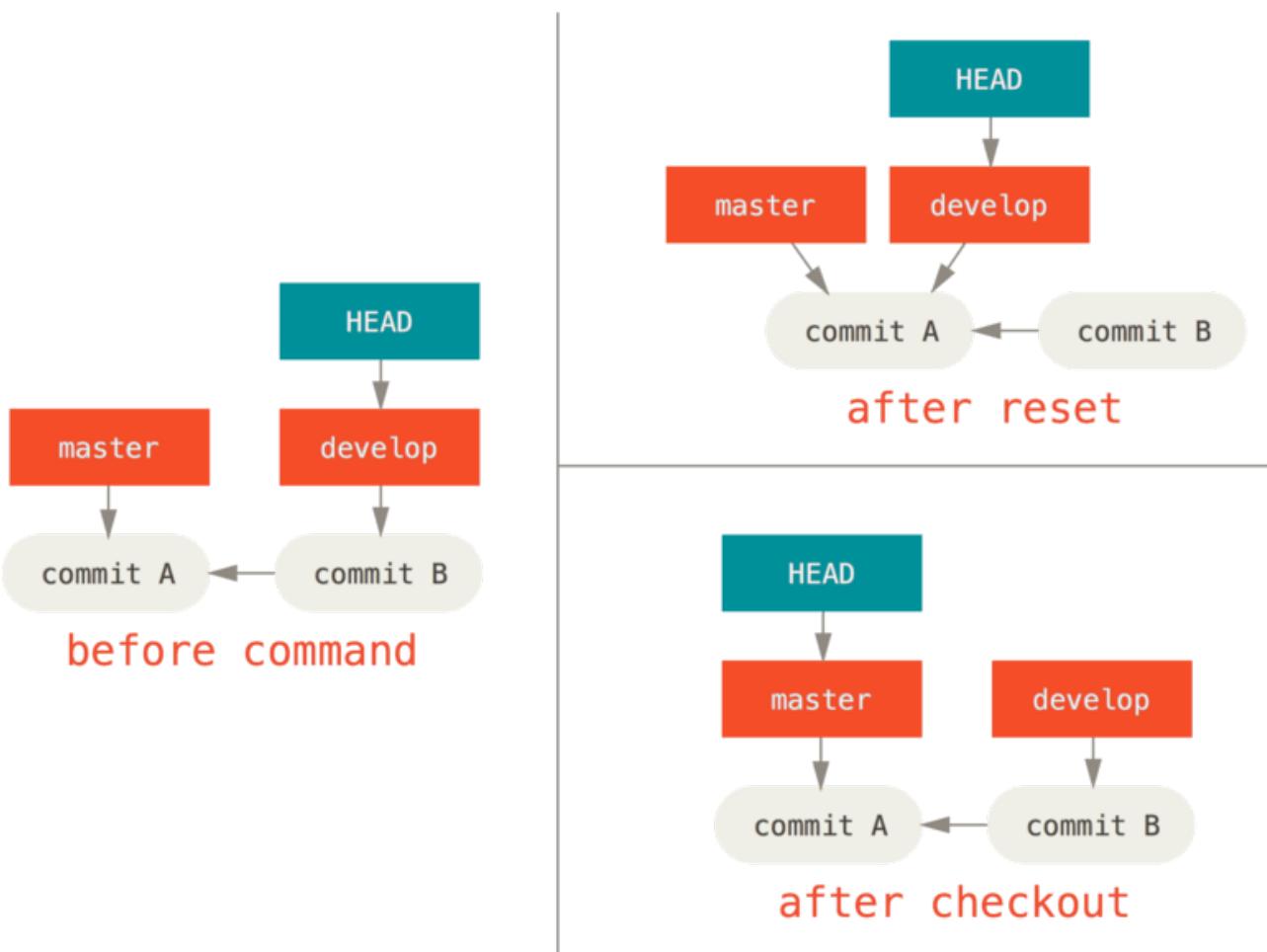
Running `git checkout [branch]` is pretty similar to running `git reset --hard [branch]` in that it updates all three trees for you to look like `[branch]`, but there are two important differences.

First, unlike `reset --hard`, `checkout` is working-directory safe; it will check to make sure it's not blowing away files that have changes to them. Actually, it's a bit smarter than that – it tries to do a trivial merge in the Working Directory, so all of the files you *haven't* changed in will be updated. `reset --hard`, on the other hand, will simply replace everything across the board without checking.

The second important difference is how it updates HEAD. Where `reset` will move the branch that HEAD points to, `checkout` will move HEAD itself to point to another branch.

For instance, say we have `master` and `develop` branches which point at different commits, and we're currently on `develop` (so HEAD points to it). If we run `git reset master`, `develop` itself will now point to the same commit that `master` does. If we instead run `git checkout master`, `develop` does not move, HEAD itself does. HEAD will now point to `master`.

So, in both cases we're moving HEAD to point to commit A, but *how* we do so is very different. `reset` will move the branch HEAD points to, `checkout` moves HEAD itself.



With Paths

The other way to run `checkout` is with a file path, which, like `reset`, does not move HEAD. It is just like `git reset [branch] file` in that it updates the index with that file at that commit, but it also

overwrites the file in the working directory. It would be exactly like `git reset --hard [branch]` if `reset` would let you run that) – it’s not working-directory safe, and it does not move HEAD.

Also, like `git reset` and `git add, checkout` will accept a `--patch` option to allow you to selectively revert file contents on a hunk-by-hunk basis.

Summary

Hopefully now you understand and feel more comfortable with the `reset` command, but are probably still a little confused about how exactly it differs from `checkout` and could not possibly remember all the rules of the different invocations.

Here’s a cheat-sheet for which commands affect which trees. The “HEAD” column reads “REF” if that command moves the reference (branch) that HEAD points to, and “HEAD” if it moves HEAD itself. Pay especial attention to the *WD Safe?* column – if it says **NO**, take a second to think before running that command.

| | HEAD | Index | Workdir | WD Safe? |
|---------------------------------------|------|-------|---------|-----------|
| Commit Level | | | | |
| <code>reset --soft [commit]</code> | REF | NO | NO | YES |
| <code>reset [commit]</code> | REF | YES | NO | YES |
| <code>reset --hard [commit]</code> | REF | YES | YES | NO |
| <code>checkout [commit]</code> | HEAD | YES | YES | YES |
| File Level | | | | |
| <code>reset (commit) [file]</code> | NO | YES | NO | YES |
| <code>checkout (commit) [file]</code> | NO | YES | YES | NO |

Advanced Merging

Merging in Git is typically fairly easy. Since Git makes it easy to merge another branch multiple times, it means that you can have a very long lived branch but you can keep it up to date as you go, solving small conflicts often, rather than be surprised by one enormous conflict at the end of the series.

However, sometimes tricky conflicts do occur. Unlike some other version control systems, Git does not try to be overly clever about merge conflict resolution. Git’s philosophy is to be smart about determining when a merge resolution is unambiguous, but if there is a conflict, it does not try to be clever about automatically resolving it. Therefore, if you wait too long to merge two branches that diverge quickly, you can run into some issues.

In this section, we’ll go over what some of those issues might be and what tools Git gives you to help handle these more tricky situations. We’ll also cover some of the different, non-standard types of merges you can do, as well as see how to back out of merges that you’ve done.

Merge Conflicts

While we covered some basics on resolving merge conflicts in [合併衝突的基本解法](#), for more complex conflicts, Git provides a few tools to help you figure out what’s going on and how to better deal with the conflict.

First of all, if at all possible, try to make sure your working directory is clean before doing a merge that may have conflicts. If you have work in progress, either commit it to a temporary branch or stash it. This makes it so that you can undo **anything** you try here. If you have unsaved changes in your working directory when you try a merge, some of these tips may help you lose that work.

Let's walk through a very simple example. We have a super simple Ruby file that prints *hello world*.

```
#! /usr/bin/env ruby

def hello
    puts 'hello world'
end

hello()
```

In our repository, we create a new branch named **whitespace** and proceed to change all the Unix line endings to DOS line endings, essentially changing every line of the file, but just with whitespace. Then we change the line "hello world" to "hello mundo".

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
- puts 'hello world'
+ puts 'hello mundo'^M
 end

hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now we switch back to our **master** branch and add some documentation for the function.

```
$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
    puts 'hello world'
end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

Now we try to merge in our `whitespace` branch and we'll get conflicts because of the whitespace changes.

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Aborting a Merge

We now have a few options. First, let's cover how to get out of this situation. If you perhaps weren't expecting conflicts and don't want to quite deal with the situation yet, you can simply back out of the merge with `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

The `git merge --abort` option tries to revert back to your state before you ran the merge. The only cases where it may not be able to do this perfectly would be if you had unstashed, uncommitted changes in your working directory when you ran it, otherwise it should work fine.

If for some reason you just want to start over, you can also run `git reset --hard HEAD`, and your repository will be back to the last committed state. Remember that any uncommitted work will be lost, so make sure you don't want any of your changes.

Ignoring Whitespace

In this specific case, the conflicts are whitespace related. We know this because the case is simple, but it's also pretty easy to tell in real cases when looking at the conflict because every line is removed on one side and added again on the other. By default, Git sees all of these lines as being changed, so it can't merge the files.

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with `-Xignore-all-space` or `-Xignore-space-change`. The first option ignores whitespace **completely** when comparing lines, the second treats sequences of one or more whitespace characters as equivalent.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Since in this case, the actual file changes were not conflicting, once we ignore the whitespace changes, everything merges just fine.

This is a lifesaver if you have someone on your team who likes to occasionally reformat everything from spaces to tabs or vice-versa.

Manual File Re-merging

Though Git handles whitespace pre-processing pretty well, there are other types of changes that perhaps Git can't handle automatically, but are scriptable fixes. As an example, let's pretend that Git could not handle the whitespace change and we needed to do it by hand.

What we really need to do is run the file we're trying to merge in through a `dos2unix` program before trying the actual file merge. So how would we do that?

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we're merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under "stages" which each have numbers associated with them. Stage 1 is the common ancestor, stage 2 is your version and stage 3 is from the `MERGE_HEAD`, the version you're merging in ("theirs").

You can extract a copy of each of these versions of the conflicted file with the `git show` command and a special syntax.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

If you want to get a little more hard core, you can also use the `ls-files -u` plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1    hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2    hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3    hello.rb
```

The `:1:hello.rb` is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known `git merge-file` command which does just that.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

  +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

At this point we have nicely merged the file. In fact, this actually works better than the `ignore-space-change` option because this actually fixes the whitespace changes before merge instead of simply ignoring them. In the `ignore-space-change` merge, we actually ended up with a few lines with DOS line endings, making things mixed.

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask `git diff` to compare what is in your working directory that you're about to commit as the result of the merge to any of these stages. Let's go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@
 
 # prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

So here we can easily see that what happened in our branch, what we're actually introducing to this file with this merge, is changing that single line.

If we want to see how the result of the merge differed from what was on their side, you can run `git diff --theirs`. In this and the following example, we have to use `-b` to strip out the whitespace because we're comparing it to what is in Git, not our cleaned up `hello.theirs.rb` file.

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
  puts 'hello mundo'
end
```

Finally, you can see how the file has changed from both sides with `git diff --base`.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

At this point we can use the `git clean` command to clear out the extra files we created to do the manual merge but no longer need.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Checking Out Conflicts

Perhaps we're not happy with the resolution at this point for some reason, or maybe manually editing one or both sides still didn't work well and we need more context.

Let's change up the example a little. For this example, we have two longer lived branches that each have a few commits in them but create a legitimate content conflict when merged.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|
* b7dcc89 initial hello world code
```

We now have three unique commits that live only on the `master` branch and three others that live on the `mundo` branch. If we try to merge the `mundo` branch in, we get a conflict.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

We would like to see what the merge conflict is. If we open up the file, we'll see something like this:

```
#!/usr/bin/env ruby

def hello
<<<<< HEAD
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> mundo
end

hello()
```

Both sides of the merge added content to this file, but some of the commits modified the file in the same place that caused this conflict.

Let's explore a couple of tools that you now have at your disposal to determine how this conflict came to be. Perhaps it's not obvious how exactly you should fix this conflict. You need more context.

One helpful tool is `git checkout` with the '--conflict' option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass `--conflict` either `diff3` or `merge` (which is the default). If you pass it `diff3`, Git will use a slightly different version of conflict markers, not only giving you the "ours" and "theirs" versions, but also the "base" version inline to give you more context.

```
$ git checkout --conflict=diff3 hello.rb
```

Once we run that, the file will look like this instead:

```
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=====
  puts 'hello mundo'
>>>>> theirs
end

hello()
```

If you like this format, you can set it as the default for future merge conflicts by setting the `merge.conflictstyle` setting to `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

The `git checkout` command can also take `--ours` and `--theirs` options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing.

Merge Log

Another useful tool when resolving merge conflicts is `git log`. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in [Triple Dot](#).

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

That’s a nice list of the six total commits involved, as well as which line of development each commit was on.

We can further simplify this though to give us much more specific context. If we add the `--merge` option to `git log`, it will only show the commits in either side of the merge that touch a file that’s

currently conflicted.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

If you run that with the `-p` option instead, you get just the diffs to the file that ended up in conflict. This can be **really** helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

Combined Diff Format

Since Git stages any merge results that are successful, when you run `git diff` while in a conflicted merge state, you only get what is currently still in conflict. This can be helpful to see what you still have to resolve.

When you run `git diff` directly after a merge conflict, it will give you information in a rather unique diff output format.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
#! /usr/bin/env ruby

def hello
++<<<<< HEAD
+ puts 'hola world'
=====
+ puts 'hello mundo'
++>>>>> mundo
end

hello()
```

The format is called “Combined Diff” and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the “ours” branch and the file in your working directory and the second column does the same between the “theirs” branch and your working directory copy.

So in that example you can see that the `<<<<<` and `>>>>>` lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we’re expected to remove them.

If we resolve the conflict and run `git diff` again, we’ll see the same thing, but it’s a little more useful.

```

$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-   puts 'hello mundo'
++ puts 'hola mundo'
  end

hello()

```

This shows us that “hola world” was in our side but not in the working copy, that “hello mundo” was in their side but not in the working copy and finally that “hola mundo” was not in either side but is now in the working copy. This can be useful to review before committing the resolution.

You can also get this from the `git log` for any merge to see how something was resolved after the fact. Git will output this format if you run `git show` on a merge commit, or if you add a `--cc` option to a `git log -p` (which by default only shows patches for non-merge commits).

```

$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

  Merge branch 'mundo'

  Conflicts:
    hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-   puts 'hello mundo'
++ puts 'hola mundo'
  end

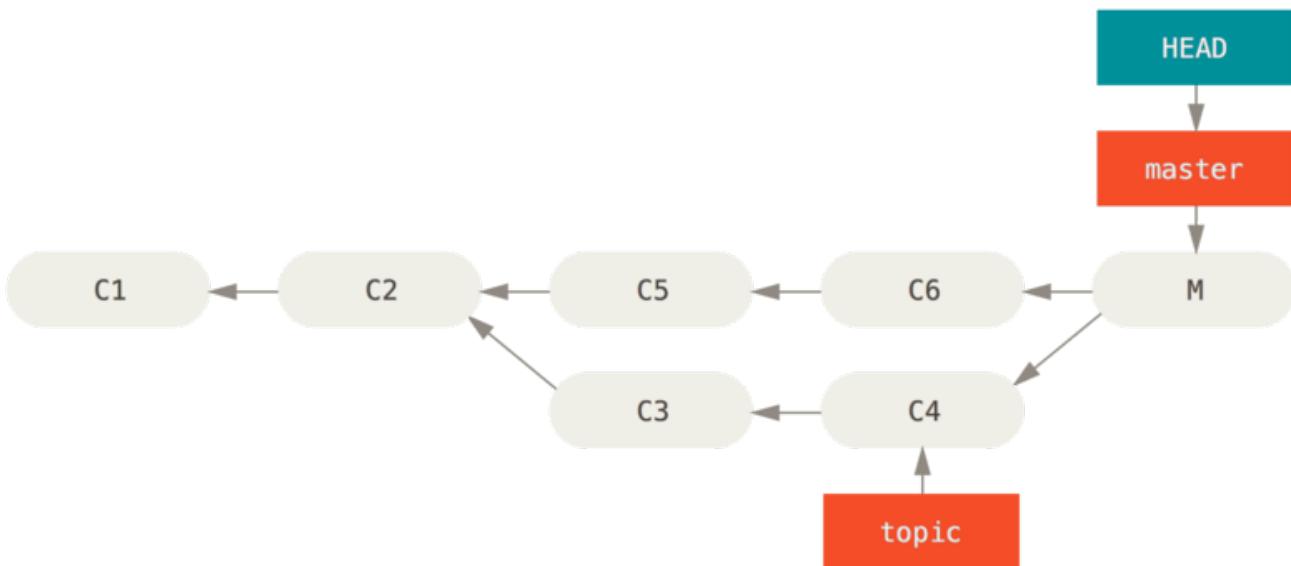
hello()

```

Undoing Merges

Now that you know how to create a merge commit, you'll probably make some by mistake. One of the great things about working with Git is that it's okay to make mistakes, because it's possible (and in many cases easy) to fix them.

Merge commits are no different. Let's say you started work on a topic branch, accidentally merged it into `master`, and now your commit history looks like this:

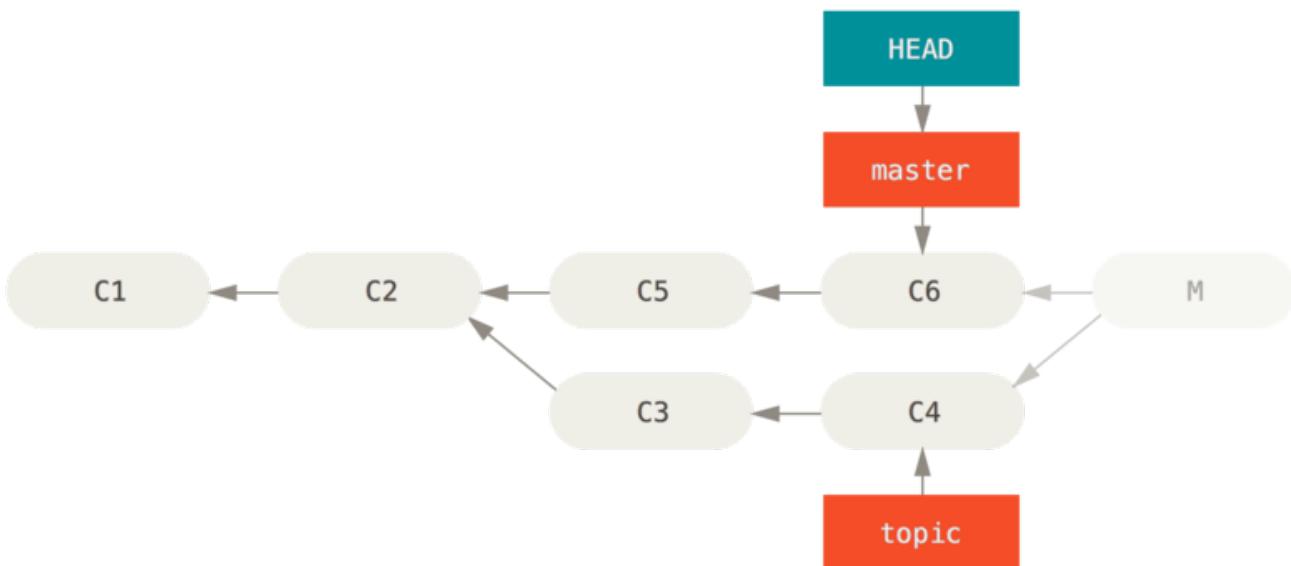


圖表 137. Accidental merge commit

There are two ways to approach this problem, depending on what your desired outcome is.

Fix the references

If the unwanted merge commit only exists on your local repository, the easiest and best solution is to move the branches so that they point where you want them to. In most cases, if you follow the errant `git merge` with `git reset --hard HEAD~`, this will reset the branch pointers so they look like this:



圖表 138. History after `git reset --hard HEAD~`

We covered `reset` back in [Reset Demystified](#), so it shouldn't be too hard to figure out what's going on here. Here's a quick refresher: `reset --hard` usually goes through three steps:

1. Move the branch HEAD points to. In this case, we want to move `master` to where it was before the merge commit (`C6`).
2. Make the index look like HEAD.
3. Make the working directory look like the index.

The downside of this approach is that it's rewriting history, which can be problematic with a shared repository. Check out [使用衍和的危險](#) for more on what can happen; the short version is that if other people have the commits you're rewriting, you should probably avoid `reset`. This approach also won't work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

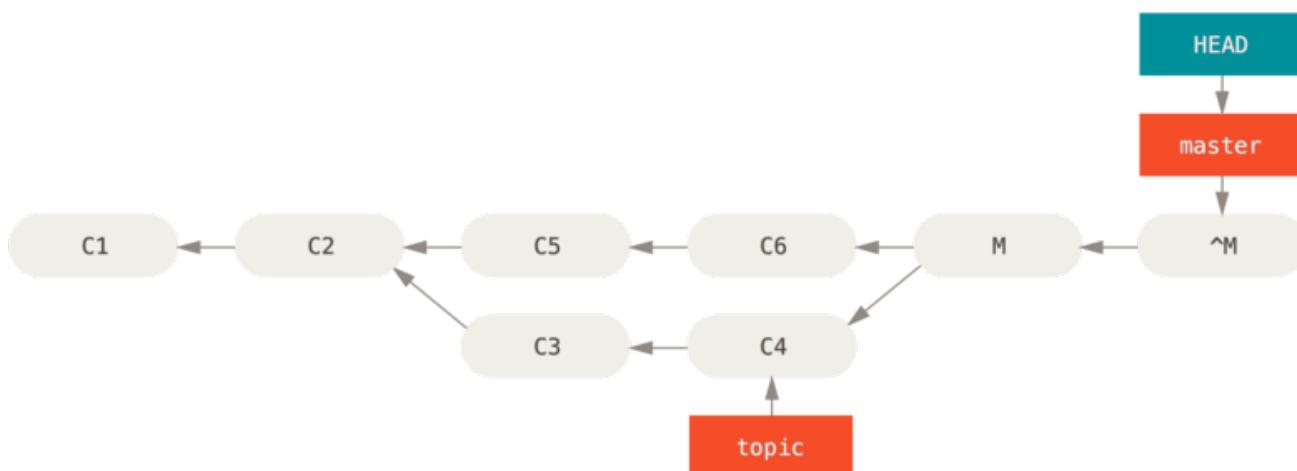
Reverse the commit

If moving the branch pointers around isn't going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a "revert", and in this particular scenario, you'd invoke it like this:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

The `-m 1` flag indicates which parent is the "mainline" and should be kept. When you invoke a merge into `HEAD` (`git merge topic`), the new commit has two parents: the first one is `HEAD` (`C6`), and the second is the tip of the branch being merged in (`C4`). In this case, we want to undo all the changes introduced by merging in parent #2 (`C4`), while keeping all the content from parent #1 (`C6`).

The history with the revert commit looks like this:

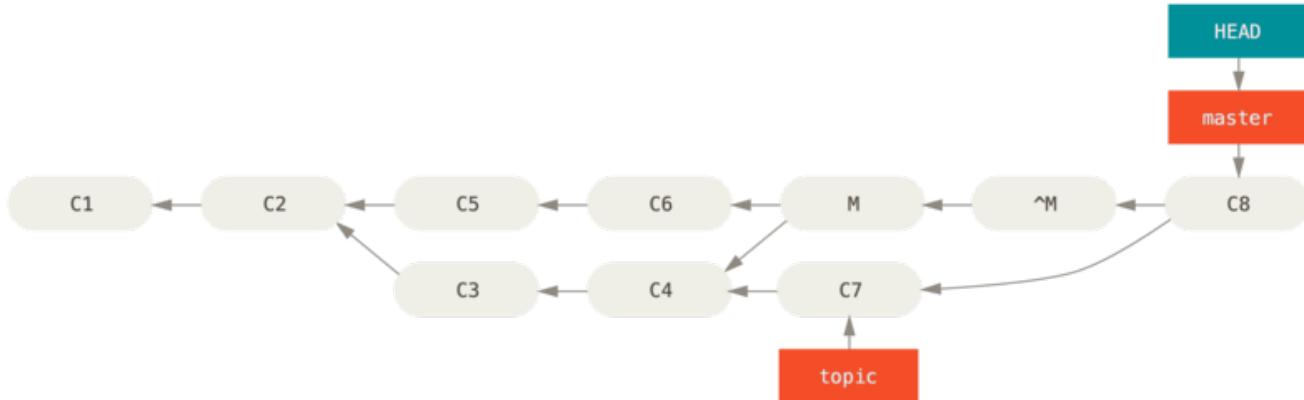


圖表 139. History after `git revert -m 1`

The new commit `^M` has exactly the same contents as `C6`, so starting from here it's as if the merge never happened, except that the now-unmerged commits are still in `HEAD`'s history. Git will get confused if you try to merge `topic` into `master` again:

```
$ git merge topic  
Already up-to-date.
```

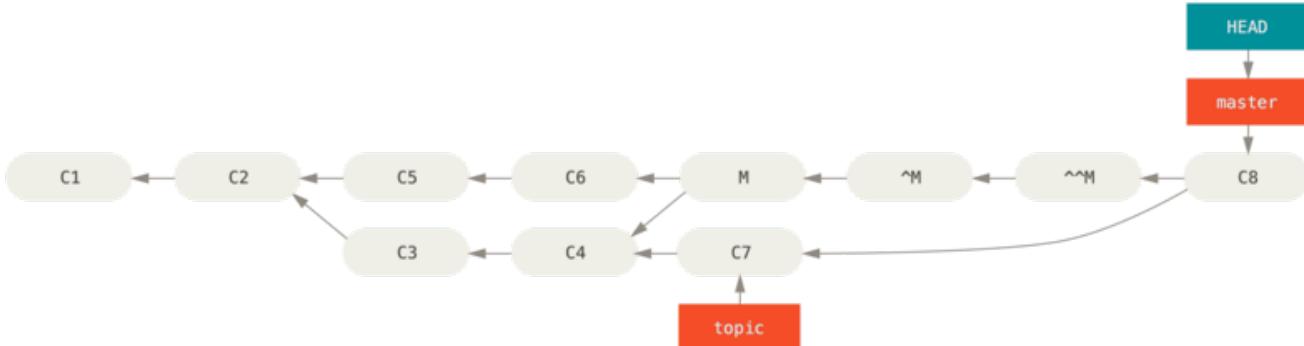
There's nothing in `topic` that isn't already reachable from `master`. What's worse, if you add work to `topic` and merge again, Git will only bring in the changes *since* the reverted merge:



圖表 140. History with a bad merge

The best way around this is to un-revert the original merge, since now you want to bring in the changes that were reverted out, **then** create a new merge commit:

```
$ git revert ^M  
[master 09f0126] Revert "Revert "Merge branch 'topic'"."  
$ git merge topic
```



圖表 141. History after re-merging a reverted merge

In this example, `M` and `^M` cancel out. `^^M` effectively merges in the changes from `C3` and `C4`, and `C8` merges in the changes from `C7`, so now `topic` is fully merged.

Other Types of Merges

So far we've covered the normal merge of two branches, normally handled with what is called the “recursive” strategy of merging. There are other ways to merge branches together however. Let's cover a few of them quickly.

Our or Theirs Preference

First of all, there is another useful thing we can do with the normal “recursive” mode of merging. We’ve already seen the `ignore-all-space` and `ignore-space-change` options which are passed with a `-X` but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually resolve the conflict, you can pass the `merge` command either a `-Xours` or `-Xtheirs`.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

If we go back to the “hello world” example we were using before, we can see that merging in our branch causes conflicts.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

However if we run it with `-Xours` or `-Xtheirs` it does not.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
test.sh  | 2 ++
2 files changed, 3 insertions(+), 1 deletion(-)
create mode 100644 test.sh
```

In that case, instead of getting conflict markers in the file with “hello mundo” on one side and “hola world” on the other, it will simply pick “hola world”. However, all the other non-conflicting changes on that branch are merged successfully in.

This option can also be passed to the `git merge-file` command we saw earlier by running something like `git merge-file --ours` for individual file merges.

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the “ours” merge *strategy*. This is different from the “ours” recursive merge *option*.

This will basically do a fake merge. It will record a new merge commit with both branches as parents, but it will not even look at the branch you’re merging in. It will simply record as the result of the merge the exact code in your current branch.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

You can see that there is no difference between the branch we were on and the result of the merge.

This can often be useful to basically trick Git into thinking that a branch is already merged when doing a merge later on. For example, say you branched off a `release` branch and have done some work on it that you will want to merge back into your `master` branch at some point. In the meantime some bugfix on `master` needs to be backported into your `release` branch. You can merge the bugfix branch into the `release` branch and also `merge -s ours` the same branch into your `master` branch (even though the fix is already there) so when you later merge the `release` branch again, there are no conflicts from the bugfix.

Subtree Merging

The idea of the subtree merge is that you have two projects, and one of the projects maps to a subdirectory of the other one. When you specify a subtree merge, Git is often smart enough to figure out that one is a subtree of the other and merge appropriately.

We'll go through an example of adding a separate project into an existing project and then merging the code of the second into a subdirectory of the first.

First, we'll add the Rack application to our project. We'll add the Rack project as a remote reference in our own project and then check it out into its own branch:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch
refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Now we have the root of the Rack project in our `rack_branch` branch and our own project in the `master` branch. If you check out one and then the other, you can see that they have different project roots:

```
$ ls
AUTHORS      KNOWN-ISSUES   Rakefile     contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

This is sort of a strange concept. Not all the branches in your repository actually have to be branches of the same project. It's not common, because it's rarely helpful, but it's fairly easy to have branches contain completely different histories.

In this case, we want to pull the Rack project into our `master` project as a subdirectory. We can do that in Git with `git read-tree`. You'll learn more about `read-tree` and its friends in [Git Internals](#), but for now know that it reads the root tree of one branch into your current staging area and working directory. We just switched back to your `master` branch, and we pull the `rack_branch` branch into the `rack` subdirectory of our `master` branch of our main project:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

When we commit, it looks like we have all the Rack files under that subdirectory – as though we copied them in from a tarball. What gets interesting is that we can fairly easily merge changes from one of the branches to the other. So, if the Rack project updates, we can pull in upstream changes by switching to that branch and pulling:

```
$ git checkout rack_branch
$ git pull
```

Then, we can merge those changes back into our `master` branch. To pull in the changes and prepopulate the commit message, use the `--squash` option, as well as the recursive merge strategy's `-s -Xsubtree` option. (The recursive strategy is the default here, but we include it for clarity.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

All the changes from the Rack project are merged in and ready to be committed locally. You can also do the opposite – make changes in the `rack` subdirectory of your `master` branch and then merge them into your `rack_branch` branch later to submit them to the maintainers or push them upstream.

This gives us a way to have a workflow somewhat similar to the submodule workflow without using submodules (which we will cover in [Submodules](#)). We can keep branches with other related projects in our repository and subtree merge them into our project occasionally. It is nice in some ways, for example all the code is committed to a single place. However, it has other drawbacks in that it's a bit more complex and easier to make mistakes in reintegrating changes or accidentally pushing a branch

into an unrelated repository.

Another slightly weird thing is that to get a diff between what you have in your `rack` subdirectory and the code in your `rack_branch` branch – to see if you need to merge them – you can’t use the normal `diff` command. Instead, you must run `git diff-tree` with the branch you want to compare to:

```
$ git diff-tree -p rack_branch
```

Or, to compare what is in your `rack` subdirectory with what the `master` branch on the server was the last time you fetched, you can run

```
$ git diff-tree -p rack_remote/master
```

Rerere

The `git rerere` functionality is a bit of a hidden feature. The name stands for “reuse recorded resolution” and as the name implies, it allows you to ask Git to remember how you’ve resolved a hunk conflict so that the next time it sees the same conflict, Git can automatically resolve it for you.

There are a number of scenarios in which this functionality might be really handy. One of the examples that is mentioned in the documentation is if you want to make sure a long lived topic branch will merge cleanly but don’t want to have a bunch of intermediate merge commits. With `rerere` turned on you can merge occasionally, resolve the conflicts, then back out the merge. If you do this continuously, then the final merge should be easy because `rerere` can just do everything for you automatically.

This same tactic can be used if you want to keep a branch rebased so you don’t have to deal with the same rebasing conflicts each time you do it. Or if you want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead – you likely won’t have to do all the same conflicts again.

Another situation is where you merge a bunch of evolving topic branches together into a testable head occasionally, as the Git project itself often does. If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

To enable the `rerere` functionality, you simply have to run this config setting:

```
$ git config --global rerere.enabled true
```

You can also turn it on by creating the `.git/rr-cache` directory in a specific repository, but the config setting is clearer and it can be done globally.

Now let’s see a simple example, similar to our previous one. Let’s say we have a file named `hello.rb` that looks like this:

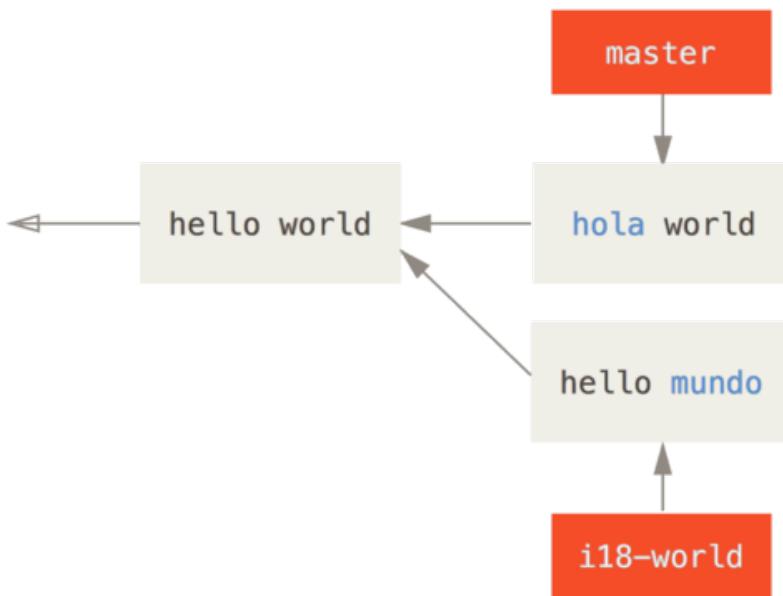
```

#!/usr/bin/env ruby

def hello
  puts 'hello world'
end

```

In one branch we change the word “hello” to “hola”, then in another branch we change the “world” to “mundo”, just like before.



When we merge the two branches together, we’ll get a merge conflict:

```

$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.

```

You should notice the new line **Recorded preimage for FILE** in there. Otherwise it should look exactly like a normal merge conflict. At this point, **rerere** can tell us a few things. Normally, you might run **git status** at this point to see what all conflicted:

```

$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#       both modified:      hello.rb
#

```

However, `git rerere` will also tell you what it has recorded the pre-merge state for with `git rerere status`:

```
$ git rerere status  
hello.rb
```

And `git rerere diff` will show the current state of the resolution - what you started with to resolve and what you've resolved it to.

```
$ git rerere diff  
--- a/hello.rb  
+++ b/hello.rb  
@@ -1,11 +1,11 @@  
#! /usr/bin/env ruby  
  
def hello  
-<<<<<  
- puts 'hello mundo'  
-=====  
+<<<<< HEAD  
    puts 'hola mundo'  
->>>>>  
+=====  
+ puts 'hello mundo'  
+>>>>> i18n-world  
end
```

Also (and this isn't really related to `rerere`), you can use `ls-files -u` to see the conflicted files and the before, left and right versions:

```
$ git ls-files -u  
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb  
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 hello.rb  
100644 54336ba847c3758ab604876419607e9443848474 3 hello.rb
```

Now you can resolve it to just be `puts 'hola mundo'` and you can run the `rerere diff` command again to see what `rerere` will remember:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
#! /usr/bin/env ruby

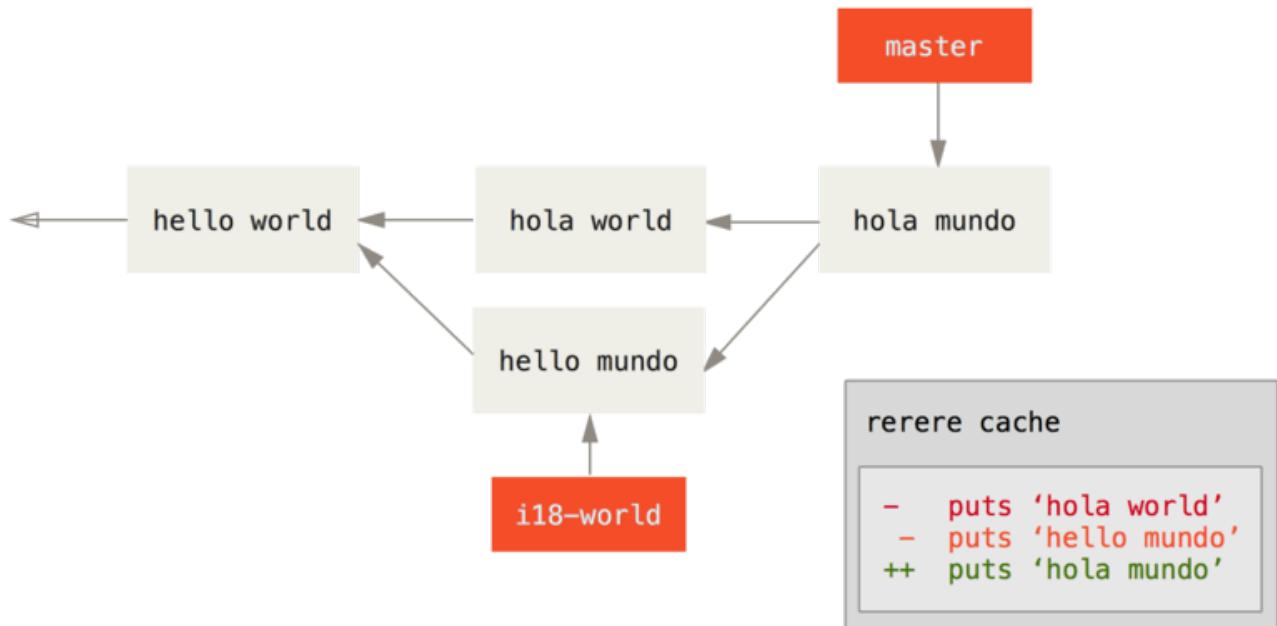
def hello
<<<<<
- puts 'hello mundo'
=====
- puts 'hola world'
->>>>>
+ puts 'hola mundo'
end
```

So that basically says, when Git sees a hunk conflict in a `hello.rb` file that has “hello mundo” on one side and “hola world” on the other, it will resolve it to “hola mundo”.

Now we can mark it as resolved and commit it:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

You can see that it "Recorded resolution for FILE".



Now, let's undo that merge and then rebase it on top of our master branch instead. We can move our branch back by using `reset` as we saw in [Reset Demystified](#).

```
$ git reset --hard HEAD^  
HEAD is now at ad63f15 i18n the hello
```

Our merge is undone. Now let's rebase the topic branch.

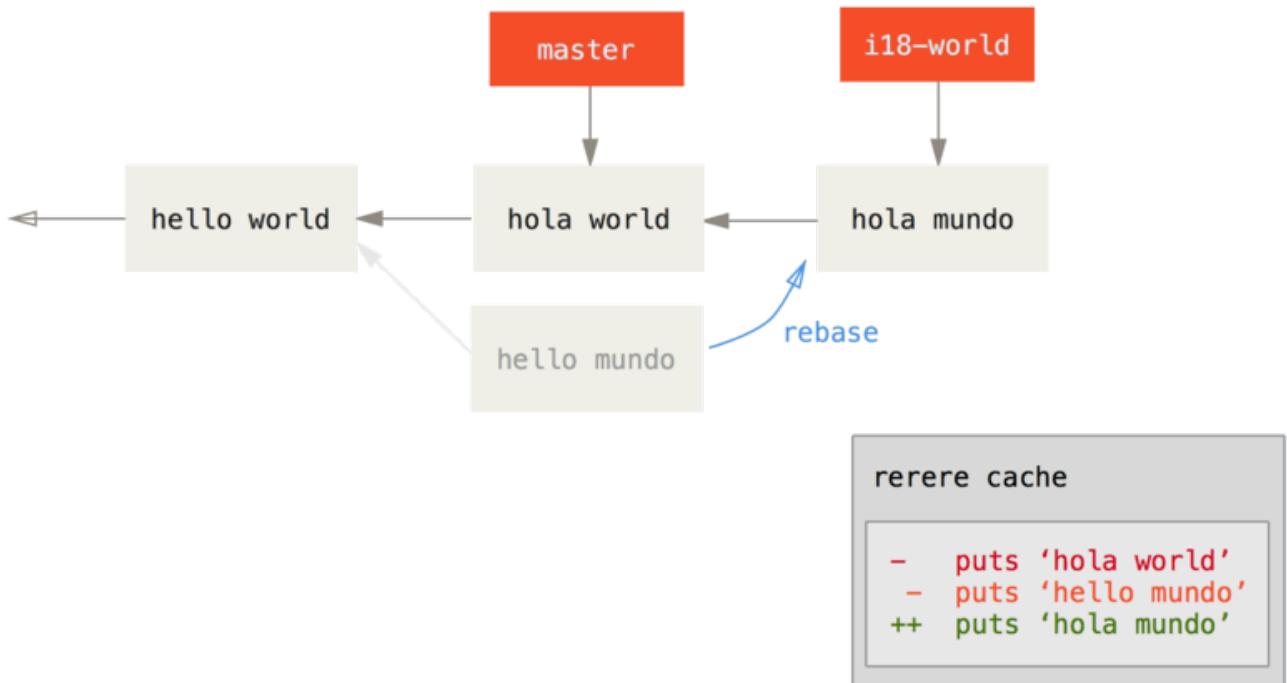
```
$ git checkout i18n-world  
Switched to branch 'i18n-world'  
  
$ git rebase master  
First, rewinding head to replay your work on top of it...  
Applying: i18n one word  
Using index info to reconstruct a base tree...  
Falling back to patching base and 3-way merge...  
Auto-merging hello.rb  
CONFLICT (content): Merge conflict in hello.rb  
Resolved 'hello.rb' using previous resolution.  
Failed to merge in the changes.  
Patch failed at 0001 i18n one word
```

Now, we got the same merge conflict like we expected, but take a look at the **Resolved FILE using previous resolution** line. If we look at the file, we'll see that it's already been resolved, there are no merge conflict markers in it.

```
#!/usr/bin/env ruby  
  
def hello  
  puts 'hola mundo'  
end
```

Also, **git diff** will show you how it was automatically re-resolved:

```
$ git diff  
diff --cc hello.rb  
index a440db6,54336ba..0000000  
--- a/hello.rb  
+++ b/hello.rb  
@@@ -1,7 -1,7 +1,7 @@@  
  #! /usr/bin/env ruby  
  
  def hello  
-   puts 'hola world'  
-   puts 'hello mundo'  
++  puts 'hola mundo'  
  end
```



You can also recreate the conflicted file state with the `checkout` command:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> theirs
end
```

We saw an example of this in [Advanced Merging](#). For now though, let's re-resolve it by just running `rerere` again:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

We have re-resolved the file automatically using the `rerere` cached resolution. You can now add and continue the rebase to complete it.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

So, if you do a lot of re-merges, or want to keep a topic branch up to date with your master branch without a ton of merges, or you rebase often, you can turn on `rerere` to help your life out a bit.

Debugging with Git

Git also provides a couple of tools to help you debug issues in your projects. Because Git is designed to work with nearly any type of project, these tools are pretty generic, but they can often help you hunt for a bug or culprit when things go wrong.

File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows you what commit was the last to modify each line of any file. So, if you see that a method in your code is buggy, you can annotate the file with `git blame` to see when each line of the method was last edited and by whom. This example uses the `-L` option to limit the output to lines 12 through 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree =
'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git
show #{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree =
'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log
#{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git
blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Notice that the first field is the partial SHA-1 of the commit that last modified that line. The next two fields are values extracted from that commit—the author name and the authored date of that commit—so you can easily see who modified that line and when. After that come the line number and the content of the file. Also note the `^4832fe2` commit lines, which designate that those lines were in this file's original commit. That commit is when this file was first added to this project, and those lines have been unchanged since. This is a tad confusing, because now you've seen at least three different ways that Git uses the `^` to modify a commit SHA-1, but that is what it means here.

Another cool thing about Git is that it doesn't track file renames explicitly. It records the snapshots

and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass `-C` to `git blame`, Git analyzes the file you’re annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. For example, say you are refactoring a file named `GITServerHandler.m` into multiple files, one of which is `GITPackUpload.m`. By blaming `GITPackUpload.m` with the `-C` option, you can see where sections of the code originally came from:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void)
gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)
//NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 146)      NSString
*parentSha;
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 147)      GITCommit
*commit = [g
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 149)
//NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m      (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)      if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)
[refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

This is really useful. Normally, you get as the original commit the commit where you copied the code over, because that is the first time you touched those lines in this file. Git tells you the original commit where you wrote those lines, even if it was in another file.

Binary Search

Annotating a file helps if you know where the issue is to begin with. If you don’t know what is breaking, and there have been dozens or hundreds of commits since the last state where you know the code worked, you’ll likely turn to `git bisect` for help. The `bisect` command does a binary search through your commit history to help you identify as quickly as possible which commit introduced an issue.

Let’s say you just pushed out a release of your code to a production environment, you’re getting bug reports about something that wasn’t happening in your development environment, and you can’t imagine why the code is doing that. You go back to your code, and it turns out you can reproduce the issue, but you can’t figure out what is going wrong. You can bisect the code to find out. First you run `git bisect start` to get things going, and then you use `git bisect bad` to tell the system that the current commit you’re on is broken. Then, you must tell bisect when the last known good state was, using `git bisect good [good_commit]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git figured out that about 12 commits came between the commit you marked as the last good commit (v1.0) and the current bad version, and it checked out the middle one for you. At this point, you can run your test to see if the issue exists as of this commit. If it does, then it was introduced sometime before this middle commit; if it doesn't, then the problem was introduced sometime after the middle commit. It turns out there is no issue here, and you tell Git that by typing `git bisect good` and continue your journey:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Now you're on another commit, halfway between the one you just tested and your bad commit. You run your test again and find that this commit is broken, so you tell Git that with `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

This commit is fine, and now Git has all the information it needs to determine where the issue was introduced. It tells you the SHA-1 of the first bad commit and show some of the commit information and which files were modified in that commit so you can figure out what happened that may have introduced this bug:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

When you're finished, you should run `git bisect reset` to reset your HEAD to where you were before you started, or you'll end up in a weird state:

```
$ git bisect reset
```

This is a powerful tool that can help you check hundreds of commits for an introduced bug in minutes. In fact, if you have a script that will exit 0 if the project is good or non-0 if the project is bad, you can fully automate `git bisect`. First, you again tell it the scope of the bisect by providing the known bad and good commits. You can do this by listing them with the `bisect start` command if you want, listing the known bad commit first and the known good commit second:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Doing so automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. You can also run something like `make` or `make tests` or whatever you have that runs automated tests for you.

Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Here's an example. Suppose you're developing a web site and creating Atom feeds. Instead of writing your own Atom-generating code, you decide to use a library. You're likely to have to either include this code from a shared library like a CPAN install or Ruby gem, or copy the source code into your own project tree. The issue with including the library is that it's difficult to customize the library in any way and often more difficult to deploy it, because you need to make sure every client has that library available. The issue with vendorizing the code into your own project is that any custom changes you make are difficult to merge when upstream changes become available.

Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

Starting with Submodules

We'll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let's start by adding an existing Git repository as a submodule of the repository that we're working on. To add a new submodule you use the `git submodule add` command with the absolute or relative URL of the project you would like to start tracking. In this example, we'll add a library called "DbConnector".

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

By default, submodules will add the subproject into a directory named the same as the repository, in this case “DbConnector” . You can add a different path at the end of the command if you want it to go elsewhere.

If you run `git status` at this point, you’ll notice a few things.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   DbConnector
```

First you should notice the new `.gitmodules` file. This is a configuration file that stores the mapping between the project’s URL and the local subdirectory you’ve pulled it into:

```
[submodule "DbConnector"]
  path = DbConnector
  url = https://github.com/chaconinc/DbConnector
```

If you have multiple submodules, you’ll have multiple entries in this file. It’s important to note that this file is version-controlled with your other files, like your `.gitignore` file. It’s pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

筆記

Since the URL in the `.gitmodules` file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would pull from, use the one that others have access to. You can overwrite this value locally with `git config submodule.DbConnector.url PRIVATE_URL` for your own use. When applicable, a relative URL can be helpful.

The other listing in the `git status` output is the project folder entry. If you run `git diff` on that, you see something interesting:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Although **DbConnector** is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

If you want a little nicer diff output, you can pass the **--submodule** option to `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+    path = DbConnector
+    url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000..c3f01dc (new submodule)
```

When you commit, you see something like this:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Notice the **160000** mode for the **DbConnector** entry. That is a special mode in Git that basically means you're recording a commit as a directory entry rather than a subdirectory or a file.

Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

```

$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x  9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x  7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--  1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x  2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--  1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x  3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

The `DbConnector` directory is there, but empty. You must run two commands: `git submodule init` to initialize your local configuration file, and `git submodule update` to fetch all the data from that project and check out the appropriate commit listed in your superproject:

```

$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector)
registered for path 'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Now your `DbConnector` subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass `--recursive` to the `git clone` command, it will automatically initialize and update each submodule in the repository.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector)
registered for path 'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Working on a Project with Submodules

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

Pulling in Upstream Changes

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let's walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run `git fetch` and `git merge` the upstream branch to update the local code.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
  c3f01dc..d0354fc  master      -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
  scripts/connect.sh | 1 +
  src/db.c          | 1 +
  2 files changed, 2 insertions(+)
```

Now if you go back into the main project and run `git diff --submodule` you can see that the submodule was updated and get a list of commits that were added to it. If you don't want to type `--submodule` every time you run `git diff`, you can set it as the default format by setting the `diff.submodule` config value to "log".

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run `git submodule update --remote`, Git will go into your submodules and fetch and update for you.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
  3f19983..d0354fc  master      -> origin/master
Submodule path 'DbConnector': checked out
'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

This command will by default assume that you want to update the checkout to the `master` branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository's "stable" branch, you can set it in either your `.gitmodules` file (so everyone else also tracks it), or just in your local `.git/config` file. Let's set it in the `.gitmodules` file:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
  27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out
'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

If you leave off the `-f .gitmodules` it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run `git status` at this point, Git will show us that we have "new commits" on the submodule.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working
     directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

If you set the configuration setting `status.submodulesummary`, Git will also show you a short summary of changes to your submodules:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working
     directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

At this point if you run `git diff` we can see both that we have modified our `.gitmodules` file and also that there are a number of commits that we've pulled down and are ready to commit to our submodule project.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
> better connection routine
```

This is pretty cool as we can actually see the log of commits that we’re about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run `git log -p`.

```
$ git log -p --submodule
commit 0a24cf121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
> better connection routine
```

Git will by default try to update **all** of your submodules when you run `git submodule update --remote` so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

Working on a Submodule

It’s quite likely that if you’re using submodules, you’re doing so because you really want to work on the code in the submodule at the same time as you’re working on the code in the main project (or

across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let's go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we've run the `git submodule update` command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what's called a "detached HEAD" state. This means that there is no local working branch (like "master", for example) tracking changes. With no working branch tracking changes, that means even if you commit changes to the submodule, those changes will quite possibly be lost the next time you run `git submodule update`. You have to do some extra steps if you want changes in a submodule to be tracked.

In order to set up your submodule to be easier to go in and hack on, you need to do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then `git submodule update --remote` pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let's go into our submodule directory and check out a branch.

```
$ git checkout stable  
Switched to branch 'stable'
```

Let's try it with the "merge" option. To specify it manually, we can just add the `--merge` option to our `update` call. Here we'll see that there was a change on the server for this submodule and it gets merged in.

```
$ git submodule update --remote --merge  
remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 4 (delta 2), reused 4 (delta 2)  
Unpacking objects: 100% (4/4), done.  
From https://github.com/chaconinc/DbConnector  
  c87d55d..92c7337 stable      -> origin/stable  
Updating c87d55d..92c7337  
Fast-forward  
  src/main.c | 1 +  
   1 file changed, 1 insertion(+)  
Submodule path 'DbConnector': merged in  
'92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

If we go into the `DbConnector` directory, we have the new changes already merged into our local `stable` branch. Now let's see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

```
$ cd DbConnector/  
$ vim src/db.c  
$ git commit -am 'unicode support'  
[stable f906e16] unicode support  
1 file changed, 1 insertion(+)
```

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

```
$ git submodule update --remote --rebase  
First, rewinding head to replay your work on top of it...  
Applying: unicode support  
Submodule path 'DbConnector': rebased into  
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If you forget the `--rebase` or `--merge`, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

```
$ git submodule update --remote  
Submodule path 'DbConnector': checked out  
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If this happens, don't worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase `origin/stable` (or whatever remote branch you want) manually.

If you haven't committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

```
$ git submodule update --remote  
remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 4 (delta 0), reused 4 (delta 0)  
Unpacking objects: 100% (4/4), done.  
From https://github.com/chaconinc/DbConnector  
 5d60ef9..c75e92a stable    -> origin/stable  
error: Your local changes to the following files would be overwritten by  
checkout:  
  scripts/setup.sh  
Please, commit your changes or stash them before you can switch  
branches.  
Aborting  
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in  
submodule path 'DbConnector'
```

If you made changes that conflict with something changed upstream, Git will let you know when you

run the update.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule
path 'DbConnector'
```

You can go into the submodule directory and fix the conflict just as you normally would.

Publishing Submodule Changes

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren't available to anyone else yet as we haven't pushed them yet.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
> Merge from origin/stable
> updated setup script
> unicode support
> remove unnecessary method
> add new option for conn pooling
```

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn't happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The `git push` command takes the `--recurse-submodules` argument which can be set to either "check" or "on-demand". The "check" option will make `push` simply fail if any of the committed submodule changes haven't been pushed.

```
$ git push --recurse-submodules=check  
The following submodule paths contain changes that can  
not be found on any remote:  
  DbConnector  
  
Please try  
  
  git push --recurse-submodules=on-demand  
  
or cd to the path and use  
  
  git push  
  
to push them to a remote.
```

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they’re externally available and then try this push again.

The other option is to use the “on-demand” value, which will try to do this for you.

```
$ git push --recurse-submodules=on-demand  
Pushing submodule 'DbConnector'  
Counting objects: 9, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (8/8), done.  
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.  
Total 9 (delta 3), reused 0 (delta 0)  
To https://github.com/chaconinc/DbConnector  
  c75e92a..82d2ad3  stable -> stable  
Counting objects: 2, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.  
Total 2 (delta 1), reused 0 (delta 0)  
To https://github.com/chaconinc/MainProject  
  3d6d338..9a377d1  master -> master
```

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail.

Merging Submodule Changes

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
  9a377d1..eb974f8  master      -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits
not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

So basically what has happened here is that Git has figured out that the two branches record points in the submodule’s history that are divergent and need to be merged. It explains it as “merge following commits not found”, which is confusing but we’ll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn’t really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it’s simple to figure out. If you run `git diff` you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

So, in this case, `eb41d76` is the commit in our submodule that **we** had and `c771610` is the commit that upstream had. If we go into our submodule directory, it should already be on `eb41d76` as the merge would not have touched it. If for whatever reason it’s not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you’ll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch based on that second SHA-1 from `git diff` and manually merge.

```

$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.

```

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

```

$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
- Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes

```

① First we resolve the conflict

② Then we go back to the main project directory

③ We can check the SHA-1s again

④ Resolve the conflicted submodule entry

⑤ Commit our merge

It can be a bit confusing, but it's really not very hard.

Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone merged branches containing these two commits, so maybe you'll want that one.

This is why the error message from before was “merge following commits not found”, because it could not do **this**. It’s confusing because who would expect it to **try** to do this?

If it does find a single acceptable merge commit, you’ll see something like this:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
  9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000
9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

What it’s suggesting that you do is to update the index like you had run **git add**, which clears the conflict, then commit. You probably shouldn’t do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you’re done.

Submodule Tips

There are a few things you can do to make working with submodules a little easier.

Submodule Foreach

There is a **foreach** submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

For example, let’s say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

```
$ git submodule foreach 'git stash'  
Entering 'CryptoLibrary'  
No local changes to save  
Entering 'DbConnector'  
Saved working directory and index state WIP on stable: 82d2ad3 Merge  
from origin/stable  
HEAD is now at 82d2ad3 Merge from origin/stable
```

Then we can create a new branch and switch to it in all our submodules.

```
$ git submodule foreach 'git checkout -b featureA'  
Entering 'CryptoLibrary'  
Switched to a new branch 'featureA'  
Entering 'DbConnector'  
Switched to a new branch 'featureA'
```

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

```

$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char
***argv)

    commit_page_choice();

+    url = url_decode(url_orig);
+
    /* build alias_argv */
    alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
    alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
    return url_decode_internal(&url, len, NULL, &out, 0);
}

+char *url_decode(const char *url)
+{
+    return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
    struct strbuf out = STRBUF_INIT;

```

Here we can see that we’re defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

Useful Aliases

You may want to set up some aliases for some of these commands as they can be quite long and you can’t set configuration options for most of them to make them defaults. We covered setting up Git aliases in [Git Aliases](#), but here is an example of what you may want to set up if you plan on working with submodules in Git a lot.

```

$ git config alias.sdiff '!' "git diff && git submodule foreach 'git
diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'

```

This way you can simply run `git supdate` when you want to update your submodules, or `git spush`

to push with submodule dependency checking.

Issues with Submodules

Using submodules isn't without hiccups, however.

For instance switching branches with submodules in them can also be tricky. If you create a new branch, add a submodule there, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    CryptoLibrary/

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

Removing the directory isn't difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run `submodule update --init` to repopulate it.

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/
$ git submodule update --init
Submodule path 'CryptoLibrary': checked out
'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile      includes      scripts      src
```

Again, not really very difficult, but it can be a little confusing.

The other main caveat that many people run into involves switching from subdirectories to submodules. If you've been tracking files in your project and you want to move them out into a submodule, you must be careful or Git will get angry at you. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. If you delete the subdirectory and then run `submodule add`, Git yells at you:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

You have to unstage the `CryptoLibrary` directory first. Then you can add the submodule:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Now suppose you did that in a branch. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

```
$ git checkout master
error: The following untracked working tree files would be overwritten
by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

You can force it to switch with `checkout -f`, but be careful that you don't have unsaved changes in there as they could be overwritten with that command.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Then, when you switch back, you get an empty `CryptoLibrary` directory for some reason and `git submodule update` may not fix it either. You may need to go into your submodule directory and run a `git checkout .` to get all your files back. You could run this in a `submodule foreach` script to run it for multiple submodules.

It's important to note that submodules these days keep all their Git data in the top project's `.git` directory, so unlike much older versions of Git, destroying a submodule directory won't lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.

Bundling

Though we've covered the common ways to transfer Git data over a network (HTTP, SSH, etc), there is actually one more way to do so that is not commonly used but can actually be quite useful.

Git is capable of "bundling" its data into a single file. This can be useful in various scenarios. Maybe your network is down and you want to send changes to your co-workers. Perhaps you're working somewhere offsite and don't have access to the local network for security reasons. Maybe your wireless/ethernet card just broke. Maybe you don't have access to a shared server for the moment, you want to email someone updates and you don't want to transfer 40 commits via `format-patch`.

This is where the `git bundle` command can be helpful. The `bundle` command will package up everything that would normally be pushed over the wire with a `git push` command into a binary file that you can email to someone or put on a flash drive, then unbundle into another repository.

Let's see a simple example. Let's say you have a repository with two commits:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

If you want to send that repository to someone and you don't have access to a repository to push to, or simply don't want to set one up, you can bundle it with `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Now you have a file named `repo.bundle` that has all the data needed to re-create the repository's `master` branch. With the `bundle` command you need to list out every reference or specific range of commits that you want to be included. If you intend for this to be cloned somewhere else, you should add `HEAD` as a reference as well as we've done here.

You can email this `repo.bundle` file to someone else, or put it on a USB drive and walk it over.

On the other side, say you are sent this `repo.bundle` file and want to work on the project. You can clone from the binary file into a directory, much like you would from a URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

If you don't include `HEAD` in the references, you have to also specify `-b master` or whatever branch is included because otherwise it won't know what branch to check out.

Now let's say you do three commits on it and want to send the new commits back via a bundle on a USB stick or email.

```
$ git log --oneline  
71b84da last commit - second repo  
c99cf5b fourth commit - second repo  
7011d3d third commit - second repo  
9a466c5 second commit  
b1ec324 first commit
```

First we need to determine the range of commits we want to include in the bundle. Unlike the network protocols which figure out the minimum set of data to transfer over the network for us, we'll have to figure this out manually. Now, you could just do the same thing and bundle the entire repository, which will work, but it's better to just bundle up the difference - just the three commits we just made locally.

In order to do that, you'll have to calculate the difference. As we described in [Commit Ranges](#), you can specify a range of commits in a number of ways. To get the three commits that we have in our master branch that weren't in the branch we originally cloned, we can use something like `origin/master..master` or `master ^origin/master`. You can test that with the `log` command.

```
$ git log --oneline master ^origin/master  
71b84da last commit - second repo  
c99cf5b fourth commit - second repo  
7011d3d third commit - second repo
```

So now that we have the list of commits we want to include in the bundle, let's bundle them up. We do that with the `git bundle create` command, giving it a filename we want our bundle to be and the range of commits we want to go into it.

```
$ git bundle create commits.bundle master ^9a466c5  
Counting objects: 11, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (9/9), 775 bytes, done.  
Total 9 (delta 0), reused 0 (delta 0)
```

Now we have a `commits.bundle` file in our directory. If we take that and send it to our partner, she can then import it into the original repository, even if more work has been done there in the meantime.

When she gets the bundle, she can inspect it to see what it contains before she imports it into her repository. The first command is the `bundle verify` command that will make sure the file is actually a valid Git bundle and that you have all the necessary ancestors to reconstitute it properly.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

If the bundler had created a bundle of just the last two commits they had done, rather than all three, the original repository would not be able to import it, since it is missing requisite history. The `verify` command would have looked like this instead:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second
repo
```

However, our first bundle is valid, so we can fetch in commits from it. If you want to see what branches are in the bundle that can be imported, there is also a command to just list the heads:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

The `verify` sub-command will tell you the heads as well. The point is to see what can be pulled in, so you can use the `fetch` or `pull` commands to import commits from this bundle. Here we'll fetch the *master* branch of the bundle to a branch named *other-master* in our repository:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master      -> other-master
```

Now we can see that we have the imported commits on the *other-master* branch as well as any commits we've done in the meantime in our own *master* branch.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

So, `git bundle` can be really useful for sharing or doing network-type operations when you don't have the proper network or shared repository to do so.

Replace

Git's objects are unchangeable, but it does provide an interesting way to pretend to replace objects in its database with other objects.

The `replace` command lets you specify an object in Git and say "every time you see this, pretend it's this other thing". This is most commonly useful for replacing one commit in your history with another one.

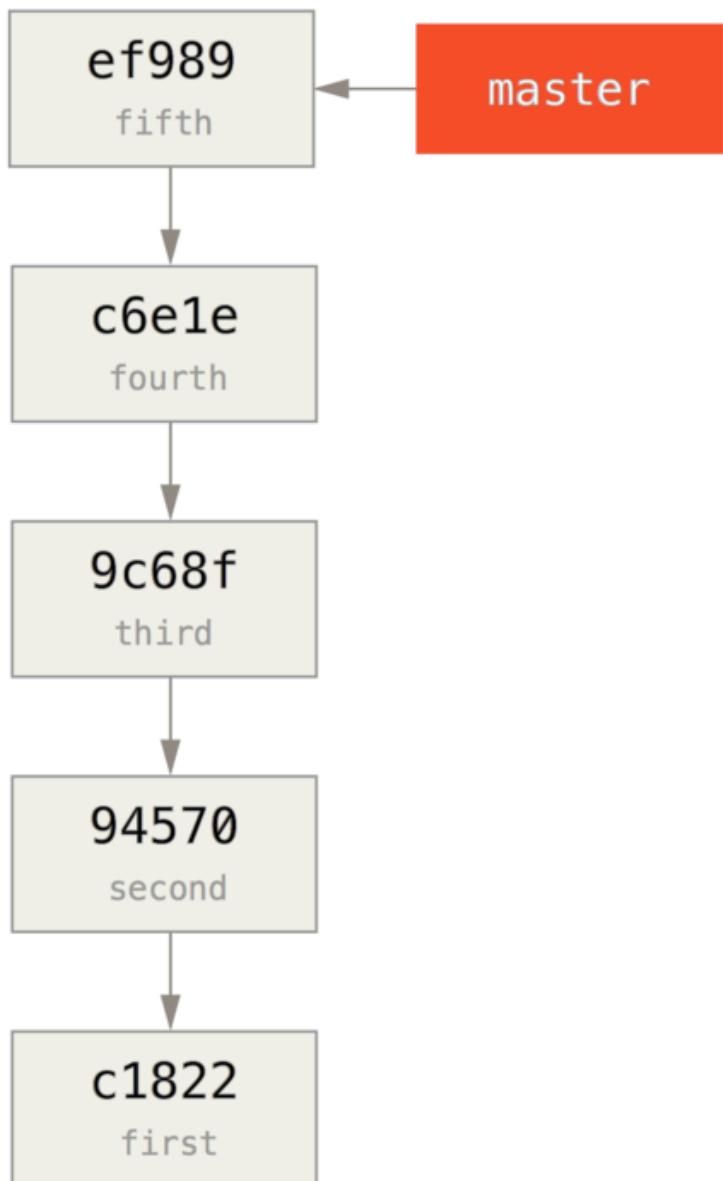
For example, let's say you have a huge code history and want to split your repository into one short history for new developers and one much longer and larger history for people interested in data mining. You can graft one history onto the other by `replace`ing the earliest commit in the new line with the latest commit on the older one. This is nice because it means that you don't actually have to rewrite every commit in the new history, as you would normally have to do to join them together (because the parentage affects the SHA-1s).

Let's try this out. Let's take an existing repository, split it into two repositories, one recent and one historical, and then we'll see how we can recombine them without modifying the recent repositories SHA-1 values via `replace`.

We'll use a simple repository with five simple commits:

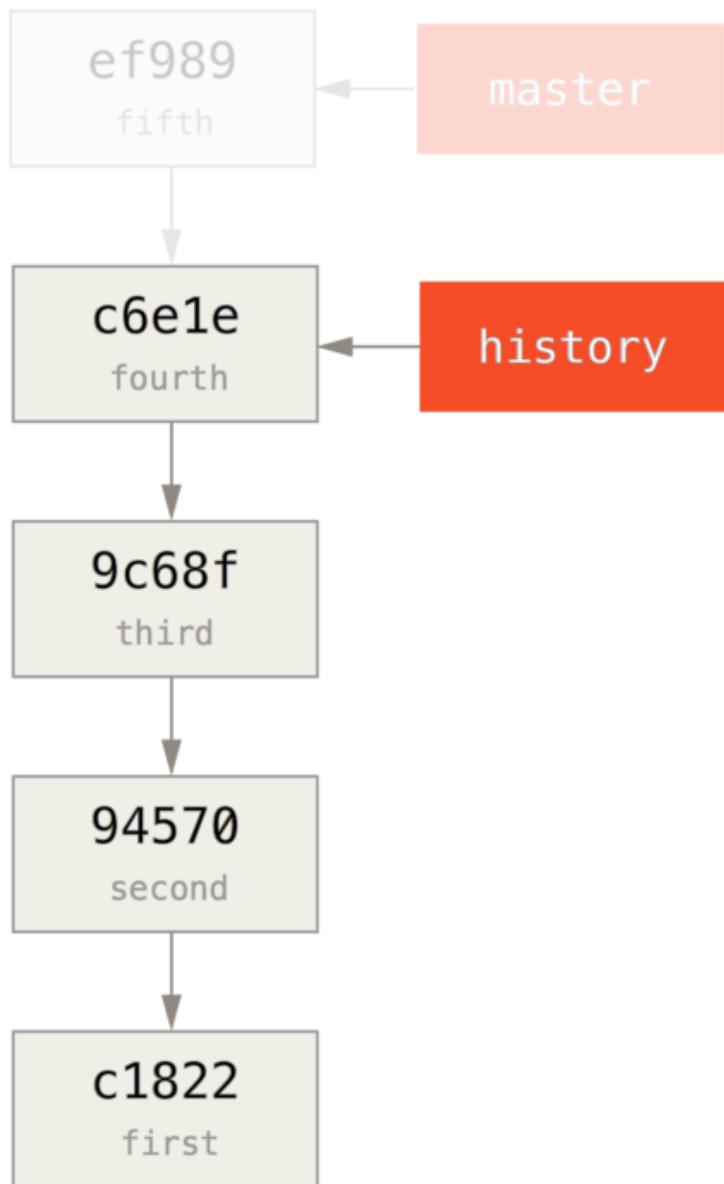
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

We want to break this up into two lines of history. One line goes from commit one to commit four - that will be the historical one. The second line will just be commits four and five - that will be the recent history.



Well, creating the historical history is easy, we can just put a branch in the history and then push that branch to the master branch of a new remote repository.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



Now we can push the new `history` branch to the `master` branch of our new repository:

```

$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master

```

OK, so our history is published. Now the harder part is truncating our recent history down so it's

smaller. We need an overlap so we can replace a commit in one with an equivalent commit in the other, so we're going to truncate this to just commits four and five (so commit four overlaps).

```
$ git log --oneline --decorate  
ef989d8 (HEAD, master) fifth commit  
c6e1e95 (history) fourth commit  
9c68fdc third commit  
945704c second commit  
c1822cf first commit
```

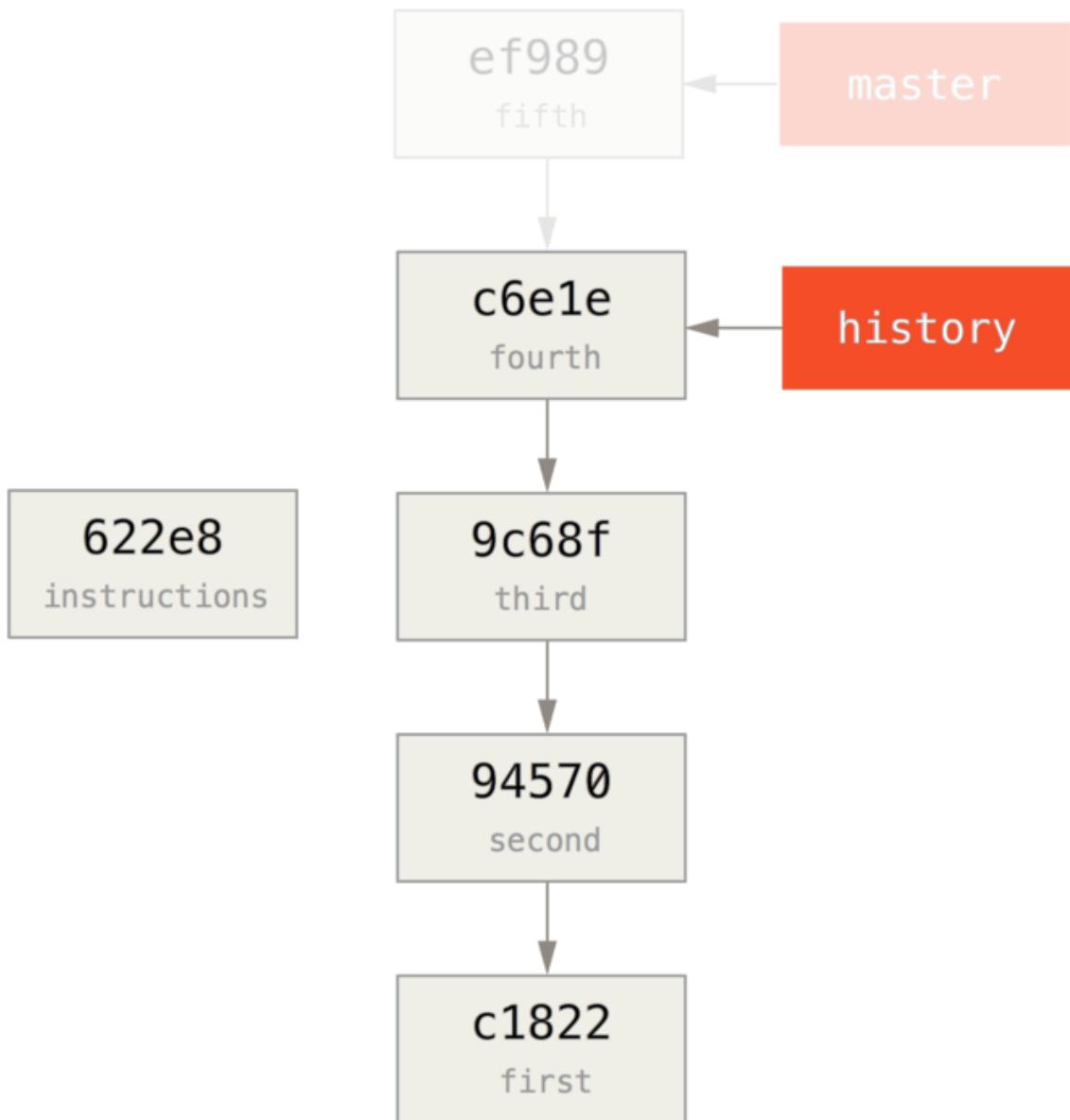
It's useful in this case to create a base commit that has instructions on how to expand the history, so other developers know what to do if they hit the first commit in the truncated history and need more. So, what we're going to do is create an initial commit object as our base point with instructions, then rebase the remaining commits (four and five) on top of it.

To do that, we need to choose a point to split at, which for us is the third commit, which is **9c68fdc** in SHA-speak. So, our base commit will be based off of that tree. We can create our base commit using the **commit-tree** command, which just takes a tree and will give us a brand new, parentless commit object SHA-1 back.

```
$ echo 'get history from blah blah blah' | git commit-tree  
9c68fdc^{tree}  
622e88e9cbfbacfb75b5279245b9fb38dfa10cf
```

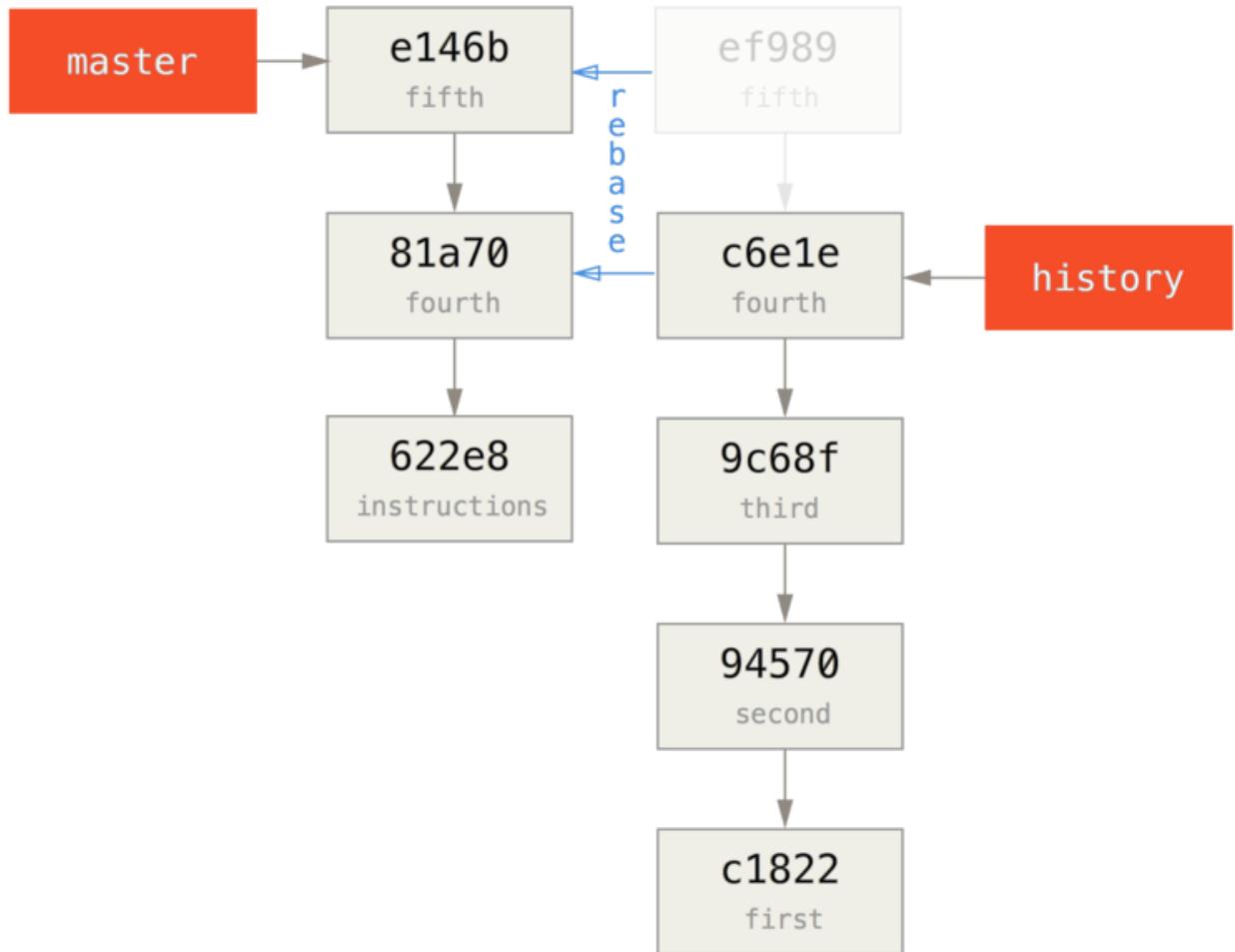
筆記

The **commit-tree** command is one of a set of commands that are commonly referred to as *plumbing* commands. These are commands that are not generally meant to be used directly, but instead are used by **other** Git commands to do smaller jobs. On occasions when we're doing weirder things like this, they allow us to do really low-level things but are not meant for daily use. You can read more about plumbing commands in [Plumbing and Porcelain](#)



OK, so now that we have a base commit, we can rebase the rest of our history on top of that with `git rebase --onto`. The `--onto` argument will be the SHA-1 we just got back from `commit-tree` and the rebase point will be the third commit (the parent of the first commit we want to keep, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



OK, so now we've re-written our recent history on top of a throw away base commit that now has instructions in it on how to reconstitute the entire history if we wanted to. We can push that new history to a new project and now when people clone that repository, they will only see the most recent two commits and then a base commit with instructions.

Let's now switch roles to someone cloning the project for the first time who wants the entire history. To get the history data after cloning this truncated repository, one would have to add a second remote for the historical repository and fetch:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-
history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master      -> project-history/master
```

Now the collaborator would have their recent commits in the `master` branch and the historical commits in the `project-history/master` branch.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

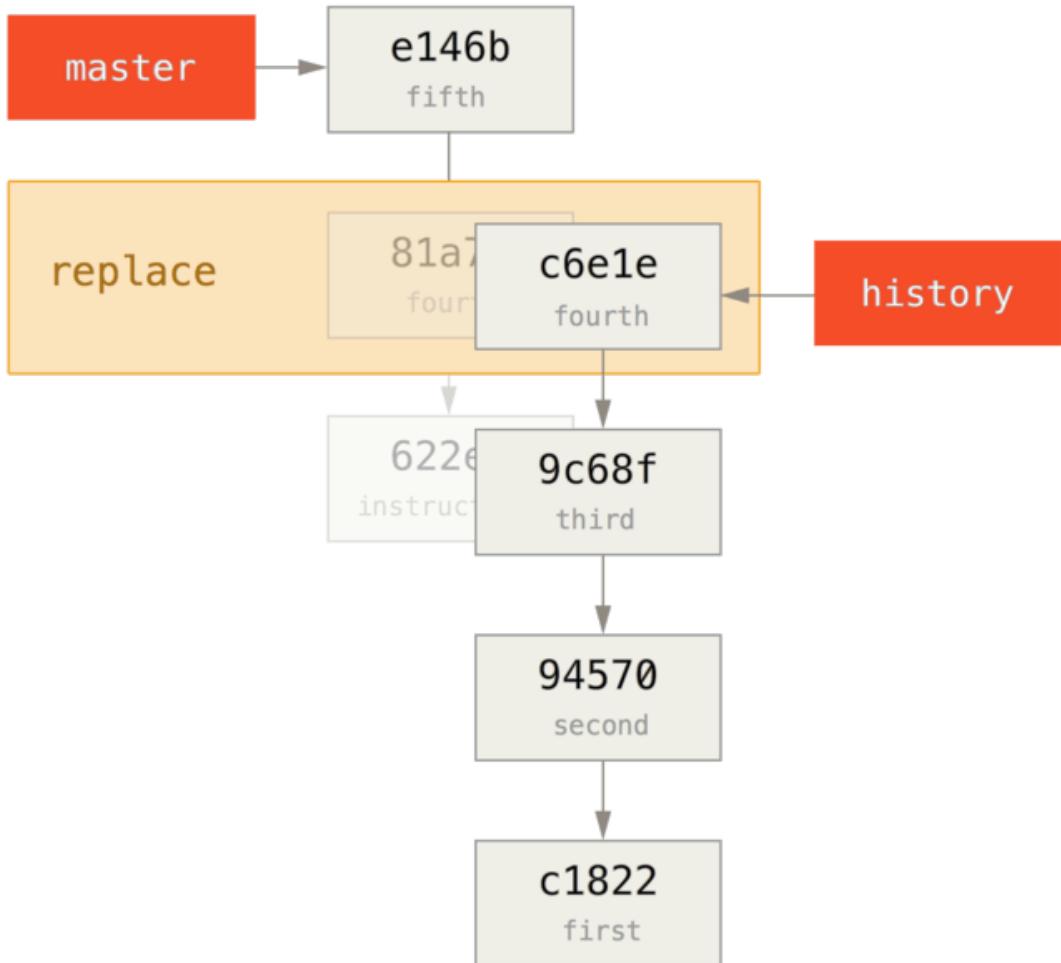
To combine them, you can simply call `git replace` with the commit you want to replace and then the commit you want to replace it with. So we want to replace the "fourth" commit in the `master` branch with the "fourth" commit in the `project-history/master` branch:

```
$ git replace 81a708d c6e1e95
```

Now, if you look at the history of the `master` branch, it appears to look like this:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Cool, right? Without having to change all the SHA-1s upstream, we were able to replace one commit in our history with an entirely different commit and all the normal tools (`bisect`, `blame`, etc) will work how we would expect them to.



Interestingly, it still shows **81a708d** as the SHA-1, even though it's actually using the **c6e1e95** commit data that we replaced it with. Even if you run a command like **cat-file**, it will show you the replaced data:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Remember that the actual parent of **81a708d** was our placeholder commit (**622e88e**), not **9c68fdce** as it states here.

Another interesting thing is that this data is kept in our references:

```
$ git for-each-ref  
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master  
c6e1e95051d41771a649f3145423f8809d1a74d4 commit  
refs/remotes/history/master  
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD  
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit  
refs/remotes/origin/master  
c6e1e95051d41771a649f3145423f8809d1a74d4 commit  
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

This means that it's easy to share our replacement with others, because we can push this to our server and other people can easily download it. This is not that helpful in the history grafting scenario we've gone over here (since everyone would be downloading both histories anyhow, so why separate them?) but it can be useful in other circumstances.

Credential Storage

If you use the SSH transport for connecting to remotes, it's possible for you to have a key without a passphrase, which allows you to securely transfer data without typing in your username and password. However, this isn't possible with the HTTP protocols – every connection needs a username and password. This gets even harder for systems with two-factor authentication, where the token you use for a password is randomly generated and unpronounceable.

Fortunately, Git has a credentials system that can help with this. Git has a few options provided in the box:

- The default is not to cache at all. Every connection will prompt you for your username and password.
- The “cache” mode keeps credentials in memory for a certain period of time. None of the passwords are ever stored on disk, and they are purged from the cache after 15 minutes.
- The “store” mode saves the credentials to a plain-text file on disk, and they never expire. This means that until you change your password for the Git host, you won’t ever have to type in your credentials again. The downside of this approach is that your passwords are stored in cleartext in a plain file in your home directory.
- If you’re using a Mac, Git comes with an “osxkeychain” mode, which caches credentials in the secure keychain that’s attached to your system account. This method stores the credentials on disk, and they never expire, but they’re encrypted with the same system that stores HTTPS certificates and Safari auto-fills.
- If you’re using Windows, you can install a helper called “wincred.” This is similar to the “osxkeychain” helper described above, but uses the Windows Credential Store to control sensitive information.

You can choose one of these methods by setting a Git configuration value:

```
$ git config --global credential.helper cache
```

Some of these helpers have options. The “store” helper can take a `--file <path>` argument, which customizes where the plain-text file is saved (the default is `~/.git-credentials`). The “cache” helper accepts the `--timeout <seconds>` option, which changes the amount of time its daemon is kept running (the default is “900”, or 15 minutes). Here’s an example of how you’d configure the “store” helper with a custom file name:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git even allows you to configure several helpers. When looking for credentials for a particular host, Git will query them in order, and stop after the first answer is provided. When saving credentials, Git will send the username and password to **all** of the helpers in the list, and they can choose what to do with them. Here’s what a `.gitconfig` would look like if you had a credentials file on a thumb drive, but wanted to use the in-memory cache to save some typing if the drive isn’t plugged in:

```
[credential]
helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 30000
```

Under the Hood

How does this all work? Git’s root command for the credential-helper system is `git credential`, which takes a command as an argument, and then more input through stdin.

This might be easier to understand with an example. Let’s say that a credential helper has been configured, and the helper has stored credentials for `mygithost`. Here’s a session that uses the “fill” command, which is invoked when Git is trying to find credentials for a host:

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① This is the command line that initiates the interaction.

- ② Git-credential is then waiting for input on stdin. We provide it with the things we know: the protocol and hostname.
- ③ A blank line indicates that the input is complete, and the credential system should answer with what it knows.
- ④ Git-credential then takes over, and writes to stdout with the bits of information it found.
- ⑤ If credentials are not found, Git asks the user for the username and password, and provides them back to the invoking stdout (here they’re attached to the same console).

The credential system is actually invoking a program that’s separate from Git itself; which one and how depends on the `credential.helper` configuration value. There are several forms it can take:

| Configuration Value | Behavior |
|--|---|
| <code>foo</code> | Runs <code>git-credential-foo</code> |
| <code>foo -a --opt=bcd</code> | Runs <code>git-credential-foo -a --opt=bcd</code> |
| <code>/absolute/path/foo -xyz</code> | Runs <code>/absolute/path/foo -xyz</code> |
| <code>!f() { echo "password=s3cre7"; }; f</code> | Code after <code>!</code> evaluated in shell |

So the helpers described above are actually named `git-credential-cache`, `git-credential-store`, and so on, and we can configure them to take command-line arguments. The general form for this is “`git-credential-foo [args] <action>`.”. The stdin/stdout protocol is the same as git-credential, but they use a slightly different set of actions:

- `get` is a request for a username/password pair.
- `store` is a request to save a set of credentials in this helper’s memory.
- `erase` purge the credentials for the given properties from this helper’s memory.

For the `store` and `erase` actions, no response is required (Git ignores it anyway). For the `get` action, however, Git is very interested in what the helper has to say. If the helper doesn’t know anything useful, it can simply exit with no output, but if it does know, it should augment the provided information with the information it has stored. The output is treated like a series of assignment statements; anything provided will replace what Git already knows.

Here’s the same example from above, but skipping git-credential and going straight for git-credential-store:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

- ① Here we tell `git-credential-store` to save some credentials: the username “bob” and the password “s3cre7” are to be used when `https://mygithost` is accessed.
- ② Now we’ll retrieve those credentials. We provide the parts of the connection we already know (`https://mygithost`), and an empty line.
- ③ `git-credential-store` replies with the username and password we stored above.

Here’s what the `~/git.store` file looks like:

```
https://bob:s3cre7@mygithost
```

It’s just a series of lines, each of which contains a credential-decorated URL. The `osxkeychain` and `wincred` helpers use the native format of their backing stores, while `cache` uses its own in-memory format (which no other process can read).

A Custom Credential Cache

Given that `git-credential-store` and friends are separate programs from Git, it’s not much of a leap to realize that *any* program can be a Git credential helper. The helpers provided by Git cover many common use cases, but not all. For example, let’s say your team has some credentials that are shared with the entire team, perhaps for deployment. These are stored in a shared directory, but you don’t want to copy them to your own credential store, because they change often. None of the existing helpers cover this case; let’s see what it would take to write our own. There are several key features this program needs to have:

1. The only action we need to pay attention to is `get`; `store` and `erase` are write operations, so we’ll just exit cleanly when they’re received.
2. The file format of the shared-credential file is the same as that used by `git-credential-store`.
3. The location of that file is fairly standard, but we should allow the user to pass a custom path just in case.

Once again, we’ll write this extension in Ruby, but any language will work so long as Git can execute the finished product. Here’s the full source code of our new credential helper:

```

#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
  opts.banner = 'USAGE: git-credential-read-only [options] <action>'
  opts.on('-f', '--file PATH', 'Specify path for backing store') do
    |argpath|
      path = File.expand_path argpath
  end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
  break if line.strip == ''
  k,v = line.strip.split '=', 2
  known[k] = v
end

File.readlines(path).each do |fileline| ④
  prot,user,pass,host =
  fileline.scan(/^\s*(.*?):\/\/(.*):(.*?)@(.*)$/).first
  if prot == known['protocol'] and host == known['host'] and user == known['username'] then
    puts "protocol=#{prot}"
    puts "host=#{host}"
    puts "username=#{user}"
    puts "password=#{pass}"
    exit(0)
  end
end

```

- ① Here we parse the command-line options, allowing the user to specify the input file. The default is `~/.git-credentials`.
- ② This program only responds if the action is `get` and the backing-store file exists.
- ③ This loop reads from stdin until the first blank line is reached. The inputs are stored in the `known` hash for later reference.
- ④ This loop reads the contents of the storage file, looking for matches. If the protocol and host from `known` match this line, the program prints the results to stdout and exits.

We'll save our helper as `git-credential-read-only`, put it somewhere in our `PATH` and mark it executable. Here's what an interactive session looks like:

```
$ git credential-read-only --file=/mnt/shared/creds get  
protocol=https  
host=mygithost  
  
protocol=https  
host=mygithost  
username=bob  
password=s3cre7
```

Since its name starts with “git-”, we can use the simple syntax for the configuration value:

```
$ git config --global credential.helper 'read-only --file  
/mnt/shared/creds'
```

As you can see, extending this system is pretty straightforward, and can solve some common problems for you and your team.

Summary

You’ve seen a number of advanced tools that allow you to manipulate your commits and staging area more precisely. When you notice issues, you should be able to easily figure out what commit introduced them, when, and by whom. If you want to use subprojects in your project, you’ve learned how to accommodate those needs. At this point, you should be able to do most of the things in Git that you’ll need on the command line day to day and feel comfortable doing so.

Customizing Git

So far, we've covered the basics of how Git works and how to use it, and we've introduced a number of tools that Git provides to help you use it easily and efficiently. In this chapter, we'll see how you can make Git operate in a more customized fashion, by introducing several important configuration settings and the hooks system. With these tools, it's easy to get Git to work exactly the way you, your company, or your group needs it to.

Git Configuration

As you briefly saw in [開始](#), you can specify Git configuration settings with the `git config` command. One of the first things you did was set up your name and email address:

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

Now you'll learn a few of the more interesting options that you can set in this manner to customize your Git usage.

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in an `/etc/gitconfig` file, which contains values for every user on the system and all of their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.

The next place Git looks is the `~/.gitconfig` (or `~/.config/git/config`) file, which is specific to each user. You can make Git read and write to this file by passing the `--global` option.

Finally, Git looks for configuration values in the configuration file in the Git directory (`.git/config`) of whatever repository you're currently using. These values are specific to that single repository.

Each of these “levels” (system, global, local) overwrites values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`, for instance.

筆記 Git's configuration files are plain-text, so you can also set these values by manually editing the file and inserting the correct syntax. It's generally easier to run the `git config` command, though.

Basic Client Configuration

The configuration options recognized by Git fall into two categories: client-side and server-side. The majority of the options are client-side – configuring your personal working preferences. Many, *many* configuration options are supported, but a large fraction of them are only useful in certain edge cases. We'll only be covering the most common and most useful here. If you want to see a list of all the options your version of Git recognizes, you can run

```
$ man git-config
```

This command lists all the available options in quite a bit of detail. You can also find this reference material at <http://git-scm.com/docs/git-config.html>.

core.editor

By default, Git uses whatever you've set as your default text editor (`$VISUAL` or `$EDITOR`) or else falls back to the `vi` editor to create and edit your commit and tag messages. To change that default to something else, you can use the `core.editor` setting:

```
$ git config --global core.editor emacs
```

Now, no matter what is set as your default shell editor, Git will fire up Emacs to edit messages.

commit.template

If you set this to the path of a file on your system, Git will use that file as the default message when you commit. For instance, suppose you create a template file at `~/ .gitmessage.txt` that looks like this:

```
subject line  
what happened  
[ticket: X]
```

To tell Git to use it as the default message that appears in your editor when you run `git commit`, set the `commit.template` configuration value:

```
$ git config --global commit.template ~/ .gitmessage.txt  
$ git commit
```

Then, your editor will open to something like this for your placeholder commit message when you commit:

```
subject line
```

```
what happened
```

```
[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

If your team has a commit-message policy, then putting a template for that policy on your system and configuring Git to use it by default can help increase the chance of that policy being followed regularly.

core.pager

This setting determines which pager is used when Git pages output such as [log](#) and [diff](#). You can set it to [more](#) or to your favorite pager (by default, it's [less](#)), or you can turn it off by setting it to a blank string:

```
$ git config --global core.pager ''
```

If you run that, Git will page the entire output of all commands, no matter how long they are.

user.signingkey

If you're making signed annotated tags (as discussed in [Signing Your Work](#)), setting your GPG signing key as a configuration setting makes things easier. Set your key ID like so:

```
$ git config --global user.signingkey <gpg-key-id>
```

Now, you can sign tags without having to specify your key every time with the [git tag](#) command:

```
$ git tag -s <tag-name>
```

core.excludesfile

You can put patterns in your project's [.gitignore](#) file to have Git not see them as untracked files or try to stage them when you run [git add](#) on them, as discussed in [忽略不需要的檔案](#).

But sometimes you want to ignore certain files for all repositories that you work with. If your computer

is running Mac OS X, you’re probably familiar with `.DS_Store` files. If your preferred editor is Emacs or Vim, you know about filenames that end with a `~` or `.swp`.

This setting lets you write a kind of global `.gitignore` file. If you create a `~/ .gitignore_global` file with these contents:

```
*~  
.*.swp  
.DS_Store
```

…and you run `git config --global core.excludesfile ~/ .gitignore_global`, Git will never again bother you about those files.

help.autocorrect

If you mistype a command, it shows you something like this:

```
$ git chekcout master  
git: 'chekcout' is not a git command. See 'git --help'.  
  
Did you mean this?  
  checkout
```

Git helpfully tries to figure out what you meant, but it still refuses to do it. If you set `help.autocorrect` to 1, Git will actually run this command for you:

```
$ git chekcout master  
WARNING: You called a Git command named 'chekcout', which does not  
exist.  
Continuing under the assumption that you meant 'checkout'  
in 0.1 seconds automatically...
```

Note that “0.1 seconds” business. `help.autocorrect` is actually an integer which represents tenths of a second. So if you set it to 50, Git will give you 5 seconds to change your mind before executing the autocorrected command.

Colors in Git

Git fully supports colored terminal output, which greatly aids in visually parsing command output quickly and easily. A number of options can help you set the coloring to your preference.

color.ui

Git automatically colors most of its output, but there’s a master switch if you don’t like this behavior. To turn off all Git’s colored terminal output, do this:

```
$ git config --global color.ui false
```

The default setting is `auto`, which colors output when it's going straight to a terminal, but omits the color-control codes when the output is redirected to a pipe or a file.

You can also set it to `always` to ignore the difference between terminals and pipes. You'll rarely want this; in most scenarios, if you want color codes in your redirected output, you can instead pass a `--color` flag to the Git command to force it to use color codes. The default setting is almost always what you'll want.

`color.*`

If you want to be more specific about which commands are colored and how, Git provides verb-specific coloring settings. Each of these can be set to `true`, `false`, or `always`:

```
color.branch  
color.diff  
color.interactive  
color.status
```

In addition, each of these has subsettings you can use to set specific colors for parts of the output, if you want to override each color. For example, to set the meta information in your diff output to blue foreground, black background, and bold text, you can run

```
$ git config --global color.diff.meta "blue black bold"
```

You can set the color to any of the following values: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`. If you want an attribute like bold in the previous example, you can choose from `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap foreground and background).

External Merge and Diff Tools

Although Git has an internal implementation of diff, which is what we've been showing in this book, you can set up an external tool instead. You can also set up a graphical merge-conflict-resolution tool instead of having to resolve conflicts manually. We'll demonstrate setting up the Perforce Visual Merge Tool (P4Merge) to do your diffs and merge resolutions, because it's a nice graphical tool and it's free.

If you want to try this out, P4Merge works on all major platforms, so you should be able to do so. We'll use path names in the examples that work on Mac and Linux systems; for Windows, you'll have to change `/usr/local/bin` to an executable path in your environment.

To begin, [download P4Merge from Perforce](#). Next, you'll set up external wrapper scripts to run your commands. We'll use the Mac path for the executable; in other systems, it will be where your `p4merge` binary is installed. Set up a merge wrapper script named `extMerge` that calls your binary with all the arguments provided:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

The diff wrapper checks to make sure seven arguments are provided and passes two of them to your merge script. By default, Git passes the following arguments to the diff program:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Because you only want the `old-file` and `new-file` arguments, you use the wrapper script to pass the ones you need.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

You also need to make sure these tools are executable:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Now you can set up your config file to use your custom merge resolution and diff tools. This takes a number of custom settings: `merge.tool` to tell Git what strategy to use, `mergetool.<tool>.cmd` to specify how to run the command, `mergetool.<tool>.trustExitCode` to tell Git if the exit code of that program indicates a successful merge resolution or not, and `diff.external` to tell Git what command to run for diffs. So, you can either run four config commands

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

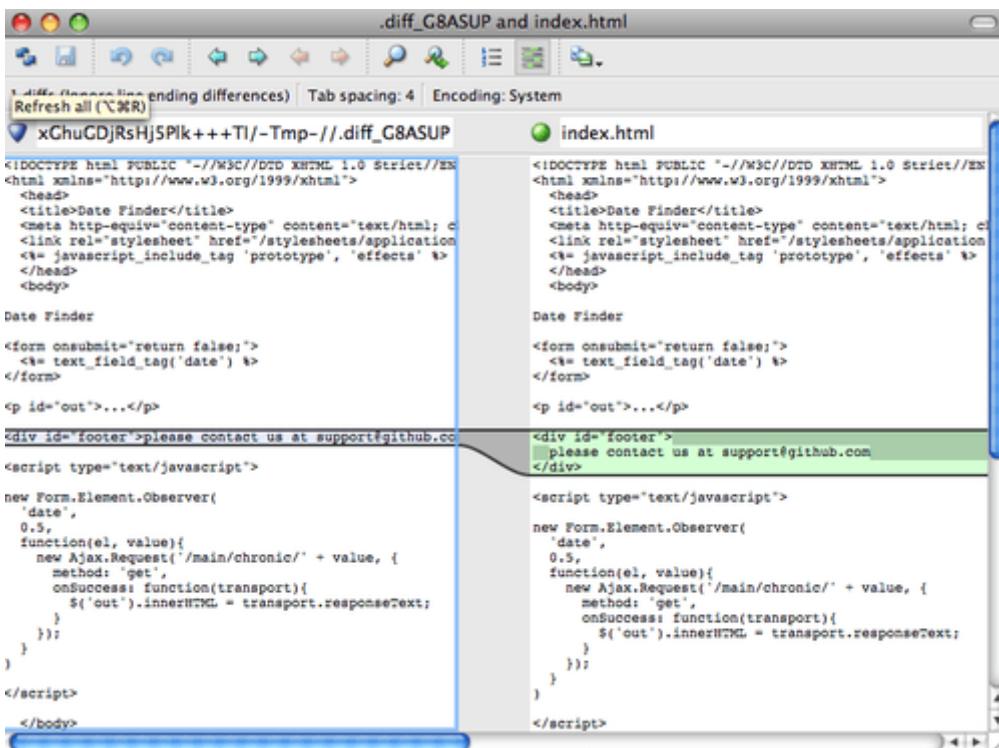
or you can edit your `~/.gitconfig` file to add these lines:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

After all this is set, if you run diff commands such as this:

```
$ git diff 32d1776b1^ 32d1776b1
```

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like this:



圖表 142. P4Merge.

If you try to merge two branches and subsequently have merge conflicts, you can run the command `git mergetool`; it starts P4Merge to let you resolve the conflicts through that GUI tool.

The nice thing about this wrapper setup is that you can change your diff and merge tools easily. For example, to change your `extDiff` and `extMerge` tools to run the KDiff3 tool instead, all you have to do is edit your `extMerge` file:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Now, Git will use the KDiff3 tool for diff viewing and merge conflict resolution.

Git comes preset to use a number of other merge-resolution tools without your having to set up the cmd configuration. To see a list of the tools it supports, try this:

```
$ git mergetool --tool-help  
'git mergetool --tool=<tool>' may be set to one of the following:  
    emerge  
    gvimdiff  
    gvimdiff2  
    opendiff  
    p4merge  
    vimdiff  
    vimdiff2
```

The following tools are valid, but not currently available:

```
    araxis  
    bc3  
    codecompare  
    deltaxalker  
    diffmerge  
    diffuse  
    ecmerge  
    kdiff3  
    meld  
    tkdiff  
    tortoisemerge  
    xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

If you're not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the `kdiff3` command is in your path, then you can run

```
$ git config --global merge.tool kdiff3
```

If you run this instead of setting up the `extMerge` and `extDiff` files, Git will use KDiff3 for merge resolution and the normal Git diff tool for diffs.

Formatting and Whitespace

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git has a few configuration options to help with these issues.

`core.autocrlf`

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending

characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the `core.autocrlf` setting. If you’re on a Windows machine, set it to `true` – this converts LF endings into CRLF when you check out code:

```
$ git config --global core.autocrlf true
```

If you’re on a Linux or Mac system that uses LF line endings, then you don’t want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting `core.autocrlf` to `input`:

```
$ git config --global core.autocrlf input
```

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you’re a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to `false`:

```
$ git config --global core.autocrlf false
```

core.whitespace

Git comes preset to detect and fix some whitespace issues. It can look for six primary whitespace issues – three are enabled by default and can be turned off, and three are disabled by default but can be activated.

The three that are turned on by default are `blank-at-eol`, which looks for spaces at the end of a line; `blank-at-eof`, which notices blank lines at the end of a file; and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The three that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with spaces instead of tabs (and is controlled by the `tabwidth` option); `tab-in-indent`, which watches for tabs in the indentation portion of a line; and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting `core.whitespace` to the values you want on or off, separated by commas. You can disable settings by either leaving them out of the setting string or prepending a `-` in front of the value. For example, if you want all but `cr-at-eol` to be set, you can do this:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git will detect these issues when you run a `git diff` command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with `git apply`. When you’re applying patches, you can ask Git to warn you if it’s applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Or you can have Git try to automatically fix the issue before applying the patch:

```
$ git apply --whitespace=fix <patch>
```

These options apply to the `git rebase` command as well. If you’ve committed whitespace issues but haven’t yet pushed upstream, you can run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it’s rewriting the patches.

Server Configuration

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

`receive.fsckObjects`

Git is capable of making sure every object received during a push still matches its SHA-1 checksum and points to valid objects. However, it doesn’t do this by default; it’s a fairly expensive operation, and might slow down the operation, especially on large repositories or pushes. If you want Git to check object consistency on every push, you can force it to do so by setting `receive.fsckObjects` to true:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty (or malicious) clients aren’t introducing corrupt data.

`receive.denyNonFastForwards`

If you rebase commits that you’ve already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn’t contain the commit that the remote branch currently points to, you’ll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you’re doing and can force-update the remote branch with a `-f` flag to your push command.

To tell Git to refuse force-pushes, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which we’ll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

`receive.denyDeletes`

One of the workarounds to the `denyNonFastForwards` policy is for the user to delete the branch and then push it back up with the new reference. To avoid this, set `receive.denyDeletes` to true:

```
$ git config --system receive.denyDeletes true
```

This denies any deletion of branches or tags – no user can do it. To remove remote branches, you must remove the ref files from the server manually. There are also more interesting ways to do this on a per-user basis via ACLs, as you’ll learn in [An Example Git-Enforced Policy](#).

Git Attributes

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don’t want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you’ll learn about some of the attributes you can set on your paths in your Git project and see a few examples of using this feature in practice.

Binary Files

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed. You’ll see how to tell Git which is which.

Identifying Binary Files

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on the Mac contain a file that ends in `.pbxproj`, which is basically a JSON (plain-text JavaScript data format) dataset written out to disk by the IDE, which records your build settings and so on. Although it’s technically a text file (because it’s all UTF-8), you don’t want to treat it as such because it’s really a lightweight database – you can’t merge the contents if two people change it, and diffs generally aren’t helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all `.pbxproj` files as binary data, add the following line to your `.gitattributes` file:

```
*.pbxproj binary
```

Now, Git won’t try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project.

Diffing Binary Files

You can also use the Git attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff.

First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific editor around, but oddly, everyone still uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run `git diff` normally, you only see something like this:

```
$ git diff  
diff --git a/chapter1.docx b/chapter1.docx  
index 88839c4..4afcb7c 100644  
Binary files a/chapter1.docx and b/chapter1.docx differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your `.gitattributes` file:

```
*.docx diff=word
```

This tells Git that any file that matches this pattern (`.docx`) should use the "word" filter when you try to view a diff that contains changes. What is the "word" filter? You have to set it up. Here you'll configure Git to use the `docx2txt` program to convert Word documents into readable text files, which it will then diff properly.

First, you'll need to install `docx2txt`; you can download it from <http://docx2txt.sourceforge.net>. Follow the instructions in the `INSTALL` file to put it somewhere your shell can find it. Next, you'll write a wrapper script to convert output to the format Git expects. Create a file that's somewhere in your path called `docx2txt`, and add these contents:

```
#!/bin/bash  
docx2txt.pl $1 -
```

Don't forget to `chmod a+x` that file. Finally, you can configure Git to use this script:

```
$ git config diff.word.textconv docx2txt
```

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in `.docx`, it should run those files through the "word" filter, which is defined as the `docx2txt` program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here's an example: Chapter 1 of this book was converted to Word format and committed in a Git repository. Then a new paragraph was added. Here's what `git diff` shows:

```
$ git diff  
diff --git a/chapter1.docx b/chapter1.docx  
index 0b013ca..ba25db5 100644  
--- a/chapter1.docx  
+++ b/chapter1.docx  
@@ -2,6 +2,7 @@
```

This chapter will be about getting started with Git. We will begin at the beginning by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it setup to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all setup to do so.

1.1. About Version Control

What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

+Testing: 1, 2, 3.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

1.1.1. Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

Git successfully and succinctly tells us that we added the string “Testing: 1, 2, 3.” , which is correct. It’s not perfect – formatting changes wouldn’t show up here – but it certainly works.

Another interesting problem you can solve this way involves diffing image files. One way to do this is to run image files through a filter that extracts their EXIF information – metadata that is recorded with most image formats. If you download and install the [exiftool](#) program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened. Put the following line in your [.gitattributes](#) file:

```
*.png diff=exif
```

Configure Git to use this tool:

```
$ git config diff.exif.textconv exiftool
```

If you replace an image in your project and run `git diff`, you see something like this:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
    ExifTool Version Number      : 7.74
-File Size                  : 70 kB
-File Modification Date/Time : 2009:04:21 07:02:45-07:00
+File Size                  : 94 kB
+File Modification Date/Time : 2009:04:21 07:02:43-07:00
    File Type                 : PNG
    MIME Type                 : image/png
-Image Width                : 1058
-Image Height               : 889
+Image Width                : 1056
+Image Height               : 827
    Bit Depth                 : 8
    Color Type                : RGB with Alpha
```

You can easily see that the file size and image dimensions have both changed.

Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked out and remove it again before it's added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an `Id` field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It's important to notice that it isn't the SHA-1 of the commit, but of the blob itself. Put the following line in your `.gitattributes` file:

```
*.txt ident
```

Add an `Id` reference to a test file:

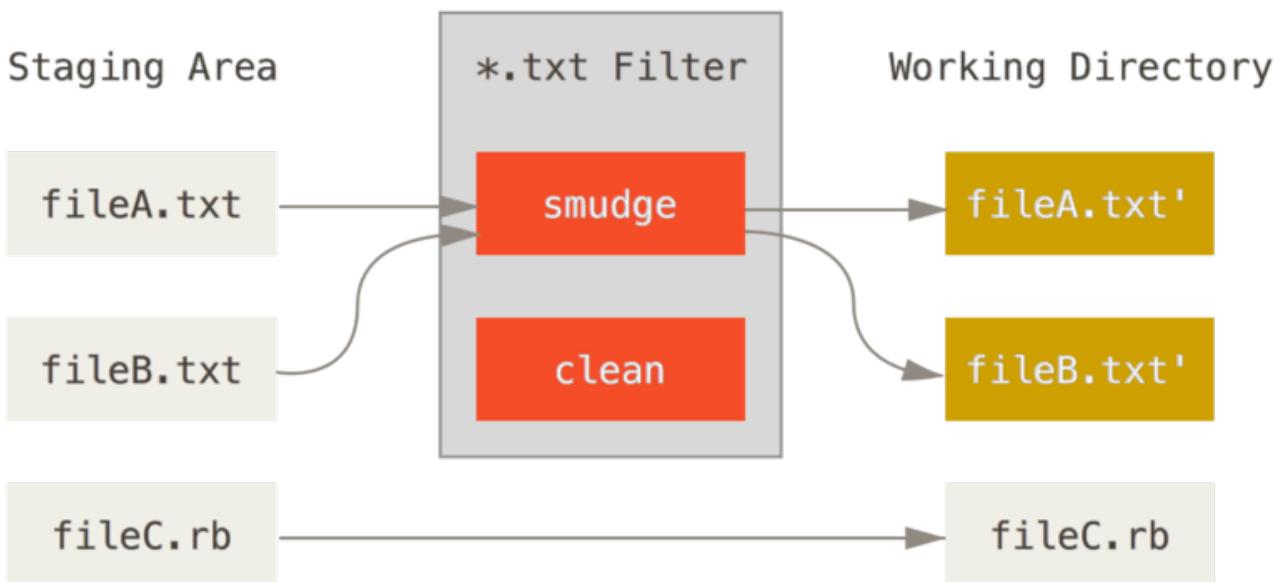
```
$ echo '$Id$' > test.txt
```

The next time you check out this file, Git injects the SHA-1 of the blob:

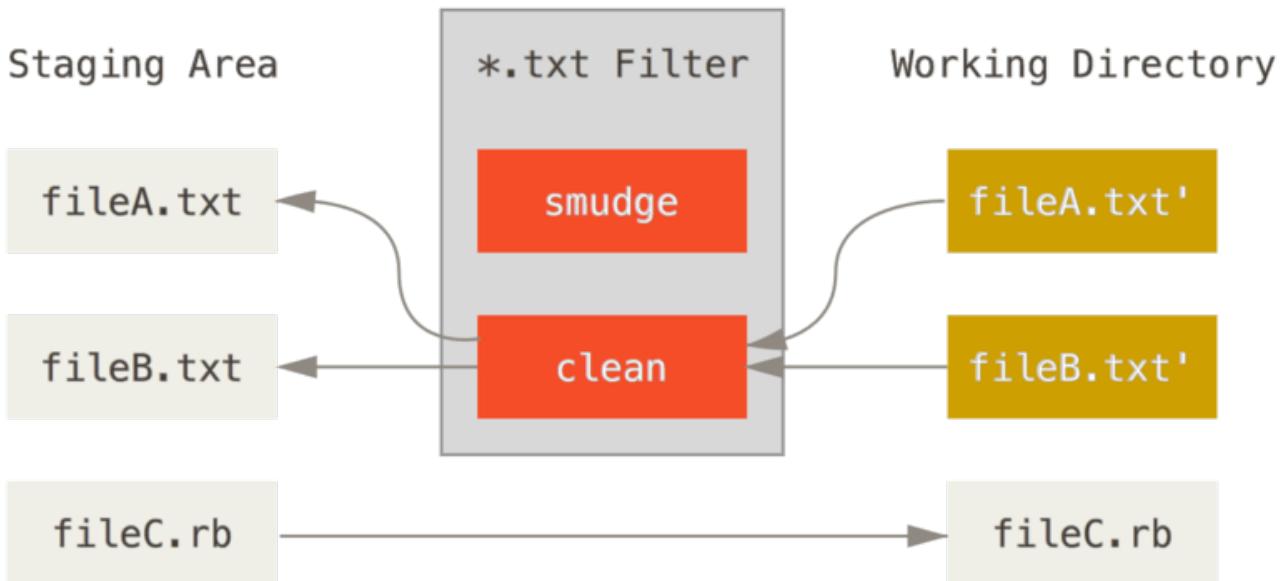
```
$ rm test.txt  
$ git checkout -- test.txt  
$ cat test.txt  
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

However, that result is of limited use. If you've used keyword substitution in CVS or Subversion, you can include a timestamp – the SHA-1 isn't all that helpful, because it's fairly random and you can't tell if one SHA-1 is older or newer than another just by looking at them.

It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are called "clean" and "smudge" filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they're checked out ("smudge", see [The "smudge" filter is run on checkout](#)) and just before they're staged ("clean", see [The "clean" filter is run when files are staged](#)). These filters can be set to do all sorts of fun things.



圖表 143. The "smudge" filter is run on checkout.



圖表 144. The “clean” filter is run when files are staged.

The original commit message for this feature gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the filter attribute in your `.gitattributes` file to filter `*.c` files with the “`indent`” filter:

```
*.c filter=indent
```

Then, tell Git what the “`indent`” filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the `indent` program before it stages them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program does essentially nothing: it spits out the same data that it comes in. This combination effectively filters all C source code files through `indent` before committing.

Another interesting example gets `$Date$` keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results – it should be simple to do in whatever language you’re most comfortable in. You can name this file `expand_date` and put it in your path. Now, you need to set up a filter in Git (call it `dater`) and tell it to use your `expand_date` filter to smudge the files

on checkout. You'll use a Perl expression to clean that up on commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe
"s/\$\Date[^\\\$]*\\\$/\$\$Date\\\$/"'
```

This Perl snippet strips out anything it sees in a `$Date$` string, to get back to where you started. Now that your filter is ready, you can test it by setting up a Git attribute for that file that engages the new filter and creating a file with your `$Date$` keyword:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

If you commit those changes and check out the file again, you see the keyword properly substituted:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project, but the driver (in this case, `dater`) isn't, so it won't work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

Exporting Your Repository

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

export-ignore

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don't want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

```
test/ export-ignore
```

Now, when you run `git archive` to create a tarball of your project, that directory won't be included in

the archive.

export-subst

When exporting files for deployment you can apply `git log`'s formatting and keyword-expansion processing to selected portions of files marked with the `export-subst` attribute.

For instance, if you want to include a file named `LAST_COMMIT` in your project, and have metadata about the last commit automatically injected into it when `git archive` runs, you can for example set up your `.gitattributes` and `LAST_COMMIT` files like this:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of the archived file will look like this:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

The substitutions can include for example the commit message and any git notes, and git log can do simple word wrapping:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' >
LAST_COMMIT
$ git commit -am 'export-subst uses git log's custom formatter

git archive uses git log's `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.

$ git archive @ | tar xf0 - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
    export-subst uses git log's custom formatter

        git archive uses git log's `pretty=format:` processor directly,
and
        strips the surrounding `$Format:` and `$` markup from the
output.
```

The resulting archive is suitable for deployment work, but like any exported archive it isn't suitable for further development work.

Merge Strategies

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

```
database.xml merge=ours
```

And then define a dummy `ours` merge strategy with:

```
$ git config --global merge.ours.driver true
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

Installing a Hook

The hooks are all stored in the `hooks` subdirectory of the Git directory. In most projects, that's `.git/hooks`. When you initialize a new repository with `git init`, Git populates the hooks directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine – you can write them in Ruby or Python or what have you. If you want to use the bundled hook scripts, you'll have to rename them; their file names all end with `.sample`.

To enable a hook script, put a file in the `hooks` subdirectory of your `.git` directory that is named appropriately (without any extension) and is executable. From that point forward, it should be called. We'll cover most of the major hook filenames here.

Client-Side Hooks

There are a lot of client-side hooks. This section splits them into committing-workflow hooks, email-workflow scripts, and everything else.

筆記

It's important to note that client-side hooks are **not** copied when you clone a repository. If your intent with these scripts is to enforce a policy, you'll probably want to do that on the server side; see the example in [An Example Git-Enforced Policy](#).

Committing-Workflow Hooks

The first four hooks have to do with the committing process.

The `pre-commit` hook is run first, before you even type in a commit message. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with `git commit --no-verify`. You can do things like check for code style (run `lint` or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

The `prepare-commit-msg` hook is run before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the commit author sees it. This hook takes a few parameters: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn't useful for normal commits; rather, it's good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert information.

The `commit-msg` hook takes one parameter, which again is the path to a temporary file that contains the commit message written by the developer. If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through. In the last section of this chapter, We'll demonstrate using this hook to check that your commit message is conformant to a required pattern.

After the entire commit process is completed, the `post-commit` hook runs. It doesn't take any parameters, but you can easily get the last commit by running `git log -1 HEAD`. Generally, this script is used for notification or something similar.

Email Workflow Hooks

You can set up three client-side hooks for an email-based workflow. They're all invoked by the `git am` command, so if you aren't using that command in your workflow, you can safely skip to the next section. If you're taking patches over email prepared by `git format-patch`, then some of these may be helpful to you.

The first hook that is run is `applypatch-msg`. It takes a single argument: the name of the temporary file that contains the proposed commit message. Git aborts the patch if this script exits non-zero. You can use this to make sure a commit message is properly formatted, or to normalize the message by having the script edit it in place.

The next hook to run when applying patches via `git am` is `pre-applypatch`. Somewhat confusingly, it is run *after* the patch is applied but before a commit is made, so you can use it to inspect the

snapshot before making the commit. You can run tests or otherwise inspect the working tree with this script. If something is missing or the tests don’t pass, exiting non-zero aborts the `git am` script without committing the patch.

The last hook to run during a `git am` operation is `post-applypatch`, which runs after the commit is made. You can use it to notify a group or the author of the patch you pulled in that you’ve done so. You can’t stop the patching process with this script.

Other Client Hooks

The `pre-rebase` hook runs before you rebase anything and can halt the process by exiting non-zero. You can use this hook to disallow rebasing any commits that have already been pushed. The example `pre-rebase` hook that Git installs does this, although it makes some assumptions that may not match with your workflow.

The `post-rewrite` hook is run by commands that replace commits, such as `git commit --amend` and `git rebase` (though not by `git filter-branch`). Its single argument is which command triggered the rewrite, and it receives a list of rewrites on `stdin`. This hook has many of the same uses as the `post-checkout` and `post-merge` hooks.

After you run a successful `git checkout`, the `post-checkout` hook runs; you can use it to set up your working directory properly for your project environment. This may mean moving in large binary files that you don’t want source controlled, auto-generating documentation, or something along those lines.

The `post-merge` hook runs after a successful `merge` command. You can use it to restore data in the working tree that Git can’t track, such as permissions data. This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.

The `pre-push` hook runs during `git push`, after the remote refs have been updated but before any objects have been transferred. It receives the name and location of the remote as parameters, and a list of to-be-updated refs through `stdin`. You can use it to validate a set of ref updates before a push occurs (a non-zero exit code will abort the push).

Git occasionally does garbage collection as part of its normal operation, by invoking `git gc --auto`. The `pre-auto-gc` hook is invoked just before the garbage collection takes place, and can be used to notify you that this is happening, or to abort the collection if now isn’t a good time.

Server-Side Hooks

In addition to the client-side hooks, you can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project. These scripts run before and after pushes to the server. The pre hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that’s as complex as you wish.

`pre-receive`

The first script to run when handling a push from a client is `pre-receive`. It takes a list of references that are being pushed from `stdin`; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they’re modifying with the push.

update

The `update` script is very similar to the `pre-receive` script, except that it’s run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, `pre-receive` runs only once, whereas `update` runs once per branch they’re pushing to. Instead of reading from `stdin`, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push. If the `update` script exits non-zero, only that reference is rejected; other references can still be updated.

post-receive

The `post-receive` hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same `stdin` data as the `pre-receive` hook. Examples include emailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can’t stop the push process, but the client doesn’t disconnect until it has completed, so be careful if you try to do anything that may take a long time.

An Example Git-Enforced Policy

In this section, you’ll use what you’ve learned to establish a Git workflow that checks for a custom commit message format, and allows only certain users to modify certain subdirectories in a project. You’ll build client scripts that help the developer know if their push will be rejected and server scripts that actually enforce the policies.

The scripts we’ll show are written in Ruby; partly because of our intellectual inertia, but also because Ruby is easy to read, even if you can’t necessarily write it. However, any language will work – all the sample hook scripts distributed with Git are in either Perl or Bash, so you can also see plenty of examples of hooks in those languages by looking at the samples.

Server-Side Hook

All the server-side work will go into the `update` file in your `hooks` directory. The `update` hook runs once per branch being pushed and takes three arguments:

- The name of the reference being pushed to
- The old revision where that branch was
- The new revision being pushed

You also have access to the user doing the pushing if the push is being run over SSH. If you’ve allowed everyone to connect with a single user (like “git”) via public-key authentication, you may have to give that user a shell wrapper that determines which user is connecting based on the public key, and set an environment variable accordingly. Here we’ll assume the connecting user is in the `$USER` environment variable, so your `update` script begins by gathering all the information you need:

```

#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies..."
puts "({$refname}) ($oldrev[0,6]) ($newrev[0,6])"

```

Yes, those are global variables. Don't judge – it's easier to demonstrate this way.

Enforcing a Specific Commit-Message Format

Your first challenge is to enforce that each commit message adheres to a particular format. Just to have a target, assume that each message has to include a string that looks like “ref: 1234” because you want each commit to link to a work item in your ticketing system. You must look at each commit being pushed up, see if that string is in the commit message, and, if the string is absent from any of the commits, exit non-zero so the push is rejected.

You can get a list of the SHA-1 values of all the commits that are being pushed by taking the `$newrev` and `$oldrev` values and passing them to a Git plumbing command called `git rev-list`. This is basically the `git log` command, but by default it prints out only the SHA-1 values and no other information. So, to get a list of all the commit SHA-1s introduced between one commit SHA-1 and another, you can run something like this:

```

$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475

```

You can take that output, loop through each of those commit SHA-1s, grab the message for it, and test that message against a regular expression that looks for a pattern.

You have to figure out how to get the commit message from each of these commits to test. To get the raw commit data, you can use another plumbing command called `git cat-file`. We'll go over all these plumbing commands in detail in [Git Internals](#); but for now, here's what that command gives you:

```

$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number

```

A simple way to get the commit message from a commit when you have the SHA-1 value is to go to the first blank line and take everything after that. You can do so with the `sed` command on Unix systems:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'  
changed the version number
```

You can use that incantation to grab the commit message from each commit that is trying to be pushed and exit if you see anything that doesn't match. To exit the script and reject the push, exit non-zero. The whole method looks like this:

```
$regex = /\[ref: (\d+)\]/  
  
# enforced custom commit message format  
def check_message_format  
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")  
  missed_revs.each do |rev|  
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`  
    if !$regex.match(message)  
      puts "[POLICY] Your message is not formatted correctly"  
      exit 1  
    end  
  end  
end  
check_message_format
```

Putting that in your `update` script will reject updates that contain commits that have messages that don't adhere to your rule.

Enforcing a User-Based ACL System

Suppose you want to add a mechanism that uses an access control list (ACL) that specifies which users are allowed to push changes to which parts of your projects. Some people have full access, and others can only push changes to certain subdirectories or specific files. To enforce this, you'll write those rules to a file named `acl` that lives in your bare Git repository on the server. You'll have the `update` hook look at those rules, see what files are being introduced for all the commits being pushed, and determine whether the user doing the push has access to update all those files.

The first thing you'll do is write your ACL. Here you'll use a format very much like the CVS ACL mechanism: it uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank meaning open access). All of these fields are delimited by a pipe (|) character.

In this case, you have a couple of administrators, some documentation writers with access to the `doc` directory, and one developer who only has access to the `lib` and `tests` directories, and your ACL file looks like this:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

You begin by reading this data into a structure that you can use. In this case, to keep the example simple, you'll only enforce the `avail` directives. Here is a method that gives you an associative array where the key is the user name and the value is an array of paths to which the user has write access:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

On the ACL file you looked at earlier, this `get_acl_access_data` method returns a data structure that looks like this:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, you need to determine what paths the commits being pushed have modified, so you can make sure the user who's pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in [Git 基礎](#)):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
README
lib/test.rb
```

If you use the ACL structure returned from the `get_acl_access_data` method and check it against the listed files in each of the commits, you can determine whether the user has access to push all of their commits:

```
# only allows certain users to modify certain subdirectories in a
project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list ${oldrev}..${newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'
#${rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path # user has access to everything
          || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

You get a list of new commits being pushed to your server with `git rev-list`. Then, for each of those commits, you find which files are modified and make sure the user who's pushing has access to all the paths being modified.

Now your users can't push any commits with badly formed messages or with modified files outside of their designated paths.

Testing It Out

If you run `chmod u+x .git/hooks/update`, which is the file into which you should have put all this code, and then try to push a commit with a non-compliant message, you get something like this:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

There are a couple of interesting things here. First, you see this where the hook starts running.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Remember that you printed that out at the very beginning of your update script. Anything your script echoes to `stdout` will be transferred to the client.

The next thing you'll notice is the error message.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

The first line was printed out by you, the other two were Git telling you that the update script exited non-zero and that is what is declining your push. Lastly, you have this:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

You'll see a remote rejected message for each reference that your hook declined, and it tells you that it was declined specifically because of a hook failure.

Furthermore, if someone tries to edit a file they don't have access to and push a commit containing it, they will see something similar. For instance, if a documentation author tries to push a commit modifying something in the `lib` directory, they see

```
[POLICY] You do not have access to push to lib/test.rb
```

From now on, as long as that `update` script is there and executable, your repository will never have a commit message without your pattern in it, and your users will be sandboxed.

Client-Side Hooks

The downside to this approach is the whining that will inevitably result when your users' commit pushes are rejected. Having their carefully crafted work rejected at the last minute can be extremely frustrating and confusing; and furthermore, they will have to edit their history to correct it, which isn't always for the faint of heart.

The answer to this dilemma is to provide some client-side hooks that users can run to notify them when they're doing something that the server is likely to reject. That way, they can correct any problems before committing and before those issues become more difficult to fix. Because hooks aren't transferred with a clone of a project, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but Git won't set them up automatically.

To begin, you should check your commit message just before each commit is recorded, so you know the server won't reject your changes due to badly formatted commit messages. To do this, you can add the `commit-msg` hook. If you have it read the message from the file passed as the first argument and compare that to the pattern, you can force Git to abort the commit if there is no match:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see this:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

No commit was completed in that instance. However, if your message contains the proper pattern, Git allows you to commit:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Next, you want to make sure you aren't modifying files that are outside your ACL scope. If your

project' s `.git` directory contains a copy of the ACL file you used previously, then the following `pre-commit` script will enforce those constraints for you:

```
#!/usr/bin/env ruby

$user      = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a
# project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only
HEAD`\.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
      if !access_path || (path.index(access_path) == 0)
        has_file_access = true
      end
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

This is roughly the same script as the server-side part, but with two important differences. First, the ACL file is in a different place, because this script runs from your working directory, not from your `.git` directory. You have to change the path to the ACL file from this

```
access = get_acl_access_data('acl')
```

to this:

```
access = get_acl_access_data('.git/acl')
```

The other important difference is the way you get a listing of the files that have been changed. Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Instead of

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

you have to use

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But those are the only two differences – otherwise, the script works the same way. One caveat is that it expects you to be running locally as the same user you push as to the remote machine. If that is different, you must set the `$user` variable manually.

One other thing we can do here is make sure the user doesn't push non-fast-forwarded references. To get a reference that isn't a fast-forward, you either have to rebase past a commit you've already pushed up or try pushing a different local branch up to the same remote branch.

Presumably, the server is already configured with `receive.denyDeletes` and `receive.denyNonFastForwards` to enforce this policy, so the only accidental thing you can try to catch is rebasing commits that have already been pushed.

Here is an example pre-rebase script that checks for that. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that is reachable from one of your remote references, it aborts the rebase.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote.refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote.refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to
#{remote_ref}"
      exit 1
    end
  end
end
```

This script uses a syntax that wasn't covered in [Revision Selection](#). You get a list of commits that have already been pushed up by running this:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

The `SHA^@` syntax resolves to all the parents of that commit. You’re looking for any commit that is reachable from the last commit on the remote and that isn’t reachable from any parent of any of the SHA-1s you’re trying to push up – meaning it’s a fast-forward.

The main drawback to this approach is that it can be very slow and is often unnecessary – if you don’t try to force the push with `-f`, the server will warn you and not accept the push. However, it’s an interesting exercise and can in theory help you avoid a rebase that you might later have to go back and fix.

Summary

We’ve covered most of the major ways that you can customize your Git client and server to best fit your workflow and projects. You’ve learned about all sorts of configuration settings, file-based attributes, and event hooks, and you’ve built an example policy-enforcing server. You should now be able to make Git fit nearly any workflow you can dream up.

Git and Other Systems

The world isn't perfect. Usually, you can't immediately switch every project you come in contact with to Git. Sometimes you're stuck on a project using another VCS, and wish it was Git. We'll spend the first part of this chapter learning about ways to use Git as a client when the project you're working on is hosted in a different system.

At some point, you may want to convert your existing project to Git. The second part of this chapter covers how to migrate your project into Git from several specific systems, as well as a method that will work if no pre-built import tool exists.

Git as a Client

Git provides such a nice experience for developers that many people have figured out how to use it on their workstation, even if the rest of their team is using an entirely different VCS. There are a number of these adapters, called "bridges," available. Here we'll cover the ones you're most likely to run into in the wild.

Git and Subversion

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It's been around for more than a decade, and for most of that time was the *de facto* VCS choice for open-source projects. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on, while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

`git svn`

The base command in Git for all the Subversion bridging commands is `git svn`. It takes quite a few commands, so we'll show the most common while going through a few simple workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you're working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make your life easier.

Setting Up

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion. For these tests, we created a new Subversion repository on Google Code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn  
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change  
#!/bin/sh  
exit 0;  
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn \  
http://progit-example.googlecode.com/svn/
```

This sets up the properties to run the sync. You can then clone the code by running

```
$ svnsync sync file:///tmp/test-svn  
Committed revision 1.  
Copied properties for revision 1.  
Transmitting file data .....[ ... ]  
Committed revision 2.  
Copied properties for revision 2.  
[ ... ]
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository – it's ridiculously inefficient, but it's the only easy way to do this.

Getting Started

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You'll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from a real

hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125
(refs/remotes/origin/trunk)
  A  m4/acx_pthread.m4
  A  m4/stl_hash.m4
  A  java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
  A  java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
(refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk =>
file:///tmp/test-svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-
calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn’t that big, but Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Note how this tool manages Subversion tags as remote refs. Let’s take a closer look with the Git

plumbing command `show-ref`:

```
$ git show-ref  
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master  
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-  
branch  
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2  
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca  
refs/remotes/origin/tags/release-2.0.1  
cbda99cb45d9abcb9793db1d4f70ae562a969f1e  
refs/remotes/origin/tags/release-2.0.2  
a9f074aa89e826d6f9d30808ce5ae3ffe711fed  
refs/remotes/origin/tags/release-2.0.2rc1  
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone:

```
$ git show-ref  
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master  
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1  
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2  
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0  
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0  
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.

Committing Back to Subversion

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'  
[master 4af61fd] Adding git-svn instructions to the README  
1 file changed, 5 insertions(+)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M README.txt
Committed r77
    M README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5
(refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-
21af03df0a68
```

Notice that the SHA-1 checksum that originally started with `4af61fd` when you committed now begins with `95e0222`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

Pulling in New Changes

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and
refs/remotes/origin/trunk differ, using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the
committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and
refs/remotes/origin/trunk differ, using rebase:
:100644 100644 65536c6e30d263495c17d781962cff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the
committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Now, all your work is on top of what is on the Subversion server, so you can successfully `dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M README.txt
Committed r85
    M README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8
(refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Note that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict (much like how Subversion works). If someone else pushes a change to one file and then you push a change to another file, your `dcommit` will work fine:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M configure.ac
Committed r87
    M autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7
(refs/remotes/origin/trunk)
    M configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4
(refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and
refs/remotes/origin/trunk differ, using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M autogen.sh
First, rewinding head to replay your work on top of it...
```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server – in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```
$ git svn rebase
    M autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b
(refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Running `git svn rebase` every once in a while makes sure your code is always up to date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` – otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an `experiment` branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M  CHANGES.txt
Committed r89
    M  CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981
(refs/remotes/origin/trunk)
    M  COPYING.txt
    M  INSTALL.txt
Committed r90
    M  INSTALL.txt
    M  COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0
(refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the `experiment` branch – instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it, as though you ran `git merge --squash`; they don't see the commit data about where it came from or when it was committed.

Subversion Branching

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using `git svn`.

Creating a New SVN Branch

To create a new branch in Subversion, you run `git svn branch [branchname]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-
svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk =>
file:///tmp/test-svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302
(refs/remotes/origin/opera)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to `trunk` on the server, not `opera`.

Switching Active Branches

Git figures out what branch your dcommits go to by looking for the tip of any of your Subversion branches in your history – you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an `opera` branch that you can work on separately, you can run

```
$ git branch opera remotes/origin/opera
```

Now, if you want to merge your `opera` branch into `trunk` (your `master` branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say “Merge branch `opera`” instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong – the `dcommit` makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this situation – Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, `opera`) after you merge it into `trunk`.

Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
-----
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines
autogen change

-----
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines
Merge branch 'experiment'

-----
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines
updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't committed don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =====
 2   temporal
79   schacon Committing in git-svn.
78   schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and
the Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two commands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to stdout the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

Git-Svn Summary

The `git svn` tools are useful if you're stuck with a Subversion server, or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

Git and Mercurial

The DVCS universe is larger than just Git. In fact, there are many other systems in this space, each with their own angle on how to do distributed version control correctly. Apart from Git, the most popular is Mercurial, and the two are very similar in many respects.

The good news, if you prefer Git's client-side behavior but are working with a project whose source code is controlled with Mercurial, is that there's a way to use Git as a client for a Mercurial-hosted repository. Since the way Git talks to server repositories is through remotes, it should come as no surprise that this bridge is implemented as a remote helper. The project's name is `git-remote-hg`, and it can be found at <https://github.com/felipec/git-remote-hg>.

git-remote-hg

First, you need to install `git-remote-hg`. This basically entails dropping its file somewhere in your path, like so:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-
remote-hg
$ chmod +x ~/bin/git-remote-hg
```

…assuming `~/bin` is in your `$PATH`. `Git-remote-hg` has one other dependency: the `mercurial` library for Python. If you have Python installed, this is as simple as:

```
$ pip install mercurial
```

(If you don't have Python installed, visit <https://www.python.org/> and get it first.)

The last thing you'll need is the Mercurial client. Go to <http://mercurial.selenic.com/> and install it if you haven't already.

Now you're ready to rock. All you need is a Mercurial repository you can push to. Fortunately, every Mercurial repository can act this way, so we'll just use the "hello world" repository everyone uses to learn Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Getting Started

Now that we have a suitable "server-side" repository, we can go through a typical workflow. As you'll see, these two systems are similar enough that there isn't much friction.

As always with Git, first we clone:

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master,
master) Create a makefile
* 65bb417 Create a standard "hello, world" program
```

You'll notice that working with a Mercurial repository uses the standard `git clone` command. That's because `git-remote-hg` is working at a fairly low level, using a similar mechanism to how Git's HTTP/S protocol is implemented (remote helpers). Since Git and Mercurial are both designed for every client to have a full copy of the repository history, this command makes a full clone, including all the project's history, and does it fairly quickly.

The `log` command shows two commits, the latest of which is pointed to by a whole slew of refs. It turns out some of these aren't actually there. Let's take a look at what's actually in the `.git` directory:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
└── notes
    └── hg
└── remotes
    └── origin
        └── HEAD
└── tags
```

9 directories, 5 files

Git-remote-hg is trying to make things more idiomatically Git-esque, but under the hood it's managing the conceptual mapping between two slightly different systems. The `refs/hg` directory is where the actual remote refs are stored. For example, the `refs/hg/origin/branches/default` is a Git ref file that contains the SHA-1 starting with "ac7955c", which is the commit that `master` points to. So the `refs/hg` directory is kind of like a fake `refs/remotes/origin`, but it has the added distinction between bookmarks and branches.

The `notes/hg` file is the starting point for how git-remote-hg maps Git commit hashes to Mercurial changeset IDs. Let's explore a bit:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f... 65bb417...
100644 blob 485e178... ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

So `refs/notes/hg` points to a tree, which in the Git object database is a list of other objects with names. `git ls-tree` outputs the mode, type, object hash, and filename for items inside a tree. Once we dig down to one of the tree items, we find that inside it is a blob named "ac9117f" (the SHA-1

hash of the commit pointed to by `master`), with contents “0a04b98” (which is the ID of the Mercurial changeset at the tip of the `default` branch).

The good news is that we mostly don’t have to worry about all of this. The typical workflow won’t be very different from working with a Git remote.

There’s one more thing we should attend to before we continue: ignores. Mercurial and Git use a very similar mechanism for this, but it’s likely you don’t want to actually commit a `.gitignore` file into a Mercurial repository. Fortunately, Git has a way to ignore files that’s local to an on-disk repository, and the Mercurial format is compatible with Git, so you just have to copy it over:

```
$ cp .hgignore .git/info/exclude
```

The `.git/info/exclude` file acts just like a `.gitignore`, but isn’t included in commits.

Workflow

Let’s assume we’ve done some work and made some commits on the `master` branch, and you’re ready to push it to the remote repository. Here’s what our repository looks like right now:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create
a makefile
* 65bb417 Create a standard "hello, world" program
```

Our `master` branch is two commits ahead of `origin/master`, but those two commits exist only on our local machine. Let’s see if anyone else has been doing important work at the same time:

```
$ git fetch
From hg::/tmp/hello
  ac7955c..df85e87  master      -> origin/master
  ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add
some documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Since we used the `--all` flag, we see the “notes” refs that are used internally by `git-remote-hg`, but

we can ignore them. The rest is what we expected; `origin/master` has advanced by one commit, and our history has now diverged. Unlike the other systems we work with in this chapter, Mercurial is capable of handling merges, so we’re not going to do anything fancy.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\ 
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
| refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add
| some documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Perfect. We run the tests and everything passes, so we’re ready to share our work with the rest of the team:

```
$ git push
To hg:::/tmp/hello
  df85e87..0c64627  master -> master
```

That’s it! If you take a look at the Mercurial repository, you’ll see that this did what we’d expect:

```
$ hg log -G --style compact
o 5[tip]:4,2 dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 64f27bcefc35 2014-08-14 19:27 -0700 ben
| | Update makefile
|
| o 3:1 4256fc29598f 2014-08-14 19:27 -0700 ben
| | Goodbye
|
@ | 2 7db0b4848b3c 2014-08-14 19:30 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

The changeset numbered 2 was made by Mercurial, and the changesets numbered 3 and 4 were made by git-remote-hg, by pushing commits made with Git.

Branches and Bookmarks

Git has only one kind of branch: a reference that moves when commits are made. In Mercurial, this kind of a reference is called a “bookmark,” and it behaves in much the same way as a Git branch.

Mercurial’s concept of a “branch” is more heavyweight. The branch that a changeset is made on is recorded *with the changeset*, which means it will always be in the repository history. Here’s an example of a commit that was made on the `develop` branch:

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

Note the line that begins with “branch”. Git can’t really replicate this (and doesn’t need to; both types of branch can be represented as a Git ref), but git-remote-hg needs to understand the difference, because Mercurial cares.

Creating Mercurial bookmarks is as easy as creating Git branches. On the Git side:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg:::/tmp/hello
 * [new branch]      featureA -> featureA
```

That’s all there is to it. On the Mercurial side, it looks like this:

```

$ hg bookmarks
  featureA
      5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]  8f65e5e02793  2014-08-14 20:06 -0700  ben
|   More documentation
|
o  5[featureA]:4,2  bd5ac26f11f9  2014-08-14 20:02 -0700  ben
|\   Merge remote-tracking branch 'origin/master'
|
| o 4  0434aaa6b91f  2014-08-14 20:01 -0700  ben
| |   update makefile
|
| o 3:1  318914536c86  2014-08-14 20:00 -0700  ben
| |   goodbye
|
o | 2  f098c7f45c4f  2014-08-14 20:01 -0700  ben
|/   Add some documentation
|
o 1  82e55d328c8c  2005-08-26 01:21 -0700  mpm
|   Create a makefile
|
o 0  0a04b987be5a  2005-08-26 01:20 -0700  mpm
|   Create a standard "hello, world" program

```

Note the new **[featureA]** tag on revision 5. These act exactly like Git branches on the Git side, with one exception: you can't delete a bookmark from the Git side (this is a limitation of remote helpers).

You can work on a “heavyweight” Mercurial branch also: just put a branch in the **branches** namespace:

```

$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent

```

Here's what that looks like on the Mercurial side:

```

$ hg branches
permanent                               7:a4529d07aad4
develop                                6:8f65e5e02793
default                                 5:bd5ac26f11f9 (inactive)
$ hg log -G
o changeset: 7:a4529d07aad4
| branch: permanent
| tag: tip
| parent: 5:bd5ac26f11f9
| user: Ben Straub <ben@straub.cc>
| date: Thu Aug 14 20:21:09 2014 -0700
| summary: A permanent change
|
| @ changeset: 6:8f65e5e02793
| / branch: develop
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:06:38 2014 -0700
| | summary: More documentation
|
o changeset: 5:bd5ac26f11f9
| \ bookmark: featureA
| | parent: 4:0434aaa6b91f
| | parent: 2:f098c7f45c4f
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:02:21 2014 -0700
| | summary: Merge remote-tracking branch 'origin/master'
[...]

```

The branch name “permanent” was recorded with the changeset marked 7.

From the Git side, working with either of these branch styles is the same: just checkout, commit, fetch, merge, pull, and push as you normally would. One thing you should know is that Mercurial doesn’t support rewriting history, only adding to it. Here’s what our Mercurial repository looks like after an interactive rebase and a force-push:

```

$ hg log --style compact -G
o 10[tip] 99611176cbc9 2014-08-14 20:21 -0700 ben
| A permanent change
|
o 9 f23e12f939c3 2014-08-14 20:01 -0700 ben
| Add some documentation
|
o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben
| goodbye
|
| o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben
| | A permanent change
|
| | @ 6 8f65e5e02793 2014-08-14 20:06 -0700 ben
| | / More documentation
|
| | o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
| | | \ Merge remote-tracking branch 'origin/master'
|
| | | o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | | | update makefile
|
+---o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | | goodbye
|
| o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program

```

Changesets 8, 9, and 10 have been created and belong to the **permanent** branch, but the old changesets are still there. This can be **very** confusing for your teammates who are using Mercurial, so try to avoid it.

Mercurial Summary

Git and Mercurial are similar enough that working across the boundary is fairly painless. If you avoid changing history that's left your machine (as is generally recommended), you may not even be aware that the other end is Mercurial.

Git and Perforce

Perforce is a very popular version-control system in corporate environments. It's been around since 1995, which makes it the oldest system covered in this chapter. As such, it's designed with the constraints of its day; it assumes you're always connected to a single central server, and only one version is kept on the local disk. To be sure, its features and constraints are well-suited to several specific problems, but there are lots of projects using Perforce where Git would actually work better.

There are two options if you'd like to mix your use of Perforce and Git. The first one we'll cover is the "Git Fusion" bridge from the makers of Perforce, which lets you expose subtrees of your Perforce depot as read-write Git repositories. The second is git-p4, a client-side bridge that lets you use Git as a Perforce client, without requiring any reconfiguration of the Perforce server.

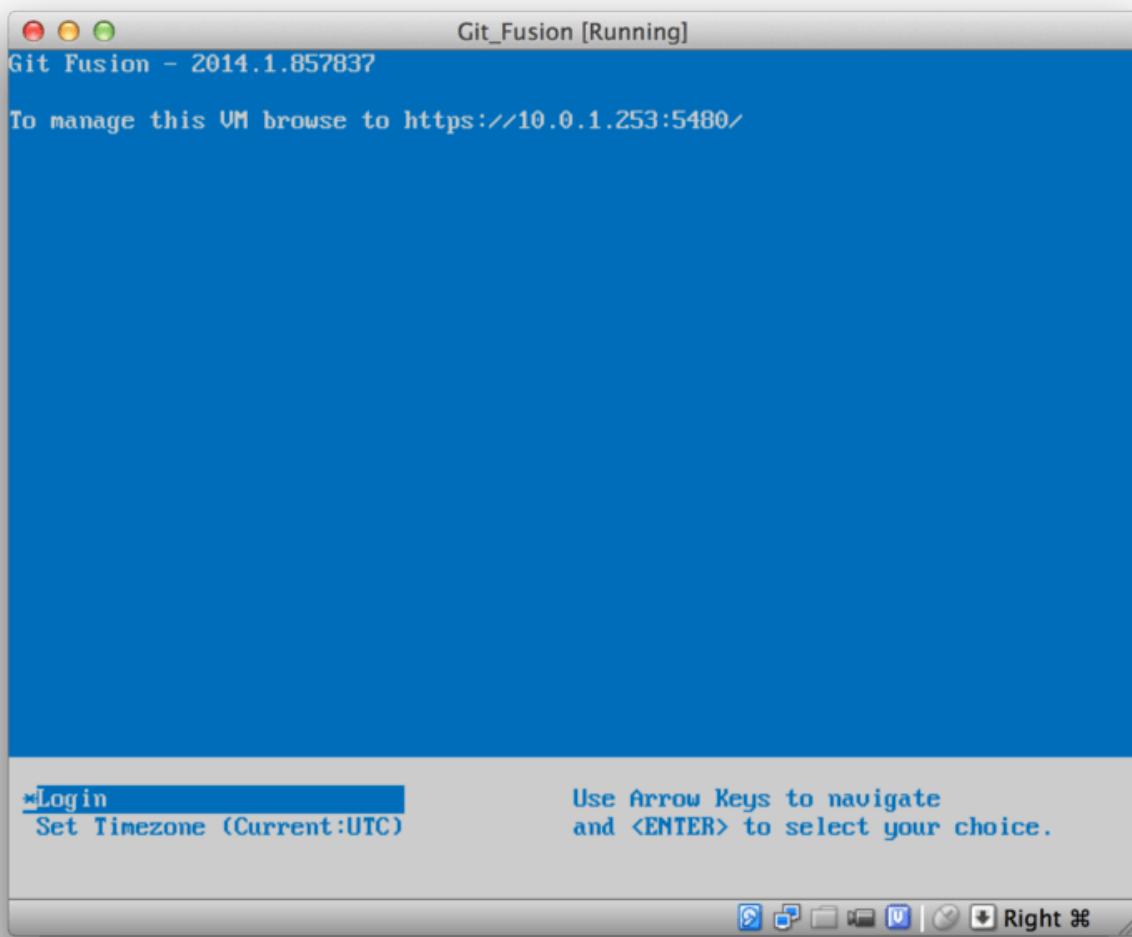
Git Fusion

Perforce provides a product called Git Fusion (available at <http://www.perforce.com/git-fusion>), which synchronizes a Perforce server with Git repositories on the server side.

Setting Up

For our examples, we'll be using the easiest installation method for Git Fusion, which is downloading a virtual machine that runs the Perforce daemon and Git Fusion. You can get the virtual machine image from <http://www.perforce.com/downloads/Perforce/20-User>, and once it's finished downloading, import it into your favorite virtualization software (we'll use VirtualBox).

Upon first starting the machine, it asks you to customize the password for three Linux users (`root`, `perforce`, and `git`), and provide an instance name, which can be used to distinguish this installation from others on the same network. When that has all completed, you'll see this:



圖表 145. The Git Fusion virtual machine boot screen.

You should take note of the IP address that’s shown here, we’ll be using it later on. Next, we’ll create a Perforce user. Select the “Login” option at the bottom and press enter (or SSH to the machine), and log in as `root`. Then use these commands to create a user:

```
$ p4 -p localhost:1666 -u super user -f john  
$ p4 -p localhost:1666 -u john passwd  
$ exit
```

The first one will open a VI editor to customize the user, but you can accept the defaults by typing `:wq` and hitting enter. The second one will prompt you to enter a password twice. That’s all we need to do with a shell prompt, so exit out of the session.

The next thing you’ll need to do to follow along is to tell Git not to verify SSL certificates. The Git Fusion image comes with a certificate, but it’s for a domain that won’t match your virtual machine’s IP address, so Git will reject the HTTPS connection. If this is going to be a permanent installation, consult the Perforce Git Fusion manual to install a different certificate; for our example purposes, this will suffice:

```
$ export GIT_SSL_NO_VERIFY=true
```

Now we can test that everything is working.

```
$ git clone https://10.0.1.254/Talkhouse  
Cloning into 'Talkhouse'...  
Username for 'https://10.0.1.254': john  
Password for 'https://john@10.0.1.254':  
remote: Counting objects: 630, done.  
remote: Compressing objects: 100% (581/581), done.  
remote: Total 630 (delta 172), reused 0 (delta 0)  
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.  
Resolving deltas: 100% (172/172), done.  
Checking connectivity... done.
```

The virtual-machine image comes equipped with a sample project that you can clone. Here we’re cloning over HTTPS, with the `john` user that we created above; Git asks for credentials for this connection, but the credential cache will allow us to skip this step for any subsequent requests.

Fusion Configuration

Once you’ve got Git Fusion installed, you’ll want to tweak the configuration. This is actually fairly easy to do using your favorite Perforce client; just map the `// .git-fusion` directory on the Perforce server into your workspace. The file structure looks like this:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
|
└── p4gf_config
    ├── repos
    │   └── Talkhouse
    │       └── p4gf_config
    └── users
        └── p4gf_usermap

498 directories, 287 files
```

The `objects` directory is used internally by Git Fusion to map Perforce objects to Git and vice versa, you won't have to mess with anything in there. There's a global `p4gf_config` file in this directory, as well as one for each repository – these are the configuration files that determine how Git Fusion behaves. Let's take a look at the file in the root:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

We won't go into the meanings of these flags here, but note that this is just an INI-formatted text file,

much like Git uses for configuration. This file specifies the global options, which can then be overridden by repository-specific configuration files, like `repos/Talkhouse/p4gf_config`. If you open this file, you'll see a `[@repo]` section with some settings that are different from the global defaults. You'll also see sections that look like this:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

This is a mapping between a Perforce branch and a Git branch. The section can be named whatever you like, so long as the name is unique. `git-branch-name` lets you convert a depot path that would be cumbersome under Git to a more friendly name. The `view` setting controls how Perforce files are mapped into the Git repository, using the standard view mapping syntax. More than one mapping can be specified, like in this example:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

This way, if your normal workspace mapping includes changes in the structure of the directories, you can replicate that with a Git repository.

The last file we'll discuss is `users/p4gf_usermap`, which maps Perforce users to Git users, and which you may not even need. When converting from a Perforce changeset to a Git commit, Git Fusion's default behavior is to look up the Perforce user, and use the email address and full name stored there for the author/committer field in Git. When converting the other way, the default is to look up the Perforce user with the email address stored in the Git commit's author field, and submit the changeset as that user (with permissions applying). In most cases, this behavior will do just fine, but consider the following mapping file:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Each line is of the format `<user> <email> "<full name>"`, and creates a single user mapping. The first two lines map two distinct email addresses to the same Perforce user account. This is useful if you've created Git commits under several different email addresses (or change email addresses), but want them to be mapped to the same Perforce user. When creating a Git commit from a Perforce changeset, the first line matching the Perforce user is used for Git authorship information.

The last two lines mask Bob and Joe's actual names and email addresses from the Git commits that are created. This is nice if you want to open-source an internal project, but don't want to publish your employee directory to the entire world. Note that the email addresses and full names should be unique, unless you want all the Git commits to be attributed to a single fictional author.

Workflow

Perforce Git Fusion is a two-way bridge between Perforce and Git version control. Let's have a look at how it feels to work from the Git side. We'll assume we've mapped in the "Jam" project using a configuration file as shown above, which we can clone like this:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest
metrowerks on Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

The first time you do this, it may take some time. What's happening is that Git Fusion is converting all the applicable changesets in the Perforce history into Git commits. This happens locally on the server, so it's relatively fast, but if you have a lot of history, it can still take some time. Subsequent fetches do incremental conversion, so it'll feel more like Git's native speed.

As you can see, our repository looks exactly like any other Git repository you might work with. There are three branches, and Git has helpfully created a local `master` branch that tracks `origin/master`. Let's do a bit of work, and create a couple of new commits:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on
Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

We have two new commits. Now let's check if anyone else has been working:

```

$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
  d254865..6afeb15 master      -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]

```

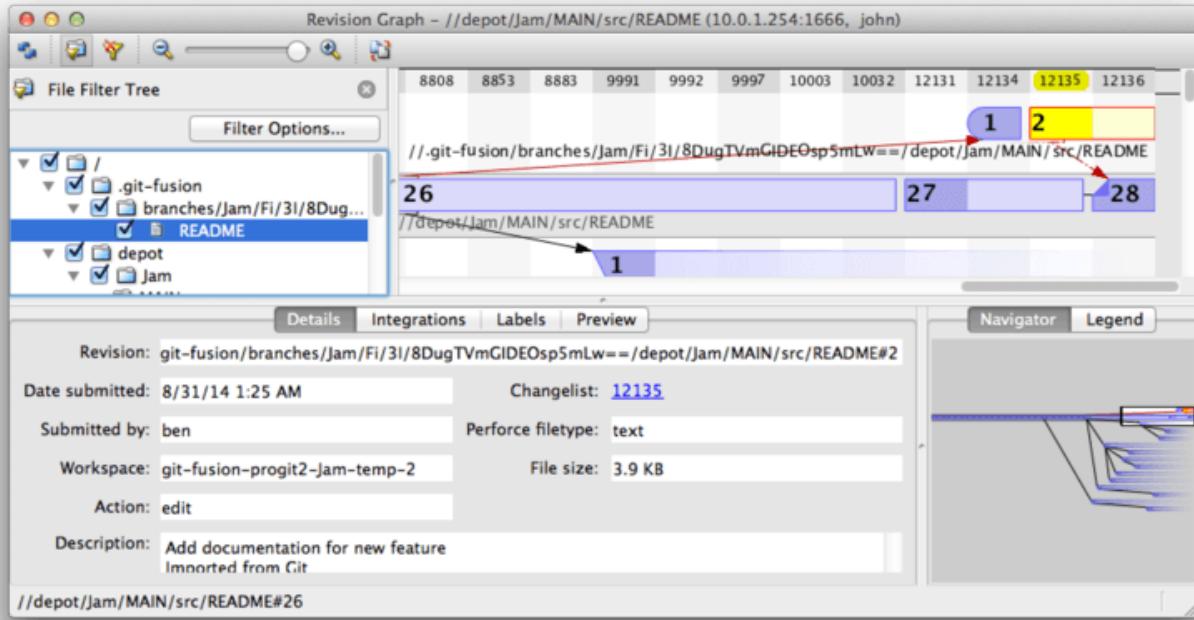
It looks like someone was! You wouldn't know it from this view, but the **6afeb15** commit was actually created using a Perforce client. It just looks like another commit from Git's point of view, which is exactly the point. Let's see how the Perforce server deals with a merge commit:

```

$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Perforce: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
  6afeb15..89cba2b master -> master

```

Git thinks it worked. Let's take a look at the history of the **README** file from Perforce's point of view, using the revision graph feature of **p4v**:



圖表 146. Perforce revision graph resulting from Git push.

If you've never seen this view before, it may seem confusing, but it shows the same concepts as a graphical viewer for Git history. We're looking at the history of the **README** file, so the directory tree at top left only shows that file as it surfaces in various branches. At top right, we have a visual graph of how different revisions of the file are related, and the big-picture view of this graph is at bottom right. The rest of the view is given to the details view for the selected revision (2 in this case).

One thing to notice is that the graph looks exactly like the one in Git's history. Perforce didn't have a named branch to store the 1 and 2 commits, so it made an "anonymous" branch in the **.git-fusion** directory to hold it. This will also happen for named Git branches that don't correspond to a named Perforce branch (and you can later map them to a Perforce branch using the configuration file).

Most of this happens behind the scenes, but the end result is that one person on a team can be using Git, another can be using Perforce, and neither of them will know about the other's choice.

Git-Fusion Summary

If you have (or can get) access to your Perforce server, Git Fusion is a great way to make Git and Perforce talk to each other. There's a bit of configuration involved, but the learning curve isn't very steep. This is one of the few sections in this chapter where cautions about using Git's full power will not appear. That's not to say that Perforce will be happy with everything you throw at it – if you try to rewrite history that's already been pushed, Git Fusion will reject it – but Git Fusion tries very hard to feel native. You can even use Git submodules (though they'll look strange to Perforce users), and merge branches (this will be recorded as an integration on the Perforce side).

If you can't convince the administrator of your server to set up Git Fusion, there is still a way to use these tools together.

Git-p4

Git-p4 is a two-way bridge between Git and Perforce. It runs entirely inside your Git repository, so you

won’t need any kind of access to the Perforce server (other than user credentials, of course). Git-p4 isn’t as flexible or complete a solution as Git Fusion, but it does allow you to do most of what you’d want to do without being invasive to the server environment.

筆記

You’ll need the **p4** tool somewhere in your **PATH** to work with git-p4. As of this writing, it is freely available at <http://www.perforce.com/downloads/Perforce/20-User>.

Setting Up

For example purposes, we’ll be running the Perforce server from the Git Fusion OVA as shown above, but we’ll bypass the Git Fusion server and go directly to the Perforce version control.

In order to use the **p4** command-line client (which git-p4 depends on), you’ll need to set a couple of environment variables:

```
$ export P4PORT=10.0.1.254:1666  
$ export P4USER=john
```

Getting Started

As with anything in Git, the first command is to clone:

```
$ git p4 clone //depot/www/live www-shallow  
Importing from //depot/www/live into www-shallow  
Initialized empty Git repository in /private/tmp/www-shallow/.git/  
Doing initial import of //depot/www/live/ from revision #head into  
refs/remotes/p4/master
```

This creates what in Git terms is a “shallow” clone; only the very latest Perforce revision is imported into Git; remember, Perforce isn’t designed to give every revision to every user. This is enough to use Git as a Perforce client, but for other purposes it’s not enough.

Once it’s finished, we have a fully-functional Git repository:

```
$ cd myproject  
$ git log --oneline --all --graph --decorate  
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of  
//depot/www/live/ from the state at revision #head
```

Note how there’s a “p4” remote for the Perforce server, but everything else looks like a standard clone. Actually, that’s a bit misleading; there isn’t actually a remote there.

```
$ git remote -v
```

No remotes exist in this repository at all. Git-p4 has created some refs to represent the state of the server, and they look like remote refs to **git log**, but they’re not managed by Git itself, and you can’t push to them.

Workflow

Okay, let's do some work. Let's assume you've made some progress on a very important feature, and you're ready to show it to the rest of your team.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from
the state at revision #head
```

We've made two new commits that we're ready to submit to the Perforce server. Let's check if anyone else was working today:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

Looks like they were, and **master** and **p4/master** have diverged. Perforce's branching system is *nothing* like Git's, so submitting merge commits doesn't make any sense. Git-p4 recommends that you rebase your commits, and even comes with a shortcut to do so:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
index.html | 2 +- 
1 file changed, 1 insertion(+), 1 deletion(-)
```

You can probably tell from the output, but **git p4 rebase** is a shortcut for **git p4 sync** followed by **git rebase p4/master**. It's a bit smarter than that, especially when working with multiple branches, but this is a good approximation.

Now our history is linear again, and we're ready to contribute our changes back to Perforce. The **git p4 submit** command will try to create a new Perforce revision for every Git commit between

`p4/master` and `master`. Running it drops us into our favorite editor, and the contents of the file look something like this:

```

# A Perforce Change Specification.

#
# Change:      The change number. 'new' on a new changelist.
# Date:        The date this specification was last modified.
# Client:      The client on which the changelist was created. Read-
#               only.
# User:        The user who created the changelist.
# Status:      Either 'pending' or 'submitted'. Read-only.
# Type:        Either 'public' or 'restricted'. Default is 'public'.
# Description: Comments about the changelist. Required.
# Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list. (New changelists
#               only.)
# Files:       What opened files from the default changelist are to be
#               added
#               to this changelist. You may delete files from this
#               list.
#               (New changelists only.)

```

Change: new

Client: john_bens-mbp_8487

User: john

Status: new

Description:
Update link

Files:
//depot/www/live/index.html # edit

```

##### git author ben@straub.cc does not match your p4 account.
##### Use option --preserve-user to modify authorship.
##### Variable git-p4.skipUserNameCheck hides this message.
##### everything below this line is just the diff #####
--- //depot/www/live/index.html 2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/index.html 2014-08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
</td>
<td valign=top>
Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
Jam/MR</a>,
a software build tool.
</td>
```

This is mostly the same content you'd see by running `p4 submit`, except the stuff at the end which git-p4 has helpfully included. Git-p4 tries to honor your Git and Perforce settings individually when it has to provide a name for a commit or changeset, but in some cases you want to override it. For example, if the Git commit you're importing was written by a contributor who doesn't have a Perforce user account, you may still want the resulting changeset to look like they wrote it (and not you).

Git-p4 has helpfully imported the message from the Git commit as the content for this Perforce changeset, so all we have to do is save and quit, twice (once for each commit). The resulting shell output will look something like this:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at
/Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

The result is as though we just did a `git push`, which is the closest analogy to what actually did happen.

Note that during this process every Git commit is turned into a Perforce changeset; if you want to squash them down into a single changeset, you can do that with an interactive rebase before running `git p4 submit`. Also note that the SHA-1 hashes of all the commits that were submitted as changesets have changed; this is because git-p4 adds a line to the end of each commit it converts:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

[git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

What happens if you try to submit a merge commit? Let's give it a try. Here's the situation we've gotten ourselves into:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
* 1dcbf21 Merge remote-tracking branch 'p4/master'
| \
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
| /
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

The Git and Perforce history diverge after 775a46f. The Git side has two commits, then a merge commit with the Perforce head, then another commit. We're going to try to submit these on top of a single changeset on the Perforce side. Let's see what would happen if we tried to submit now:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at
/Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would apply
    b4959b6 Trademark
    cbacd0a Table borders: yes please
    3be6fd8 Correct email address
```

The **-n** flag is short for **--dry-run**, which tries to report what would happen if the submit command were run for real. In this case, it looks like we'd be creating three Perforce changesets, which correspond to the three non-merge commits that don't yet exist on the Perforce server. That sounds like exactly what we want, let's see how it turns out:

```
$ git p4 submit
[...]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision
#head
```

Our history became linear, just as though we had rebased before submitting (which is in fact exactly what happened). This means you can be free to create, work on, throw away, and merge branches on the Git side without fear that your history will somehow become incompatible with Perforce. If you can rebase it, you can contribute it to a Perforce server.

Branching

If your Perforce project has multiple branches, you’re not out of luck; git-p4 can handle that in a way that makes it feel like Git. Let’s say your Perforce depot is laid out like this:

```
//depot
└── project
    ├── main
    └── dev
```

And let’s say you have a **dev** branch, which has a view spec that looks like this:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 can automatically detect that situation and do the right thing:

```

$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfac Populate //depot/project/main/... //depot/project/dev/...
|/
* 2b83451 Project init

```

Note the “@all” specifier in the depot path; that tells git-p4 to clone not just the latest changeset for that subtree, but all changesets that have ever touched those paths. This is closer to Git’s concept of a clone, but if you’re working on a project with a long history, it could take a while.

The `--detect-branches` flag tells git-p4 to use Perforce’s branch specs to map the branches to Git refs. If these mappings aren’t present on the Perforce server (which is a perfectly valid way to use Perforce), you can tell git-p4 what the branch mappings are, and you get the same result:

```

$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .

```

Setting the `git-p4.branchList` configuration variable to `main:dev` tells git-p4 that “main” and “dev” are both branches, and the second one is a child of the first one.

If we now `git checkout -b dev p4/project/dev` and make some commits, git-p4 is smart enough to target the right branch when we do `git p4 submit`. Unfortunately, git-p4 can’t mix shallow clones and multiple branches; if you have a huge project and want to work on more than one branch, you’ll have to `git p4 clone` once for each branch you want to submit to.

For creating or integrating branches, you’ll have to use a Perforce client. Git-p4 can only sync and submit to existing branches, and it can only do it one linear changeset at a time. If you merge two branches in Git and try to submit the new changeset, all that will be recorded is a bunch of file changes; the metadata about which branches are involved in the integration will be lost.

Git and Perforce Summary

Git-p4 makes it possible to use a Git workflow with a Perforce server, and it’s pretty good at it. However, it’s important to remember that Perforce is in charge of the source, and you’re only using Git to work locally. Just be really careful about sharing Git commits; if you have a remote that other people use, don’t push any commits that haven’t already been submitted to the Perforce server.

If you want to freely mix the use of Perforce and Git as clients for source control, and you can convince the server administrator to install it, Git Fusion makes using Git a first-class version-control client for a Perforce server.

Git and TFS

Git is becoming popular with Windows developers, and if you’re writing code on Windows, there’s a good chance you’re using Microsoft’s Team Foundation Server (TFS). TFS is a collaboration suite that includes defect and work-item tracking, process support for Scrum and others, code review, and version control. There’s a bit of confusion ahead: **TFS** is the server, which supports controlling source code using both Git and their own custom VCS, which they’ve dubbed **TFVC** (Team Foundation Version Control). Git support is a somewhat new feature for TFS (shipping with the 2013 version), so all of the tools that predate that refer to the version-control portion as “TFS”, even though they’re mostly working with TFVC.

If you find yourself on a team that’s using TFVC but you’d rather use Git as your version-control client, there’s a project for you.

Which Tool

In fact, there are two: git-tf and git-tfs.

Git-tfs (found at <https://github.com/git-tfs/git-tfs>) is a .NET project, and (as of this writing) it only runs on Windows. To work with Git repositories, it uses the .NET bindings for libgit2, a library-oriented implementation of Git which is highly performant and allows a lot of flexibility with the guts of a Git repository. Libgit2 is not a complete implementation of Git, so to cover the difference git-tfs will actually call the command-line Git client for some operations, so there are no artificial limits on what it can do with Git repositories. Its support of TFVC features is very mature, since it uses the Visual Studio assemblies for operations with servers. This does mean you’ll need access to those assemblies, which means you need to install a recent version of Visual Studio (any edition since version 2010, including Express since version 2012), or the Visual Studio SDK.

Git-tf (whose home is at <https://gittf.codeplex.com>) is a Java project, and as such runs on any computer with a Java runtime environment. It interfaces with Git repositories through JGit (a JVM implementation of Git), which means it has virtually no limitations in terms of Git functions. However, its support for TFVC is limited as compared to git-tfs – it does not support branches, for instance.

So each tool has pros and cons, and there are plenty of situations that favor one over the other. We’ll cover the basic usage of both of them in this book.

筆記

You’ll need access to a TFVC-based repository to follow along with these instructions. These aren’t as plentiful in the wild as Git or Subversion repositories, so you may need to create one of your own. Codeplex (<https://www.codeplex.com>) or Visual Studio Online (<http://www.visualstudio.com>) are both good choices for this.

Getting Started: `git-tf`

The first thing you do, just as with any Git project, is clone. Here’s what that looks like with `git-tf`:

```
$ git tf clone https://tfvc.codeplex.com:443/tfs/TFS13 $/myproject/Main  
project_git
```

The first argument is the URL of a TFVC collection, the second is of the form `$/project/branch`, and the third is the path to the local Git repository that is to be created (this last one is optional). Git-tf can only work with one branch at a time; if you want to make checkins on a different TFVC branch, you'll have to make a new clone from that branch.

This creates a fully functional Git repository:

```
$ cd project_git  
$ git log --all --oneline --decorate  
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

This is called a *shallow* clone, meaning that only the latest changeset has been downloaded. TFVC isn't designed for each client to have a full copy of the history, so git-tf defaults to only getting the latest version, which is much faster.

If you have some time, it's probably worth it to clone the entire project history, using the `--deep` option:

```
$ git tf clone https://tfvc.codeplex.com:443/tfs/TFS13 $/myproject/Main \  
project_git --deep  
Username: domain\user  
Password:  
Connecting to TFS...  
Cloning $/myproject into /tmp/project_git: 100%, done.  
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a  
$ cd project_git  
$ git log --all --oneline --decorate  
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye  
126aa7b (tag: TFS_C35189)  
8f77431 (tag: TFS_C35178) FIRST  
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the  
\  
Team Project Creation Wizard
```

Notice the tags with names like `TFS_C35189`; this is a feature that helps you know which Git commits are associated with TFVC changesets. This is a nice way to represent it, since you can see with a simple log command which of your commits is associated with a snapshot that also exists in TFVC. They aren't necessary (and in fact you can turn them off with `git config git-tf.tag false`) – git-tf keeps the real commit-changeset mappings in the `.git/git-tf` file.

Getting Started: `git-tfs`

Git-tfs cloning behaves a bit differently. Observe:

```

PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeefb961958b674

```

Notice the **--with-branches** flag. Git-tfs is capable of mapping TFVC branches to Git branches, and this flag tells it to set up a local Git branch for every TFVC branch. This is highly recommended if you've ever branched or merged in TFS, but it won't work with a server older than TFS 2010 – before that release, “branches” were just folders, so git-tfs can't tell them from regular folders.

Let's take a look at the resulting Git repository:

```

PS> git log --oneline --graph --decorate --all
* 44cd729 (tfv/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfv/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

    Hello

    git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]$/.myproject/Trunk;C
16

```

There are two local branches, **master** and **featureA**, which represent the initial starting point of the clone (**Trunk** in TFVC) and a child branch (**featureA** in TFVC). You can also see that the **tfv** “remote” has a couple of refs too: **default** and **featureA**, which represent TFVC branches. Git-tfs maps the branch you cloned from to **tfv/default**, and others get their own names.

Another thing to notice is the **git-tfs-id:** lines in the commit messages. Instead of tags, git-tfs uses these markers to relate TFVC changesets to Git commits. This has the implication that your Git commits will have a different SHA-1 hash before and after they have been pushed to TFVC.

Git-tf[s] Workflow

Regardless of which tool you’re using, you should set a couple of Git configuration values to avoid running into issues.

筆記

```
$ git config set --local core.ignorecase=true  
$ git config set --local core.autocrlf=false
```

The obvious next thing you’re going to want to do is work on the project. TFVC and TFS have several features that may add complexity to your workflow:

1. Feature branches that aren’t represented in TFVC add a bit of complexity. This has to do with the **very** different ways that TFVC and Git represent branches.
2. Be aware that TFVC allows users to “checkout” files from the server, locking them so nobody else can edit them. This obviously won’t stop you from editing them in your local repository, but it could get in the way when it comes time to push your changes up to the TFVC server.
3. TFS has the concept of “gated” checkins, where a TFS build-test cycle has to complete successfully before the checkin is allowed. This uses the “shelve” function in TFVC, which we don’t cover in detail here. You can fake this in a manual fashion with `git-tf`, and `git-tfs` provides the `checkintool` command which is gate-aware.

In the interest of brevity, what we’ll cover here is the happy path, which sidesteps or avoids most of these issues.

Workflow: `git-tf`

Let’s say you’ve done some work, made a couple of Git commits on `master`, and you’re ready to share your progress on the TFVC server. Here’s our Git repository:

```
$ git log --oneline --graph --decorate --all  
* 4178a82 (HEAD, master) update code  
* 9df2ae3 update readme  
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye  
* 126aa7b (tag: TFS_C35189)  
* 8f77431 (tag: TFS_C35178) FIRST  
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via  
the \  
Team Project Creation Wizard
```

We want to take the snapshot that’s in the `4178a82` commit and push it up to the TFVC server. First things first: let’s see if any of our teammates did anything since we last connected:

```

$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via
the \
    Team Project Creation Wizard

```

Looks like someone else is working, too, and now we have divergent history. This is where Git shines, but we have two choices of how to proceed:

1. Making a merge commit feels natural as a Git user (after all, that's what `git pull` does), and `git-tf` can do this for you with a simple `git tf pull`. Be aware, however, that TFVC doesn't think this way, and if you push merge commits your history will start to look different on both sides, which can be confusing. However, if you plan on submitting all of your changes as one changeset, this is probably the easiest choice.
2. Rebasing makes our commit history linear, which means we have the option of converting each of our Git commits into a TFVC changeset. Since this leaves the most options open, we recommend you do it this way; `git-tf` even makes it easy for you with `git tf pull --rebase`.

The choice is yours. For this example, we'll be rebasing:

```

$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via
the \
    Team Project Creation Wizard

```

Now we're ready to make a checkin to the TFVC server. `Git-tf` gives you the choice of making a single changeset that represents all the changes since the last one (`--shallow`, which is the default) and

creating a new changeset for each Git commit (`--deep`). For this example, we'll just create one changeset:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via
the \
    Team Project Creation Wizard
```

There's a new `TFS_C35348` tag, indicating that TFVC is storing the exact same snapshot as the `5a0e25e` commit. It's important to note that not every Git commit needs to have an exact counterpart in TFVC; the `6eb3eb5` commit, for example, doesn't exist anywhere on the server.

That's the main workflow. There are a couple of other considerations you'll want to keep in mind:

- There is no branching. Git-tf can only create Git repositories from one TFVC branch at a time.
- Collaborate using either TFVC or Git, but not both. Different git-tf clones of the same TFVC repository may have different commit SHA-1 hashes, which will cause no end of headaches.
- If your team's workflow includes collaborating in Git and syncing periodically with TFVC, only connect to TFVC with one of the Git repositories.

Workflow: `git-tfs`

Let's walk through the same scenario using git-tfs. Here are the new commits we've made to the `master` branch in our Git repository:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfss/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfss/default) Hello
* b75da1a New project
```

Now let's see if anyone else has done work while we were hacking away:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfss/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfss/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Yes, it turns out our coworker has added a new TFVC changeset, which shows up as the new `aea74a0` commit, and the `tfss/default` remote branch has moved.

As with `git-tf`, we have two fundamental options for how to resolve this divergent history:

1. Rebase to preserve a linear history.
2. Merge to preserve what actually happened.

In this case, we’re going to do a “deep” checkin, where every Git commit becomes a TFVC changeset, so we want to rebase.

```
PS> git rebase tfss/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfss/default) update documentation
| * 44cd729 (tfss/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Now we’re ready to complete our contribution by checking in our code to the TFVC server. We’ll use the `rcheckin` command here to create a TFVC changeset for each Git commit in the path from HEAD to the first `tfss` remote branch found (the `checkin` command would only create one changeset, sort of like squashing Git commits).

```

PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
  add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto
new TFS-commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
  edit
.git\tfs\default\workspace\ConsoleApplication1\ConsoleApplication1\Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto
new TFS-commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project

```

Notice how after every successful checkin to the TFVC server, git-tfs is rebasing the remaining work onto what it just did. That's because it's adding the `git-tfs-id` field to the bottom of the commit messages, which changes the SHA-1 hashes. This is exactly as designed, and there's nothing to worry about, but you should be aware that it's happening, especially if you're sharing Git commits with others.

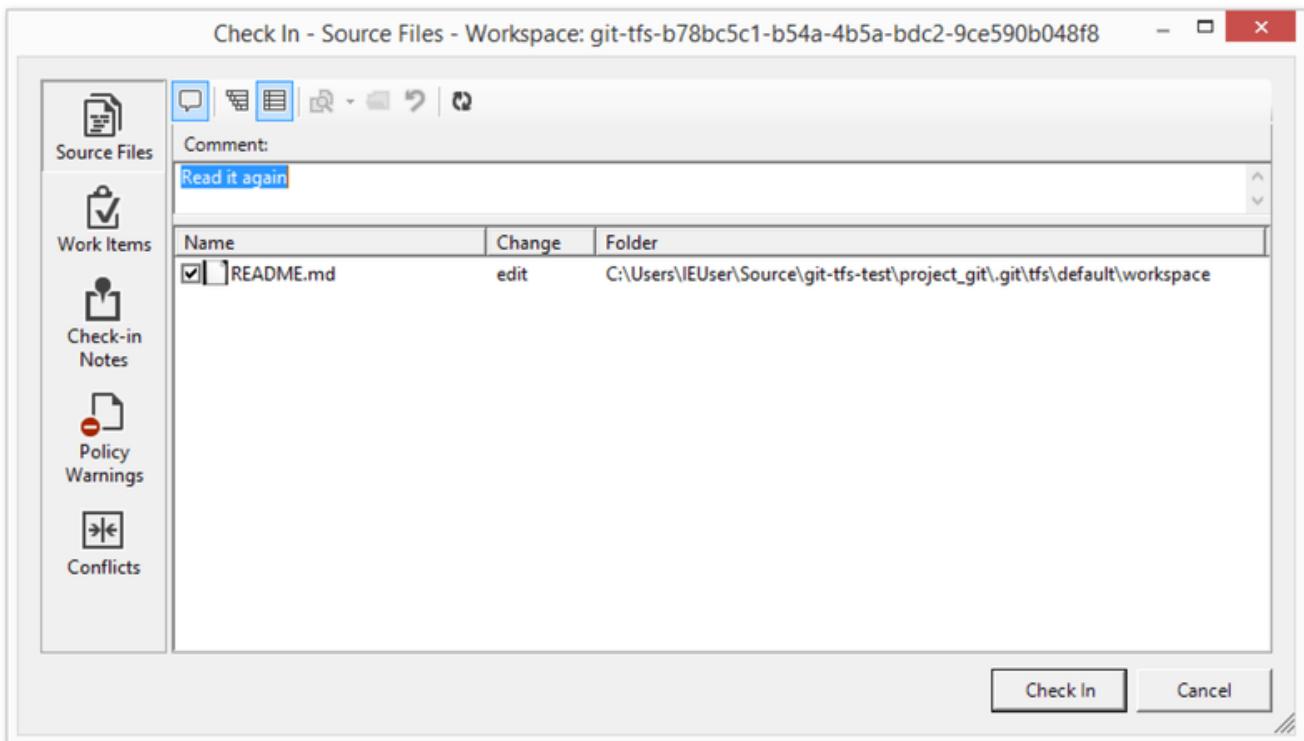
TFS has many features that integrate with its version control system, such as work items, designated reviewers, gated checkins, and so on. It can be cumbersome to work with these features using only a command-line tool, but fortunately git-tfs lets you launch a graphical checkin tool very easily:

```

PS> git tfs checkin
PS> git tfs ct

```

It looks a bit like this:



圖表 147. The *git-tfs checkin* tool.

This will look familiar to TFS users, as it's the same dialog that's launched from within Visual Studio.

Git-tfs also lets you control TFVC branches from your Git repository. As an example, let's create one:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfv/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfv/default, master) update code
* 71a5ddc update readme
*aea74a0 update documentation
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Creating a branch in TFVC means adding a changeset where that branch now exists, and this is projected as a Git commit. Note also that git-tfs **created** the **tfv/featureBee** remote branch, but **HEAD** is still pointing to **master**. If you want to work on the newly-minted branch, you'll want to base your new commits on the **1d54865** commit, perhaps by creating a topic branch from that commit.

Git and TFS Summary

Git-tf and Git-tfs are both great tools for interfacing with a TFVC server. They allow you to use the power of Git locally, avoid constantly having to round-trip to the central TFVC server, and make your life as a developer much easier, without forcing your entire team to migrate to Git. If you're working

on Windows (which is likely if your team is using TFS), you’ll probably want to use git-tfs, since its feature set is more complete, but if you’re working on another platform, you’ll be using git-tf, which is more limited. As with most of the tools in this chapter, you should choose one of these version-control systems to be canonical, and use the other one in a subordinate fashion – either Git or TFVC should be the center of collaboration, but not both.

Migrating to Git

If you have an existing codebase in another VCS but you’ve decided to start using Git, you must migrate your project one way or another. This section goes over some importers for common systems, and then demonstrates how to develop your own custom importer. You’ll learn how to import data from several of the bigger professionally used SCM systems, because they make up the majority of users who are switching, and because high-quality tools for them are easy to come by.

Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. If you want the history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn’t perfect; and because it will take so long, you may as well do it right. The first problem is the author information. In Subversion, each person committing has a user on the system who is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` output and the `git svn log`. If you want to map this to better Git author data, you need a mapping from the Subversion users to the Git authors. Create a file called `users.txt` that has this mapping in a format like this:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that SVN uses, you can run this:

```
$ svn log --xml | grep author | sort -u | \
perl -pe 's/.*/(&.*?)<.*?/$1 = /'
```

That generates the log output in XML format, then keeps only the lines with author information, discards duplicates, strips out the XML tags. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your `users.txt` file so you can add the equivalent Git user data next to each entry.

You can provide this file to `git svn` to help it map the author data more accurately. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command (though if you want to keep the synchronisation-metadata, feel free to omit this parameter). This makes your `import` command look like this:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata -s my_project
```

Now you should have a nicer Subversion import in your `my_project` directory. Instead of commits that look like this

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk

git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-
373f-11de-
be05-5f7a86268029
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date: Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk
```

Not only does the Author field look a lot better, but the `git-svn-id` is no longer there, either.

You should also do a bit of post-import cleanup. For one thing, you should clean up the weird references that `git svn` set up. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run:

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

This takes the references that were remote branches that started with `remotes/origin/tags/` and makes them real (lightweight) tags.

Next, move the rest of the references under `refs/remotes` to be local branches:

```
$ cp -Rf .git/refs/remotes/origin/* .git/refs/heads/
$ rm -Rf .git/refs/remotes/origin
```

It may happen that you'll see some extra branches which are suffixed by `@xxx` (where xxx is a number), while in Subversion you only see one branch. This is actually a Subversion feature called

"peg-revisions", which is something that Git simply has no syntactical counterpart for. Hence, `git svn` simply adds the svn version number to the branch name just in the same way as you would have written it in svn to address the peg-revision of that branch. If you do not care anymore about the peg-revisions, simply remove them using `git branch -d`.

Now all the old branches are real Git branches and all the old tags are real Git tags.

There's one last thing to clean up. Unfortunately, `git svn` creates an extra branch named `trunk`, which maps to Subversion's default branch, but the `trunk` ref points to the same place as `master`. Since `master` is more idiomatically Git, here's how to remove the extra branch:

```
$ git branch -d trunk
```

The last thing to do is add your new Git server as a remote and push to it. Here is an example of adding your server as a remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Because you want all your branches and tags to go up, you can now run this:

```
$ git push origin --all  
$ git push origin --tags
```

All your branches and tags should be on your new Git server in a nice, clean import.

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool called "hg-fast-export", which you'll need a copy of:

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

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what it will put in the author field for changesets, so this is a good time to clean house. Generating this is a one-line command in a `bash` shell:

```
$ cd /tmp/hg-repo  
$ hg log | grep user: | sort | uniq | sed 's/user: */' > ../authors
```

This will take a few seconds, depending on how long your project's history is, and afterwards the `/tmp/authors` file will look something like this:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In this example, the same person (Bob) has created changesets under four different names, one of which actually looks correct, and one of which would be completely invalid for a Git commit. Hg-fast-export lets us fix this by adding `={new name and email address}` at the end of every line we want to change, and removing the lines for any usernames that we want to leave alone. If all the usernames look fine, we won't need this file at all. In this example, we want our file to look like this:

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
```

The next step is to create our new Git repository, and run the export script:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

The `-r` flag tells hg-fast-export where to find the Mercurial repository we want to convert, and the `-A` flag tells it where to find the author-mapping file. The script parses Mercurial changesets and converts them into a script for Git's "fast-import" feature (which we'll discuss in detail a bit later on). This takes a bit (though it's *much* faster than it would be over the network), and the output is fairly verbose:

```

$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0
added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0
added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0
added/changed/removed files
[...]
master: Exporting simple delta revision 22206/22208 with 0/4/0
added/changed/removed files
master: Exporting simple delta revision 22207/22208 with 0/2/0
added/changed/removed files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[...]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
-----
Alloc'd objects: 120000
Total objects: 115032 ( 208171 duplicates ) )
blobs : 40504 ( 205320 duplicates 26117 deltas of
39602 attempts)
trees : 52320 ( 2851 duplicates 47467 deltas of
47599 attempts)
commits: 22208 ( 0 duplicates 0 deltas of
0 attempts)
tags : 0 ( 0 duplicates 0 deltas of
0 attempts)
Total branches: 109 ( 2 loads )
marks: 1048576 ( 22208 unique )
atoms: 1952
Memory total: 7860 KiB
pools: 2235 KiB
objects: 5625 KiB
-----
pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 90430
pack_report: pack_mmap_calls = 46771
pack_report: pack_open_windows = 1 / 1
pack_report: pack_mapped = 340852700 / 340852700
-----
$ git shortlog -sn
 369 Bob Jones
 365 Joe Smith

```

That's pretty much all there is to it. All of the Mercurial tags have been converted to Git tags, and Mercurial branches and bookmarks have been converted to Git branches. Now you're ready to push the repository up to its new server-side home:

```
$ git remote add origin git@my-git-server:myrepository.git  
$ git push origin --all
```

Perforce

The next system you'll look at importing from is Perforce. As we discussed above, there are two ways to let Git and Perforce talk to each other: git-p4 and Perforce Git Fusion.

Perforce Git Fusion

Git Fusion makes this process fairly painless. Just configure your project settings, user mappings, and branches using a configuration file (as discussed in [Git Fusion](#)), and clone the repository. Git Fusion leaves you with what looks like a native Git repository, which is then ready to push to a native Git host if you desire. You could even use Perforce as your Git host if you like.

Git-p4

Git-p4 can also act as an import tool. As an example, we'll import the Jam project from the Perforce Public Depot. To set up your client, you must export the P4PORT environment variable to point to the Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

筆記

In order to follow along, you'll need a Perforce depot to connect with. We'll be using the public depot at public.perforce.com for our examples, but you can use any depot you have access to.

Run the `git p4 clone` command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import  
Importing from //guest/perforce_software/jam@all into p4import  
Initialized empty Git repository in /private/tmp/p4import/.git/  
Import destination: refs/remotes/p4/master  
Importing revision 9957 (100%)
```

This particular project has only one branch, but if you have branches that are configured with branch views (or just a set of directories), you can use the `--detect-branches` flag to `git p4 clone` to import all the project's branches as well. See [Branching](#) for a bit more detail on this.

At this point you're almost done. If you go to the `p4import` directory and run `git log`, you can see your imported work:

```

$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

[git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

[git-p4: depot-paths = "//public/jam/src/": change = 7304]

```

You can see that **git-p4** has left an identifier in each commit message. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so – before you start doing work on the new repository. You can use **git filter-branch** to remove the identifier strings en masse:

```

$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten

```

If you run **git log**, you can see that all the SHA-1 checksums for the commits have changed, but the **git-p4** strings are no longer in the commit messages:

```

$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

```

Your import is ready to push up to your new Git server.

TFS

If your team is converting their source control from TFVC to Git, you'll want the highest-fidelity conversion you can get. This means that, while we covered both `git-tfs` and `git-tf` for the interop

section, we'll only be covering git-tfs for this part, because git-tfs supports branches, and this is prohibitively difficult using git-tf.

筆記

This is a one-way conversion. The resulting Git repository won't be able to connect with the original TFVC project.

The first thing to do is map usernames. TFVC is fairly liberal with what goes into the author field for changesets, but Git wants a human-readable name and email address. You can get this information from the `tf` command-line client, like so:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

This grabs all of the changesets in the history of the project and put it in the `AUTHORS_TMP` file that we will process to extract the data of the *User* column (the 2nd one). Open the file and find at which characters start and end the column and replace, in the following command-line, the parameters **11-20** of the `cut` command with the ones found:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | sort | uniq > AUTHORS
```

The `cut` command keeps only the characters between 11 and 20 from each line. The `tail` command skips the first two lines, which are field headers and ASCII-art underlines. The result of all of this is piped to `sort` and `uniq` to eliminate duplicates, and saved to a file named `AUTHORS`. The next step is manual; in order for git-tfs to make effective use of this file, each line must be in this format:

```
DOMAIN\username = User Name <email@address.com>
```

The portion on the left is the “User” field from TFVC, and the portion on the right side of the equals sign is the user name that will be used for Git commits.

Once you have this file, the next thing to do is make a full clone of the TFVC project you're interested in:

```
PS> git tfs clone --with-branches --authors=AUTHORS  
https://username.visualstudio.com/DefaultCollection $/project/Trunk  
project_git
```

Next you'll want to clean the `git-tfs-id` sections from the bottom of the commit messages. The following command will do that:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$/g"' '--'  
--all
```

That uses the `sed` command from the Git-bash environment to replace any line starting with “git-tfs-id:” with emptiness, which Git will then ignore.

Once that's all done, you're ready to add a new remote, push all your branches up, and have your team start working from Git.

A Custom Importer

If your system isn't one of the above, you should look for an importer online – quality importers are available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see [Git Internals](#) for more information). This way, you can write an import script that reads the necessary information out of the system you're importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you'll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As we did in [An Example Git-Enforced Policy](#), we'll write this in Ruby, because it's what we generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with – it just needs to print the appropriate information to `stdout`. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end of your lines – `git fast-import` is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```

last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end

```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly.

“Mark” is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You’ll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```

$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end

```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you’ll parse it out. The next line in your `print_export` file is:

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as:

```

def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end

```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'John Doe <john@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```

# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark

```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word data, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```

def export_data(string)
  print "data #{string.size}\n#{string}"
end

```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory – you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, fast-import can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex – you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the [fast-import](#) man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while git fast-import expects only LF. To get around this problem and make git fast-import happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```

#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"


$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    puts 'commit refs/heads/master'
    puts "mark :#{mark}"
    puts "committer #$author #{date} -0700"
    export_data("imported from #{dir}")
    puts "from :#{last_mark}" if last_mark

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
    end
    mark
end

# Loop through the directories

```

```

last_mark = nil
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end

```

If you run this script, you'll get content that looks something like this:

```

$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your script:

```

$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:        13 (          6 duplicates      )
    blobs :           5 (          4 duplicates      ) 3 deltas of
5 attempts)
    trees :           4 (          1 duplicates      ) 0 deltas of
4 attempts)
    commits:          4 (          1 duplicates      ) 0 deltas of
0 attempts)
    tags   :           0 (          0 duplicates      ) 0 deltas of
0 attempts)
Total branches:       1 (          1 loads      )
    marks:            1024 (         5 unique      )
    atoms:             2
Memory total:        2344 KiB
    pools:            2110 KiB
    objects:          234 KiB
-----
pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 10
pack_report: pack_mmap_calls = 5
pack_report: pack_open_windows = 2 / 2
pack_report: pack_mapped = 1457 / 1457
-----
```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 13 objects total for 4 commits into 1 branch. Now, you can run `git log` to see your new history:

```

$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

There you go – a nice, clean Git repository. It’s important to note that nothing is checked out – you don’t have any files in your working directory at first. To get them, you must reset your branch to

where `master` is now:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

You can do a lot more with the `fast-import` tool – handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code.

Summary

You should feel comfortable using Git as a client for other version-control systems, or importing nearly any existing repository into Git without losing data. In the next chapter, we'll cover the raw internals of Git so you can craft every single byte, if need be.

Git Internals

You may have skipped to this chapter from a previous chapter, or you may have gotten here after reading the rest of the book – in either case, this is where we’ll go over the inner workings and implementation of Git. We found that learning this information was fundamentally important to understanding how useful and powerful Git is, but others have argued to us that it can be confusing and unnecessarily complex for beginners. Thus, we’ve made this discussion the last chapter in the book so you could read it early or later in your learning process. We leave it up to you to decide.

Now that you’re here, let’s get started. First, if it isn’t yet clear, Git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it. You’ll learn more about what this means in a bit.

In the early days of Git (mostly pre 1.5), the user interface was much more complex because it emphasized this filesystem rather than a polished VCS. In the last few years, the UI has been refined until it’s as clean and easy to use as any system out there; but often, the stereotype lingers about the early Git UI that was complex and difficult to learn.

The content-addressable filesystem layer is amazingly cool, so we’ll cover that first in this chapter; then, you’ll learn about the transport mechanisms and the repository maintenance tasks that you may eventually have to deal with.

Plumbing and Porcelain

This book covers how to use Git with 30 or so verbs such as `checkout`, `branch`, `remote`, and so on. But because Git was initially a toolkit for a VCS rather than a full user-friendly VCS, it has a bunch of verbs that do low-level work and were designed to be chained together UNIX style or called from scripts. These commands are generally referred to as “plumbing” commands, and the more user-friendly commands are called “porcelain” commands.

The book’s first nine chapters deal almost exclusively with porcelain commands. But in this chapter, you’ll be dealing mostly with the lower-level plumbing commands, because they give you access to the inner workings of Git, and help demonstrate how and why Git does what it does. Many of these commands aren’t meant to be used manually on the command line, but rather to be used as building blocks for new tools and custom scripts.

When you run `git init` in a new or existing directory, Git creates the `.git` directory, which is where almost everything that Git stores and manipulates is located. If you want to back up or clone your repository, copying this single directory elsewhere gives you nearly everything you need. This entire chapter basically deals with the stuff in this directory. Here’s what it looks like:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

You may see some other files in there, but this is a fresh `git init` repository – it’s what you see by default. The `description` file is only used by the GitWeb program, so don’t worry about it. The `config` file contains your project-specific configuration options, and the `info` directory keeps a global exclude file for ignored patterns that you don’t want to track in a `.gitignore` file. The `hooks` directory contains your client- or server-side hook scripts, which are discussed in detail in [Git Hooks](#).

This leaves four important entries: the `HEAD` and (yet to be created) `index` files, and the `objects` and `refs` directories. These are the core parts of Git. The `objects` directory stores all the content for your database, the `refs` directory stores pointers into commit objects in that data (branches), the `HEAD` file points to the branch you currently have checked out, and the `index` file is where Git stores your staging area information. You’ll now look at each of these sections in detail to see how Git operates.

Git Objects

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time. To demonstrate, you can use the plumbing command `hash-object`, which takes some data, stores it in your `.git` directory, and gives you back the key the data is stored as. First, you initialize a new Git repository and verify that there is nothing in the `objects` directory:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git has initialized the `objects` directory and created `pack` and `info` subdirectories in it, but there are no regular files. Now, store some text in your Git database:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

The `-w` tells `hash-object` to store the object; otherwise, the command simply tells you what the key would be. `--stdin` tells the command to read the content from stdin; if you don’t specify this, `hash-object` expects a file path at the end. The output from the command is a 40-character checksum hash. This is the SHA-1 hash – a checksum of the content you’re storing plus a header, which you’ll learn about in a bit. Now you can see how Git has stored your data:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

You can see a file in the `objects` directory. This is how Git stores the content initially – as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory

is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

You can pull the content back out of Git with the `cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to it instructs the `cat-file` command to figure out the type of content and display it nicely for you:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4  
test content
```

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

```
$ echo 'version 1' > test.txt  
$ git hash-object -w test.txt  
83baae61804e65cc73a7201a7252750c76066a30
```

Then, write some new content to the file, and save it again:

```
$ echo 'version 2' > test.txt  
$ git hash-object -w test.txt  
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Your database contains the two new versions of the file as well as the first content you stored there:

```
$ find .git/objects -type f  
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a  
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Now you can revert the file back to the first version

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt  
$ cat test.txt  
version 1
```

or the second version:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt  
$ cat test.txt  
version 2
```

But remembering the SHA-1 key for each version of your file isn't practical; plus, you aren't storing the filename in your system – just the content. This object type is called a blob. You can have Git tell

you the object type of any object in Git, given its SHA-1 key, with `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Objects

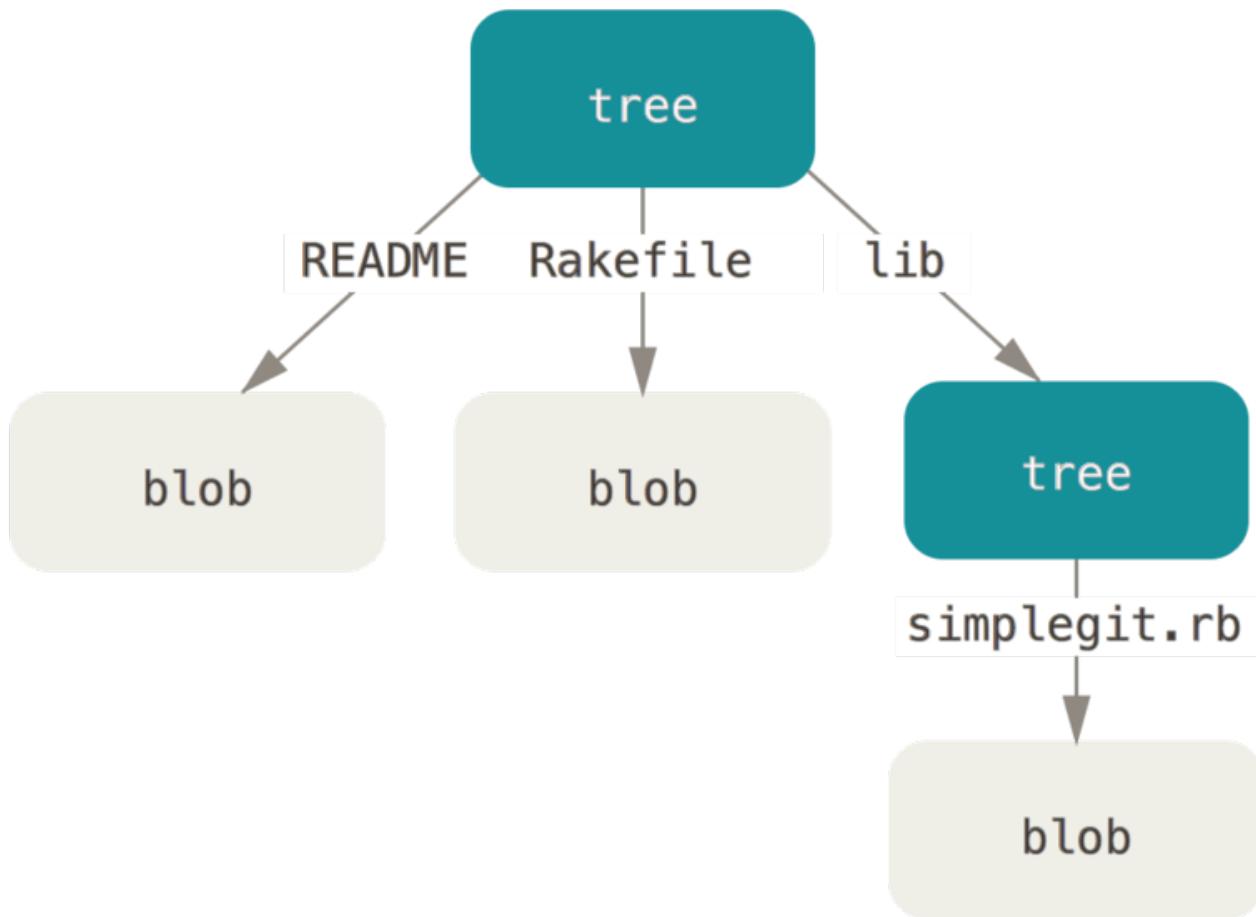
The next type we'll look at is the tree, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more tree entries, each of which contains a SHA-1 pointer to a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptually, the data that Git is storing is something like this:



圖表 148. Simple version of the Git data model.

You can fairly easily create your own tree. Git normally creates a tree by taking the state of your staging area or index and writing a series of tree objects from it. So, to create a tree object, you first have to set up an index by staging some files. To create an index with a single entry – the first version of your `test.txt` file – you can use the plumbing command `update-index`. You use this command to artificially add the earlier version of the `test.txt` file to a new staging area. You must pass it the `--add` option because the file doesn't yet exist in your staging area (you don't even have a staging area set up yet) and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, you specify the mode, SHA-1, and filename:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In this case, you're specifying a mode of `100644`, which means it's a normal file. Other options are `100755`, which means it's an executable file; and `120000`, which specifies a symbolic link. The mode is taken from normal UNIX modes but is much less flexible – these three modes are the only ones that are valid for files (blobs) in Git (although other modes are used for directories and submodules).

Now, you can use the `write-tree` command to write the staging area out to a tree object. No `-w` option is needed – calling `write-tree` automatically creates a tree object from the state of the index if that tree doesn't yet exist:

```
$ git write-tree  
d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

You can also verify that this is a tree object:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
tree
```

You'll now create a new tree with the second version of `test.txt` and a new file as well:

```
$ echo 'new file' > new.txt  
$ git update-index test.txt  
$ git update-index --add new.txt
```

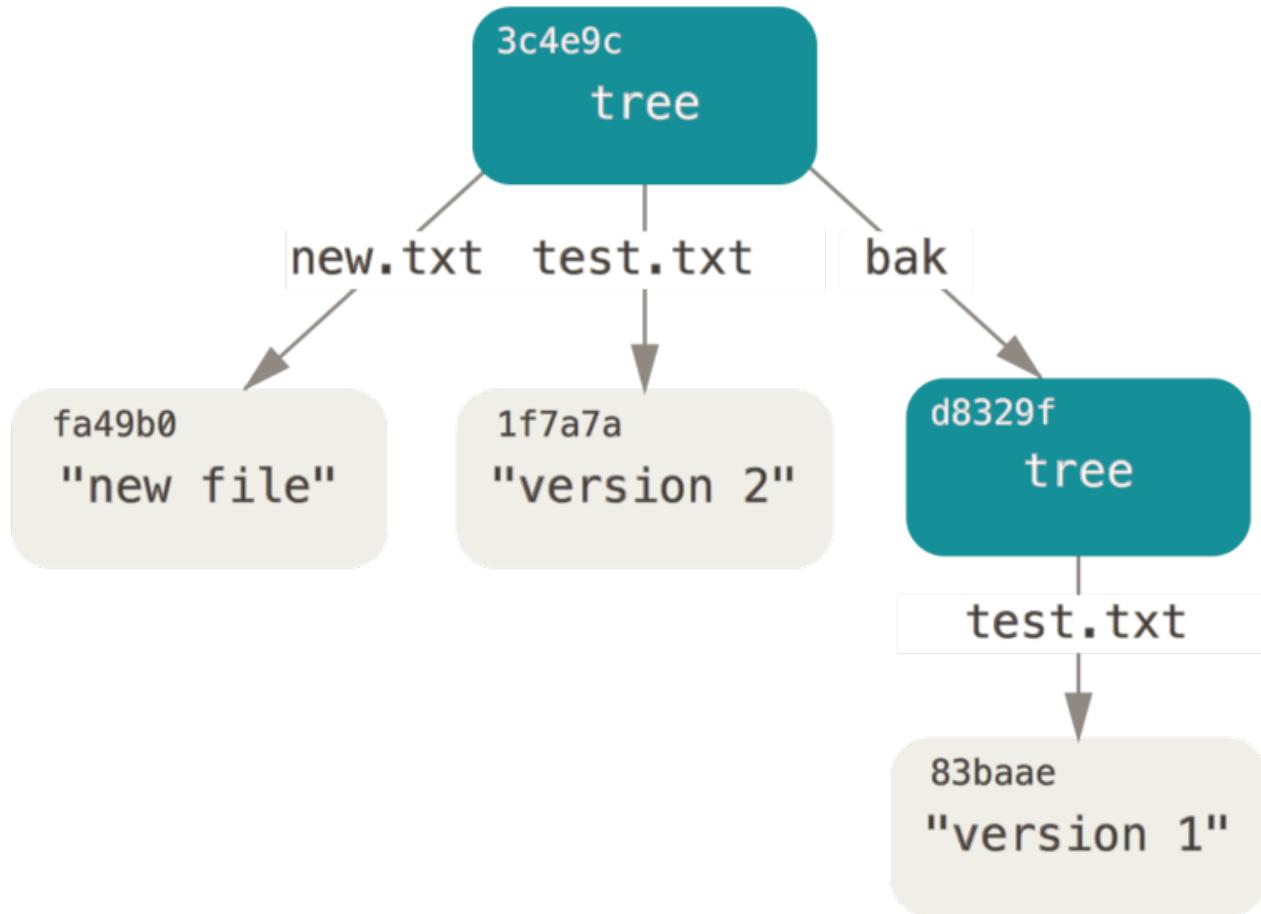
Your staging area now has the new version of `test.txt` as well as the new file `new.txt`. Write out that tree (recording the state of the staging area or index to a tree object) and see what it looks like:

```
$ git write-tree  
0155eb4229851634a0f03eb265b69f5a2d56f341  
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341  
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt  
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Notice that this tree has both file entries and also that the `test.txt` SHA-1 is the “version 2” SHA-1 from earlier ([1f7a7a](#)). Just for fun, you'll add the first tree as a subdirectory into this one. You can read trees into your staging area by calling `read-tree`. In this case, you can read an existing tree into your staging area as a subtree by using the `--prefix` option to `read-tree`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
$ git write-tree  
3c4e9cd789d88d8d89c1073707c3585e41b0e614  
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614  
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak  
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt  
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named `bak` that contained the first version of the `test.txt` file. You can think of the data that Git contains for these structures as being like this:



图表 149. The content structure of your current Git data.

Commit Objects

You have three trees that specify the different snapshots of your project that you want to track, but the earlier problem remains: you must remember all three SHA-1 values in order to recall the snapshots. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores for you.

To create a commit object, you call `commit-tree` and specify a single tree SHA-1 and which commit objects, if any, directly preceded it. Start with the first tree you wrote:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

You will get a different hash value because of different creation time and author data. Replace commit and tag hashes with your own checksums further in this chapter. Now you can look at your new commit object with `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the author/committer information (which uses your `user.name` and `user.email` configuration settings and a timestamp); a blank line, and then the commit message.

Next, you'll write the other two commit objects, each referencing the commit that came directly before it:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cf9
```

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you run it on the last commit SHA-1:

```

$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

bак/test.txt | 1 +
1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

new.txt | 1 +
test.txt | 2 +-
2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

test.txt | 1 +
1 file changed, 1 insertion(+)

```

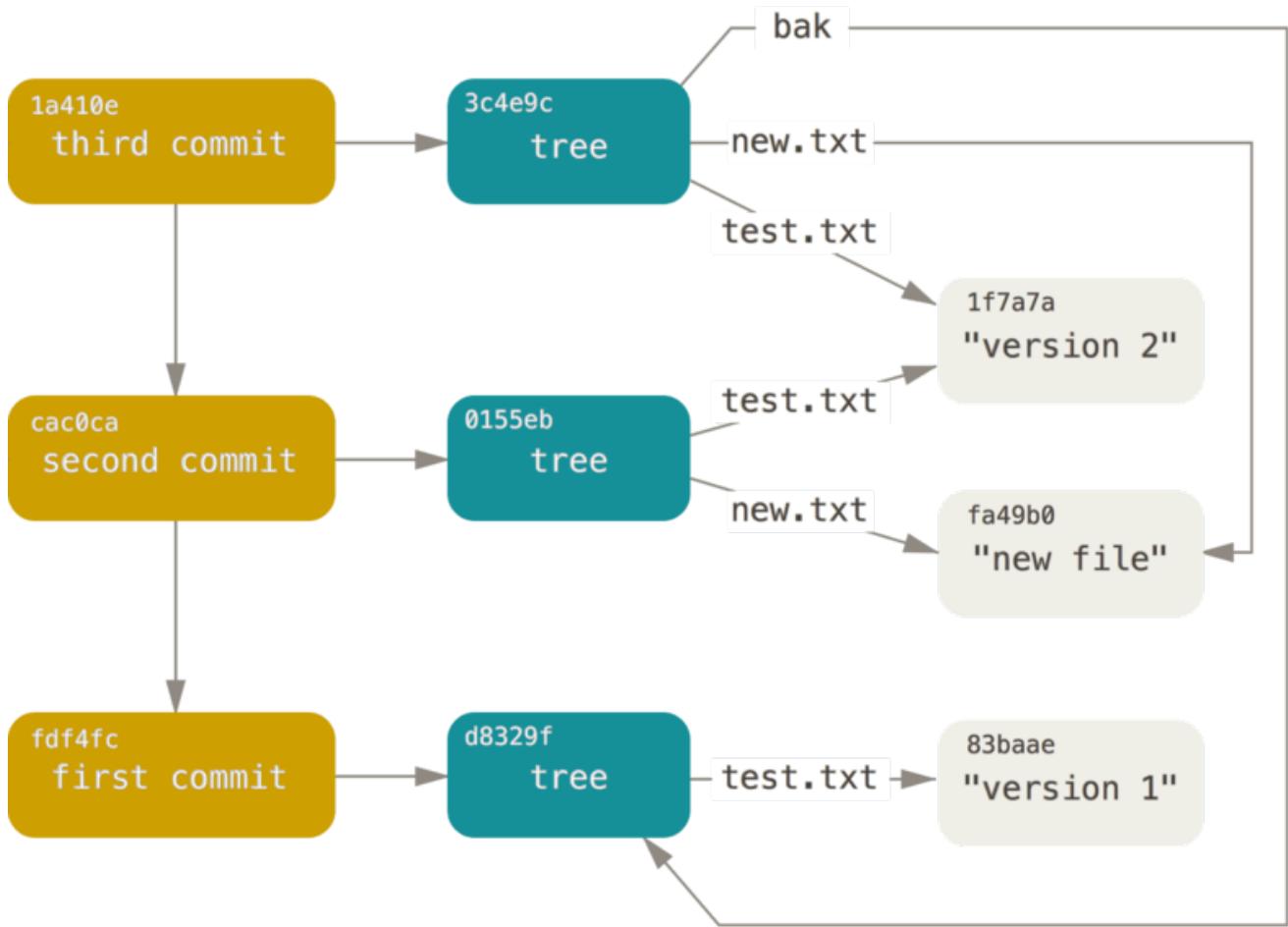
Amazing. You've just done the low-level operations to build up a Git history without using any of the front end commands. This is essentially what Git does when you run the `git add` and `git commit` commands – it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects – the blob, the tree, and the commit – are initially stored as separate files in your `.git/objects` directory. Here are all the objects in the example directory now, commented with what they store:

```

$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1

```

If you follow all the internal pointers, you get an object graph something like this:



圖表 150. All the objects in your Git directory.

Object Storage

We mentioned earlier that a header is stored with the content. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object – in this case, the string “what is up, doc?” – interactively in the Ruby scripting language.

You can start up interactive Ruby mode with the `irb` command:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\0000"
```

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. You can calculate the SHA-1 value of a string in Ruby by including the SHA1 digest library with the `require` command and then calling `Digest::SHA1hexdigest()` with the string:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresses the new content with zlib, which you can do in Ruby with the zlib library. First, you need to require the library and then run `Zlib::Deflate.deflate()` on the content:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC90R04c(\xCFH,Q\xC8,V(-\xD0QH\xC90\xB6\aN\x00_\x1C\aN\x9D"
```

Finally, you'll write your zlib-deflated content to an object on disk. You'll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously zlib-compressed content to the file with a `write()` call on the resulting file handle:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

That's it – you've created a valid Git blob object. All Git objects are stored the same way, just with different types – instead of the string blob, the header will begin with commit or tree. Also, although the blob content can be nearly anything, the commit and tree content are very specifically formatted.

Git References

You can run something like `git log 1a410e` to look through your whole history, but you still have to remember that `1a410e` is the last commit in order to walk that history to find all those objects. You need a file in which you can store the SHA-1 value under a simple name so you can use that pointer rather than the raw SHA-1 value.

In Git, these are called “references” or “refs”; you can find the files that contain the SHA-1 values in the `.git/refs` directory. In the current project, this directory contains no files, but it does contain a simple structure:

```
$ find .git/refs  
.git/refs  
.git/refs/heads  
.git/refs/tags  
$ find .git/refs -type f
```

To create a new reference that will help you remember where your latest commit is, you can technically do something as simple as this:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" >  
.git/refs/heads/master
```

Now, you can use the head reference you just created instead of the SHA-1 value in your Git commands:

```
$ git log --pretty=oneline master  
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit  
cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You aren't encouraged to directly edit the reference files. Git provides a safer command to do this if you want to update a reference called **update-ref**:

```
$ git update-ref refs/heads/master  
1a410efbd13591db07496601ebc7a059dd55cfe9
```

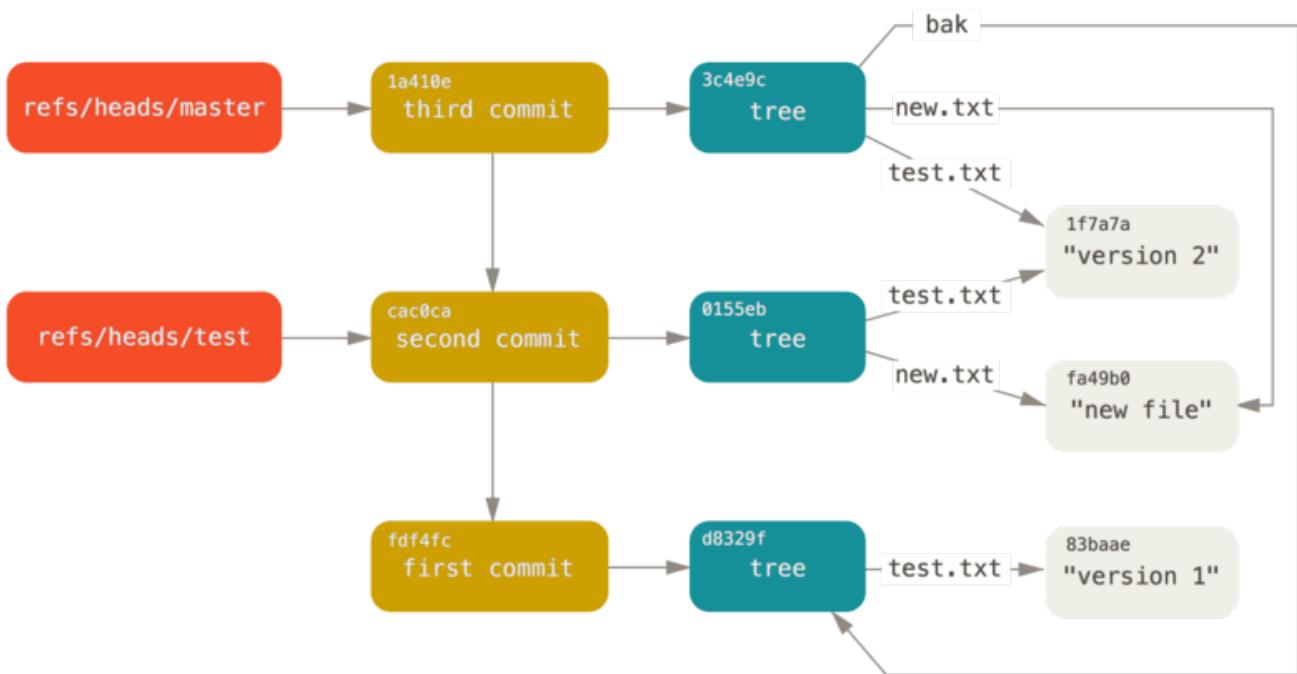
That's basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit, you can do this:

```
$ git update-ref refs/heads/test cac0ca
```

Your branch will contain only work from that commit down:

```
$ git log --pretty=oneline test  
cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, your Git database conceptually looks something like this:



圖表 151. Git directory objects with branch head references included.

When you run commands like `git branch (branchname)`, Git basically runs that `update-ref` command to add the SHA-1 of the last commit of the branch you’re on into whatever new reference you want to create.

The HEAD

The question now is, when you run `git branch (branchname)`, how does Git know the SHA-1 of the last commit? The answer is the HEAD file.

The HEAD file is a symbolic reference to the branch you’re currently on. By symbolic reference, we mean that unlike a normal reference, it doesn’t generally contain a SHA-1 value but rather a pointer to another reference. If you look at the file, you’ll normally see something like this:

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you run `git checkout test`, Git updates the file to look like this:

```
$ cat .git/HEAD
ref: refs/heads/test
```

When you run `git commit`, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to.

You can also manually edit this file, but again a safer command exists to do so: `symbolic-ref`. You can read the value of your HEAD via this command:

```
$ git symbolic-ref HEAD  
refs/heads/master
```

You can also set the value of HEAD:

```
$ git symbolic-ref HEAD refs/heads/test  
$ cat .git/HEAD  
ref: refs/heads/test
```

You can't set a symbolic reference outside of the refs style:

```
$ git symbolic-ref HEAD test  
fatal: Refusing to point HEAD outside of refs/
```

Tags

We just finished discussing Git's three main object types, but there is a fourth. The tag object is very much like a commit object – it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It's like a branch reference, but it never moves – it always points to the same commit but gives it a friendlier name.

As discussed in [Git 基礎](#), there are two types of tags: annotated and lightweight. You can make a lightweight tag by running something like this:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

That is all a lightweight tag is – a reference that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (`-a` specifies that it's an annotated tag):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Here's the object SHA-1 value it created:

```
$ cat .git/refs/tags/v1.1  
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Now, run the `cat-file` command on that SHA-1 value:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700
test tag
```

Notice that the object entry points to the commit SHA-1 value that you tagged. Also notice that it doesn't need to point to a commit; you can tag any Git object. In the Git source code, for example, the maintainer has added their GPG public key as a blob object and then tagged it. You can view the public key by running this in a clone of the Git repository:

```
$ git cat-file blob junio-gpg-pub
```

The Linux kernel repository also has a non-commit-pointing tag object – the first tag created points to the initial tree of the import of the source code.

Remotes

The third type of reference that you'll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the `refs/remotes` directory. For instance, you can add a remote called `origin` and push your `master` branch to it:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

Then, you can see what the `master` branch on the `origin` remote was the last time you communicated with the server, by checking the `refs/remotes/origin/master` file:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references differ from branches (`refs/heads` references) mainly in that they're considered read-only. You can `git checkout` to one, but Git won't point HEAD at one, so you'll never update it with a `commit` command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

Packfiles

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects – 4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresses the contents of these files with zlib, and you're not storing much, so all these files collectively take up only 925 bytes. You'll add some larger content to the repository to demonstrate an interesting feature of Git. To demonstrate, we'll add the `repo.rb` file from the Grit library – this is about a 22K source code file:

```
$ curl
https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ git checkout master
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

If you look at the resulting tree, you can see the SHA-1 value your `repo.rb` file got for the blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

You can then use `git cat-file` to see how big that object is:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Now, modify that file a little, and see what happens:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Check the tree created by that commit, and you see something interesting:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

The blob is now a different blob, which means that although you added only a single line to the end of a 400-line file, Git stored that new content as a completely new object:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

You have two nearly identical 22K objects on your disk (each compressed to approximately 7K). Wouldn't it be nice if Git could store one of them in full but then the second object only as the delta between it and the first?

It turns out that it can. The initial format in which Git saves objects on disk is called a “loose” object format. However, occasionally Git packs up several of these objects into a single binary file called a “packfile” in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the `git gc` command manually, or if you push to a remote server. To see what happens, you can manually ask Git to pack up the objects by calling the `git gc` command:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

If you look in your `objects` directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

The objects that remain are the blobs that aren't pointed to by any commit – in this case, the “what is up, doc?” example and the “test content” example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the `gc` were collectively about 15K in size, the new packfile is only 7K. You've cut your disk usage by half by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed up:

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
    deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
    deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \
    b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Here, the `033b4` blob, which if you remember was the first version of your `repo.rb` file, is referencing the `b042a` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you can see that `b042a` takes up 22K of the file, but that `033b4` only takes up 9 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta – this is because you’re most likely to need faster access to the most recent version of the file.

The really nice thing about this is that it can be repacked at any time. Git will occasionally repack your database automatically, always trying to save more space, but you can also manually repack at any time by running `git gc` by hand.

The Refspec

Throughout this book, we’ve used simple mappings from remote branches to local references, but they can be more complex. Suppose you add a remote like this:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

It adds a section to your `.git/config` file, specifying the name of the remote (`origin`), the URL of the remote repository, and the refspec for fetching:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
```

The format of the refspec is an optional `+`, followed by `<src>:<dst>`, where `<src>` is the pattern for references on the remote side and `<dst>` is where those references will be written locally. The `+` tells Git to update the reference even if it isn’t a fast-forward.

In the default case that is automatically written by a `git remote add` command, Git fetches all the references under `refs/heads/` on the server and writes them to `refs/remotes/origin/` locally. So, if there is a `master` branch on the server, you can access the log of that branch locally via

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

They’re all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

If you want Git instead to pull down only the `master` branch each time, and not every other branch on the remote server, you can change the fetch line to

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

This is just the default refspec for `git fetch` for that remote. If you want to do something one time, you can specify the refspec on the command line, too. To pull the `master` branch on the remote down

to `origin/mymaster` locally, you can run

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

You can also specify multiple refspecs. On the command line, you can pull down several branches like so:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
  topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]      master    -> origin/mymaster  (non fast forward)
 * [new branch]    topic     -> origin/topic
```

In this case, the `master` branch pull was rejected because it wasn't a fast-forward reference. You can override that by specifying the `+` in front of the refspec.

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the `master` and `experiment` branches, add two lines:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

You can't use partial globs in the pattern, so this would be invalid:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

However, you can use namespaces (or directories) to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the `master` branch and any of the QA team's branches but nothing else, you can use a config section like this:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a `qa/` namespace in the first place? You accomplish that by using refspecs to push.

If the QA team wants to push their `master` branch to `qa/master` on the remote server, they can run

```
$ git push origin master:refs/heads/qa/master
```

If they want Git to do that automatically each time they run `git push origin`, they can add a `push` value to their config file:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
push = refs/heads/master:refs/heads/qa/master
```

Again, this will cause a `git push origin` to push the local `master` branch to the remote `qa/master` branch by default.

Deleting References

You can also use the refspec to delete references from the remote server by running something like this:

```
$ git push origin :topic
```

Because the refspec is `<src>:<dst>`, by leaving off the `<src>` part, this basically says to make the `topic` branch on the remote nothing, which deletes it.

Transfer Protocols

Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

If you’re setting up a repository to be served read-only over HTTP, the dumb protocol is likely what will be used. This protocol is called “dumb” because it requires no Git-specific code on the server side during the transport process; the fetch process is a series of HTTP `GET` requests, where the client can assume the layout of the Git repository on the server.

筆記 The dumb protocol is fairly rarely used these days. It’s difficult to secure or make private, so most Git hosts (both cloud-based and on-premises) will refuse to use it. It’s generally advised to use the smart protocol, which we describe a bit further on.

Let’s follow the `http-fetch` process for the simplegit library:

```
$ git clone http://server/simplegit-progit.git
```

The first thing this command does is pull down the `info/refs` file. This file is written by the `update-`

`server-info` command, which is why you need to enable that as a `post-receive` hook in order for the HTTP transport to work properly:

```
=> GET info/refs  
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Now you have a list of the remote references and SHA-1s. Next, you look for what the HEAD reference is so you know what to check out when you’re finished:

```
=> GET HEAD  
ref: refs/heads/master
```

You need to check out the `master` branch when you’ve completed the process. At this point, you’re ready to start the walking process. Because your starting point is the `ca82a6` commit object you saw in the `info/refs` file, you start by fetching that:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949  
(179 bytes of binary data)
```

You get an object back – that object is in loose format on the server, and you fetched it over a static HTTP GET request. You can zlib-uncompress it, strip off the header, and look at the commit content:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949  
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf  
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
author Scott Chacon <schacon@gmail.com> 1205815931 -0700  
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700  
  
changed the version number
```

Next, you have two more objects to retrieve – `cfda3b`, which is the tree of content that the commit we just retrieved points to; and `085bb3`, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
(179 bytes of data)
```

That gives you your next commit object. Grab the tree object:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf  
(404 - Not Found)
```

Oops – it looks like that tree object isn’t in loose format on the server, so you get a 404 response back. There are a couple of reasons for this – the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

```
=> GET objects/info/http-alternates  
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there – this is a nice mechanism for projects that are forks of one another to share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the [objects/info/packs](#) file, which contains a listing of them (also generated by [update-server-info](#)):

```
=> GET objects/info/packs  
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx  
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it – because the index lists the SHA-1s of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the [master](#) branch that was pointed to by the HEAD reference you downloaded at the beginning.

The Smart Protocol

The dumb protocol is simple but a bit inefficient, and it can't handle writing of data from the client to the server. The smart protocol is a more common method of transferring data, but it requires a process on the remote end that is intelligent about Git – it can read local data, figure out what the client has and needs, and generate a custom packfile for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

Uploading Data

To upload data to a remote process, Git uses the [send-pack](#) and [receive-pack](#) processes. The [send-pack](#) process runs on the client and connects to a [receive-pack](#) process on the remote side.

SSH

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which initiates a connection over SSH to your server. It tries to run a command on the remote server via an SSH call that looks something like this:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"  
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master report-  
status \  
    delete-refs side-band-64k quiet ofs-delta \  
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs  
0000
```

The `git-receive-pack` command immediately responds with one line for each reference it currently has – in this case, just the `master` branch and its SHA-1. The first line also has a list of the server’s capabilities (here, `report-status`, `delete-refs`, and some others, including the client identifier).

Each line starts with a 4-character hex value specifying how long the rest of the line is. Your first line starts with `00a5`, which is hexadecimal for 165, meaning that 165 bytes remain on that line. The next line is `0000`, meaning the server is done with its references listing.

Now that it knows the server’s state, your `send-pack` process determines what commits it has that the server doesn’t. For each reference that this push will update, the `send-pack` process tells the `receive-pack` process that information. For instance, if you’re updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like this:

```
0076ca82a6dff817ec66f44342007202690a93763949  
15027957951b64cf874c3557a0f3547bd83b3ff6 \  
    refs/heads/master report-status  
006c0000000000000000000000000000000000000000000000000000000  
cdfdb42577e2506715f8cfeacdbabc092bf63e8d \  
    refs/heads/experiment  
0000
```

Git sends a line for each reference you’re updating with the line’s length, the old SHA-1, the new SHA-1, and the reference that is being updated. The first line also has the client’s capabilities. The SHA-1 value of all ‘0’ s means that nothing was there before – because you’re adding the experiment reference. If you were deleting a reference, you would see the opposite: all ‘0’ s on the right side.

Next, the client sends a packfile of all the objects the server doesn’t have yet. Finally, the server responds with a success (or failure) indication:

```
000eunpack ok
```

HTTP(S)

This process is mostly the same over HTTP, though the handshaking is a bit different. The connection is initiated with this request:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack  
001f# service=git-receive-pack  
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master□report-status \  
    delete-refs side-band-64k quiet ofs-delta \  
    agent=git/2:2.1.1~vgm-bitmaps-bugaloo-608-g116744e  
0000
```

That's the end of the first client-server exchange. The client then makes another request, this time a **POST**, with the data that **send-pack** provides.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

The **POST** request includes the **send-pack** output and the packfile as its payload. The server then indicates success or failure with its HTTP response.

Downloading Data

When you download data, the **fetch-pack** and **upload-pack** processes are involved. The client initiates a **fetch-pack** process that connects to an **upload-pack** process on the remote side to negotiate what data will be transferred down.

SSH

If you're doing the fetch over SSH, **fetch-pack** runs something like this:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

After **fetch-pack** connects, **upload-pack** sends back something like this:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \  
    side-band side-band-64k ofs-delta shallow no-progress include-tag \  
    multi_ack_detailed symref=HEAD:refs/heads/master \  
    agent=git/2:2.1.1+github-607-gfba4028  
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master  
0000
```

This is very similar to what **receive-pack** responds with, but the capabilities are different. In addition, it sends back what HEAD points to (**symref=HEAD:refs/heads/master**) so the client knows what to check out if this is a clone.

At this point, the **fetch-pack** process looks at what objects it has and responds with the objects that it needs by sending “want” and then the SHA-1 it wants. It sends all the objects it already has with “have” and then the SHA-1. At the end of this list, it writes “done” to initiate the **upload-pack** process to begin sending the packfile of the data it needs:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

The handshake for a fetch operation takes two HTTP requests. The first is a `GET` to the same endpoint used in the dumb protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fc82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

This is very similar to invoking `git-upload-pack` over an SSH connection, but the second exchange is performed as a separate request:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fd9a93eb2908e52742248faf0ee993
0000
```

Again, this is the same format as above. The response to this request indicates success or failure, and includes the packfile.

Protocols Summary

This section contains a very basic overview of the transfer protocols. The protocol includes many other features, such as `multi_ack` or `side-band` capabilities, but covering them is outside the scope of this book. We've tried to give you a sense of the general back-and-forth between client and server; if you need more knowledge than this, you'll probably want to take a look at the Git source code.

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called “auto gc”. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The “gc” stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it

consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

You can run auto gc manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real gc command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in [Git References](#). You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    modified repo.rb a bit
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named **recover-branch** at that commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Cool – now you have a branch named **recover-branch** that is where your **master** branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing **recover-branch** and deleting the reflog. Now the first two commits aren’t reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the **.git/logs/** directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the **git fsck** utility, which checks your database for integrity. If you run it with the **--full** option, it shows you all objects that aren’t pointed to by another object:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling treeaea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the string “dangling commit” . You can recover it the same way, by adding a branch that points to that SHA-1.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a [git clone](#) downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it’s reachable from the history, it will always be there.

This can be a huge problem when you’re converting Subversion or Perforce repositories into Git. Because you don’t download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you’re fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you’ll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz >
git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oops – you didn’t want to add a huge tarball to your project. Better get rid of it:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Now, `gc` your database and see how much space you're using:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-29..69.idx \
| sort -k 3 -n \
| tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob 4975916 4976258 1438
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in [Enforcing a Specific Commit-Message Format](#). If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 82c99a3  
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --oneline --branches -- git.tgz  
dadf725 oops - removed large tarball  
7b30847 add git tarball
```

You must rewrite all the commits downstream from [7b30847](#) to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in [Rewriting History](#):

```
$ git filter-branch --index-filter \  
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..  
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2) rm 'git.tgz'  
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)  
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in [Rewriting History](#), except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` – you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the [7b30847](#) commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ rm -Rf .git/refs/original  
$ rm -Rf .git/logs/  
$ git gc  
Counting objects: 15, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (11/11), done.  
Writing objects: 100% (15/15), done.  
Total 15 (delta 1), reused 12 (delta 0)
```

Let's see how much space you saved.

```
$ git count-objects -v  
count: 11  
size: 4904  
in-pack: 15  
packs: 1  
size-pack: 8  
prune-packable: 0  
garbage: 0  
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ git prune --expire now  
$ git count-objects -v  
count: 0  
size: 0  
in-pack: 15  
packs: 1  
size-pack: 8  
prune-packable: 0  
garbage: 0  
size-garbage: 0
```

Environment Variables

Git always runs inside a `bash` shell, and uses a number of shell environment variables to determine how it behaves. Occasionally, it comes in handy to know what these are, and how they can be used to make Git behave the way you want it to. This isn't an exhaustive list of all the environment variables Git pays attention to, but we'll cover the most useful.

Global Behavior

Some of Git's general behavior as a computer program depends on environment variables.

GIT_EXEC_PATH determines where Git looks for its sub-programs (like `git-commit`, `git-diff`, and others). You can check the current setting by running `git --exec-path`.

HOME isn't usually considered customizable (too many other things depend on it), but it's where Git looks for the global configuration file. If you want a truly portable Git installation, complete with global configuration, you can override **HOME** in the portable Git's shell profile.

PREFIX is similar, but for the system-wide configuration. Git looks for this file at `$PREFIX/etc/gitconfig`.

GIT_CONFIG_NOSYSTEM, if set, disables the use of the system-wide configuration file. This is useful if your system config is interfering with your commands, but you don't have access to change or remove it.

GIT_PAGER controls the program used to display multi-page output on the command line. If this is unset, **PAGER** will be used as a fallback.

GIT_EDITOR is the editor Git will launch when the user needs to edit some text (a commit message, for example). If unset, **EDITOR** will be used.

Repository Locations

Git uses several environment variables to determine how it interfaces with the current repository.

GIT_DIR is the location of the `.git` folder. If this isn't specified, Git walks up the directory tree until it gets to `~` or `/`, looking for a `.git` directory at every step.

GIT_CEILING_DIRECTORIES controls the behavior of searching for a `.git` directory. If you access directories that are slow to load (such as those on a tape drive, or across a slow network connection), you may want to have Git stop trying earlier than it might otherwise, especially if Git is invoked when building your shell prompt.

GIT_WORK_TREE is the location of the root of the working directory for a non-bare repository. If not specified, the parent directory of **GIT_DIR** is used.

GIT_INDEX_FILE is the path to the index file (non-bare repositories only).

GIT_OBJECT_DIRECTORY can be used to specify the location of the directory that usually resides at `.git/objects`.

GIT_ALTERNATE_OBJECT_DIRECTORIES is a colon-separated list (formatted like `/dir/one:/dir/two:...`) which tells Git where to check for objects if they aren't in **GIT_OBJECT_DIRECTORY**. If you happen to have a lot of projects with large files that have the exact same contents, this can be used to avoid storing too many copies of them.

Pathspecs

A “pathspec” refers to how you specify paths to things in Git, including the use of wildcards. These are used in the `.gitignore` file, but also on the command-line (`git add *.c`).

GIT_GLOB_PATHSPECS and **GIT_NOGLOB_PATHSPECS** control the default behavior of wildcards in

pathspecs. If `GIT_GLOB_PATHSPECS` is set to 1, wildcard characters act as wildcards (which is the default); if `GIT_NOGLOB_PATHSPECS` is set to 1, wildcard characters only match themselves, meaning something like `*.c` would only match a file named `“*.c”`, rather than any file whose name ends with `.c`. You can override this in individual cases by starting the pathspec with `:(glob)` or `:(literal)`, as in `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` disables both of the above behaviors; no wildcard characters will work, and the override prefixes are disabled as well.

`GIT_ICASE_PATHSPECS` sets all pathspecs to work in a case-insensitive manner.

Committing

The final creation of a Git commit object is usually done by `git-commit-tree`, which uses these environment variables as its primary source of information, falling back to configuration values only if these aren’t present.

`GIT_AUTHOR_NAME` is the human-readable name in the “author” field.

`GIT_AUTHOR_EMAIL` is the email for the “author” field.

`GIT_AUTHOR_DATE` is the timestamp used for the “author” field.

`GIT_COMMITTER_NAME` sets the human name for the “committer” field.

`GIT_COMMITTER_EMAIL` is the email address for the “committer” field.

`GIT_COMMITTER_DATE` is used for the timestamp in the “committer” field.

`EMAIL` is the fallback email address in case the `user.email` configuration value isn’t set. If *this* isn’t set, Git falls back to the system user and host names.

Networking

Git uses the `curl` library to do network operations over HTTP, so `GIT_CURL_VERBOSE` tells Git to emit all the messages generated by that library. This is similar to doing `curl -v` on the command line.

`GIT_SSL_NO_VERIFY` tells Git not to verify SSL certificates. This can sometimes be necessary if you’re using a self-signed certificate to serve Git repositories over HTTPS, or you’re in the middle of setting up a Git server but haven’t installed a full certificate yet.

If the data rate of an HTTP operation is lower than `GIT_HTTP_LOW_SPEED_LIMIT` bytes per second for longer than `GIT_HTTP_LOW_SPEED_TIME` seconds, Git will abort that operation. These values override the `http.lowSpeedLimit` and `http.lowSpeedTime` configuration values.

`GIT_HTTP_USER_AGENT` sets the user-agent string used by Git when communicating over HTTP. The default is a value like `git/2.0.0`.

Diffing and Merging

`GIT_DIFF_OPTS` is a bit of a misnomer. The only valid values are `-u<n>` or `--unified=<n>`, which controls the number of context lines shown in a `git diff` command.

GIT_EXTERNAL_DIFF is used as an override for the `diff.external` configuration value. If it's set, Git will invoke this program when `git diff` is invoked.

GIT_DIFF_PATH_COUNTER and **GIT_DIFF_PATH_TOTAL** are useful from inside the program specified by **GIT_EXTERNAL_DIFF** or `diff.external`. The former represents which file in a series is being diffed (starting with 1), and the latter is the total number of files in the batch.

GIT_MERGE_VERBOSITY controls the output for the recursive merge strategy. The allowed values are as follows:

- 0 outputs nothing, except possibly a single error message.
- 1 shows only conflicts.
- 2 also shows file changes.
- 3 shows when files are skipped because they haven't changed.
- 4 shows all paths as they are processed.
- 5 and above show detailed debugging information.

The default value is 2.

Debugging

Want to *really* know what Git is up to? Git has a fairly complete set of traces embedded, and all you need to do is turn them on. The possible values of these variables are as follows:

- “true” , “1” , or “2” – the trace category is written to stderr.
- An absolute path starting with `/` – the trace output will be written to that file.

GIT_TRACE controls general traces, which don't fit into any specific category. This includes the expansion of aliases, and delegation to other sub-programs.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554          trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341    trace: run_command: 'git-lga'
20:12:49.879529 git.c:282          trace: alias expansion: lga =>
'log' '--graph' '--pretty=oneline' '--abbrev-commit' '--decorate' '--
all'
20:12:49.879885 git.c:349          trace: built-in: git 'log' '--
graph' '--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341    trace: run_command: 'less'
20:12:49.899675 run-command.c:192    trace: exec: 'less'
```

GIT_TRACE_PACK_ACCESS controls tracing of packfile access. The first field is the packfile being accessed, the second is the offset within that file:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088          .git/objects/pack/pack-
c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088          .git/objects/pack/pack-
c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088          .git/objects/pack/pack-
c3fa...291e.pack 35175
# [...]
20:10:12.087398 sha1_file.c:2088          .git/objects/pack/pack-
e80e...e3d2.pack 56914983
20:10:12.087419 sha1_file.c:2088          .git/objects/pack/pack-
e80e...e3d2.pack 14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

GIT_TRACE_PACKET enables packet-level tracing for network operations.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46          packet:      git< #
service=git-upload-pack
20:15:14.867071 pkt-line.c:46          packet:      git< 0000
20:15:14.867079 pkt-line.c:46          packet:      git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-
band side-band-64k ofs-delta shallow no-progress include-tag
multi_ack_detailed no-done symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46          packet:      git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-
show-diff-func-name
20:15:14.867094 pkt-line.c:46          packet:      git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# [...]
```

GIT_TRACE_PERFORMANCE controls logging of performance data. The output shows how long each particular git invocation takes.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414           performance: 0.374835000 s: git
command: 'git' 'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414           performance: 0.343020000 s: git
command: 'git' 'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414           performance: 3.715349000 s: git
command: 'git' 'pack-objects' '--keep-true-parents' '--honor-pack-keep'
'--non-empty' '--all' '--reflog' '--unpack-unreachable=2.weeks.ago' '--local'
'--delta-base-offset' '.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414           performance: 0.000910000 s: git
command: 'git' 'prune-packed'
20:18:23.605218 trace.c:414           performance: 0.017972000 s: git
command: 'git' 'update-server-info'
20:18:23.606342 trace.c:414           performance: 3.756312000 s: git
command: 'git' 'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414           performance: 1.616423000 s: git
command: 'git' 'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414           performance: 0.001051000 s: git
command: 'git' 'rerere' 'gc'
20:18:25.233159 trace.c:414           performance: 6.112217000 s: git
command: 'git' 'gc'
```

GIT_TRACE_SETUP shows information about what Git is discovering about the repository and environment it's interacting with.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315           setup: git_dir: .git
20:19:47.087184 trace.c:316           setup: worktree:
/Users/ben/src/git
20:19:47.087191 trace.c:317           setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318           setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Miscellaneous

GIT_SSH, if specified, is a program that is invoked instead of `ssh` when Git tries to connect to an SSH host. It is invoked like `$GIT_SSH [username@]host [-p <port>] <command>`. Note that this isn't the easiest way to customize how `ssh` is invoked; it won't support extra command-line parameters, so you'd have to write a wrapper script and set `GIT_SSH` to point to it. It's probably easier just to use the `~/.ssh/config` file for that.

GIT_ASKPASS is an override for the `core.askpass` configuration value. This is the program invoked whenever Git needs to ask the user for credentials, which can expect a text prompt as a command-line argument, and should return the answer on `stdout`. (See [Credential Storage](#) for more on this subsystem.)

GIT_NAMESPACE controls access to namespaced refs, and is equivalent to the `--namespace` flag. This is mostly useful on the server side, where you may want to store multiple forks of a single repository in one repository, only keeping the refs separate.

GIT_FLUSH can be used to force Git to use non-buffered I/O when writing incrementally to `stdout`. A value of 1 causes Git to flush more often, a value of 0 causes all output to be buffered. The default value (if this variable is not set) is to choose an appropriate buffering scheme depending on the activity and the output mode.

GIT_REFLOG_ACTION lets you specify the descriptive text written to the reflog. Here's an example:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'  
[master 9e3d55a] my message  
$ git reflog -1  
9e3d55a HEAD@{0}: my action: my message
```

Summary

You should have a pretty good understanding of what Git does in the background and, to some degree, how it's implemented. This chapter has covered a number of plumbing commands – commands that are lower level and simpler than the porcelain commands you've learned about in the rest of the book. Understanding how Git works at a lower level should make it easier to understand why it's doing what it's doing and also to write your own tools and helping scripts to make your specific workflow work for you.

Git as a content-addressable filesystem is a very powerful tool that you can easily use as more than just a VCS. We hope you can use your newfound knowledge of Git internals to implement your own cool application of this technology and feel more comfortable using Git in more advanced ways.

附錄 A: Git in Other Environments

If you read through the whole book, you’ve learned a lot about how to use Git at the command line. You can work with local files, connect your repository to others over a network, and work effectively with others. But the story doesn’t end there; Git is usually used as part of a larger ecosystem, and the terminal isn’t always the best way to work with it. Now we’ll take a look at some of the other kinds of environments where Git can be useful, and how other applications (including yours) work alongside Git.

Graphical Interfaces

Git’s native environment is in the terminal. New features show up there first, and only at the command line is the full power of Git completely at your disposal. But plain text isn’t the best choice for all tasks; sometimes a visual representation is what you need, and some users are much more comfortable with a point-and-click interface.

It’s important to note that different interfaces are tailored for different workflows. Some clients expose only a carefully curated subset of Git functionality, in order to support a specific way of working that the author considers effective. When viewed in this light, none of these tools can be called “better” than any of the others, they’re simply more fit for their intended purpose. Also note that there’s nothing these graphical clients can do that the command-line client can’t; the command-line is still where you’ll have the most power and control when working with your repositories.

gitk and git-gui

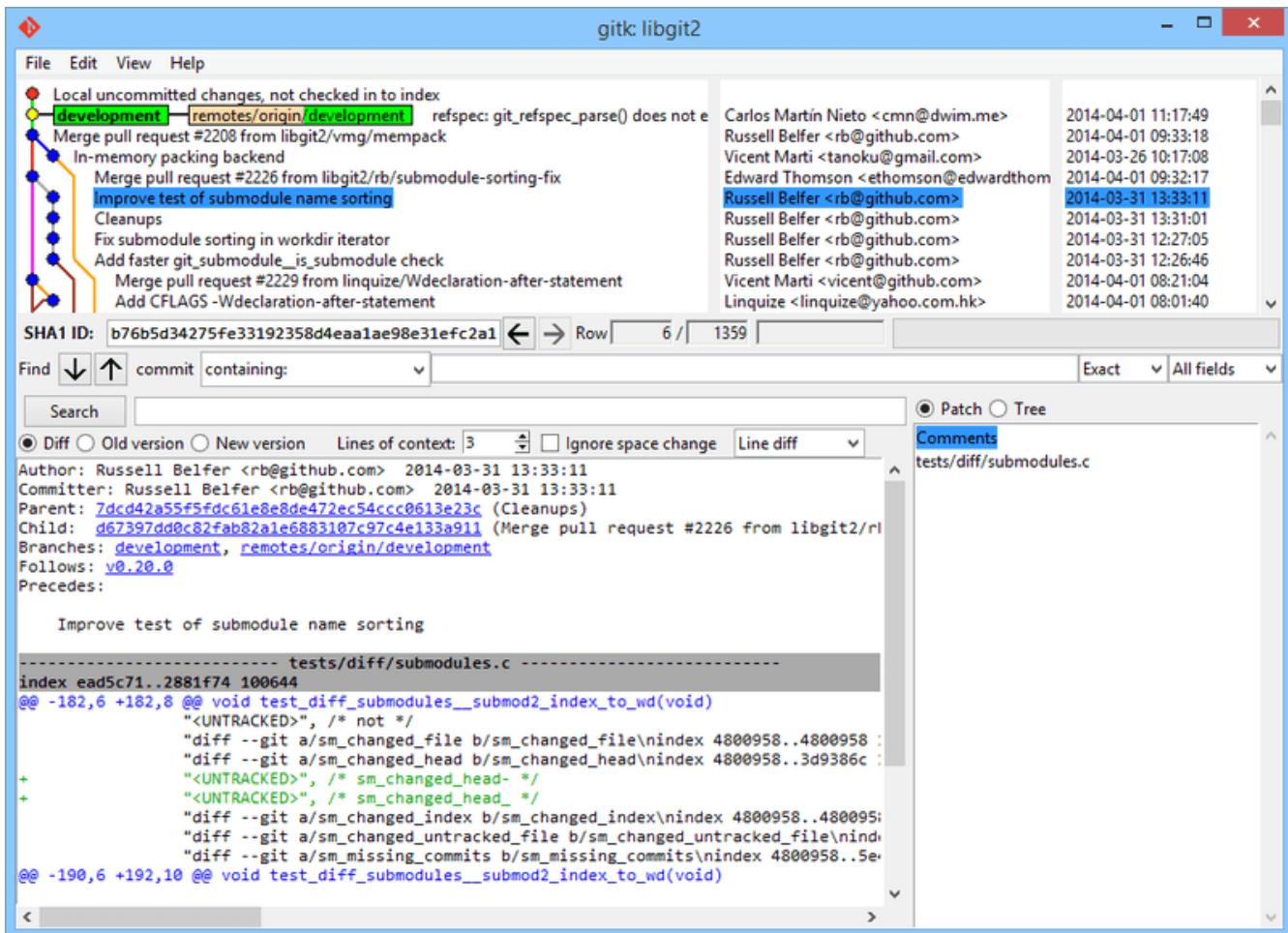
When you install Git, you also get its visual tools, `gitk` and `git-gui`.

`gitk` is a graphical history viewer. Think of it like a powerful GUI shell over `git log` and `git grep`. This is the tool to use when you’re trying to find something that happened in the past, or visualize your project’s history.

Gitk is easiest to invoke from the command-line. Just `cd` into a Git repository, and type:

```
$ gitk [git log options]
```

Gitk accepts many command-line options, most of which are passed through to the underlying `git log` action. Probably one of the most useful is the `--all` flag, which tells gitk to show commits reachable from *any* ref, not just HEAD. Gitk’s interface looks like this:



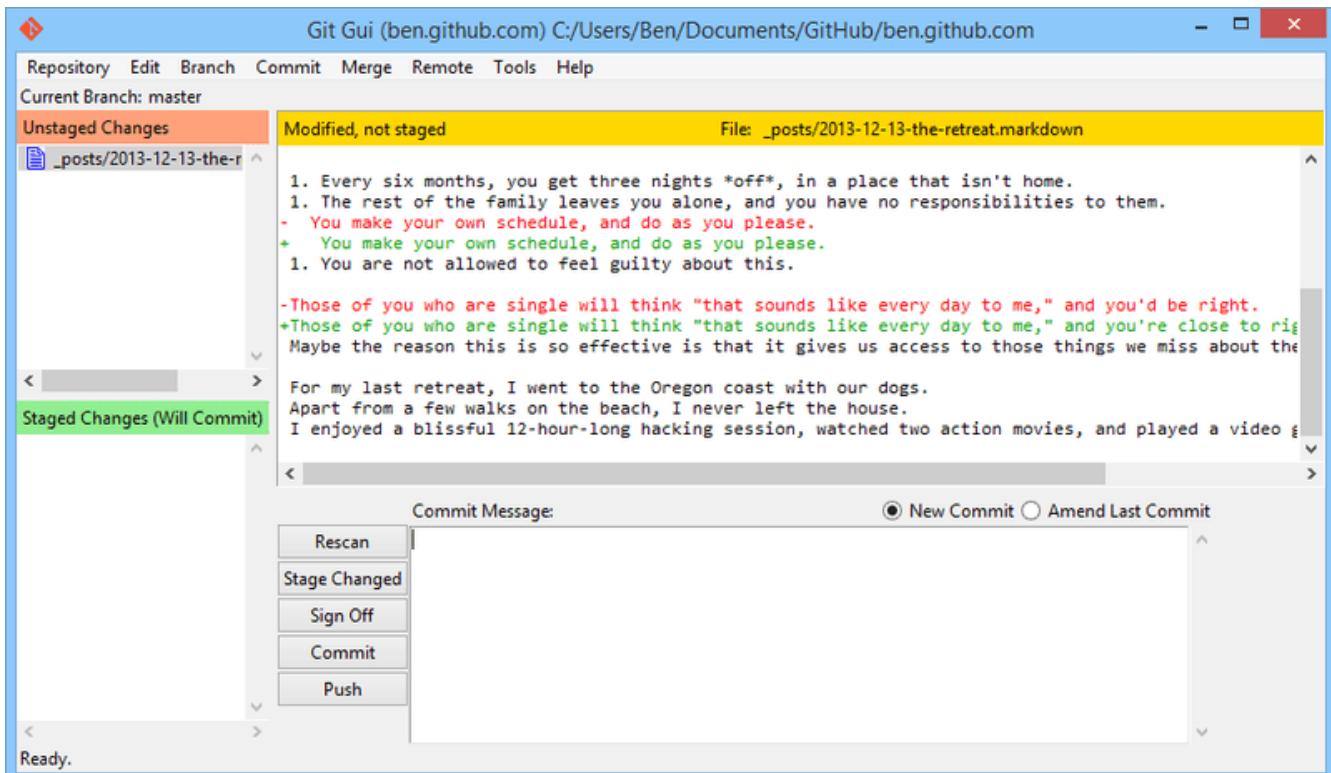
圖表 152. The `gitk` history viewer.

On the top is something that looks a bit like the output of `git log --graph`; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

`git-gui`, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

```
$ git gui
```

And it looks something like this:



圖表 153. The `git-gui` commit tool.

On the left is the index; unstaged changes are on top, staged changes on the bottom. You can move entire files between the two states by clicking on their icons, or you can select a file for viewing by clicking on its name.

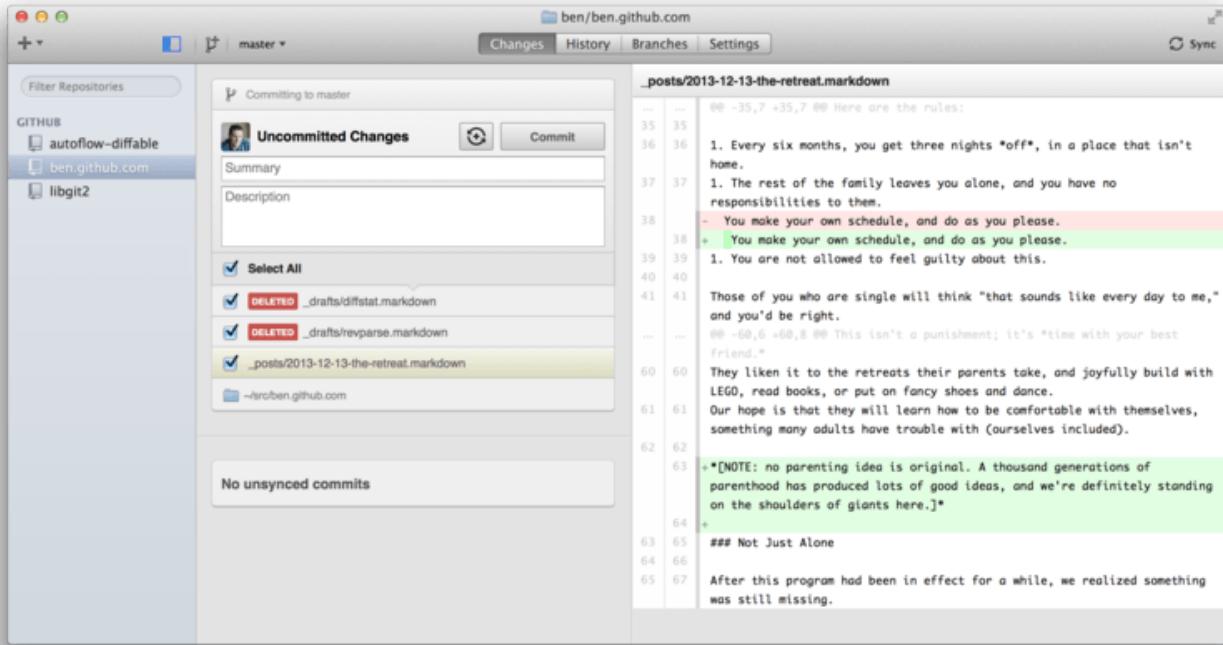
At top right is the diff view, which shows the changes for the currently-selected file. You can stage individual hunks (or individual lines) by right-clicking in this area.

At the bottom right is the message and action area. Type your message into the text box and click “Commit” to do something similar to `git commit`. You can also choose to amend the last commit by choosing the “Amend” radio button, which will update the “Staged Changes” area with the contents of the last commit. Then you can simply stage or unstage some changes, alter the commit message, and click “Commit” again to replace the old commit with a new one.

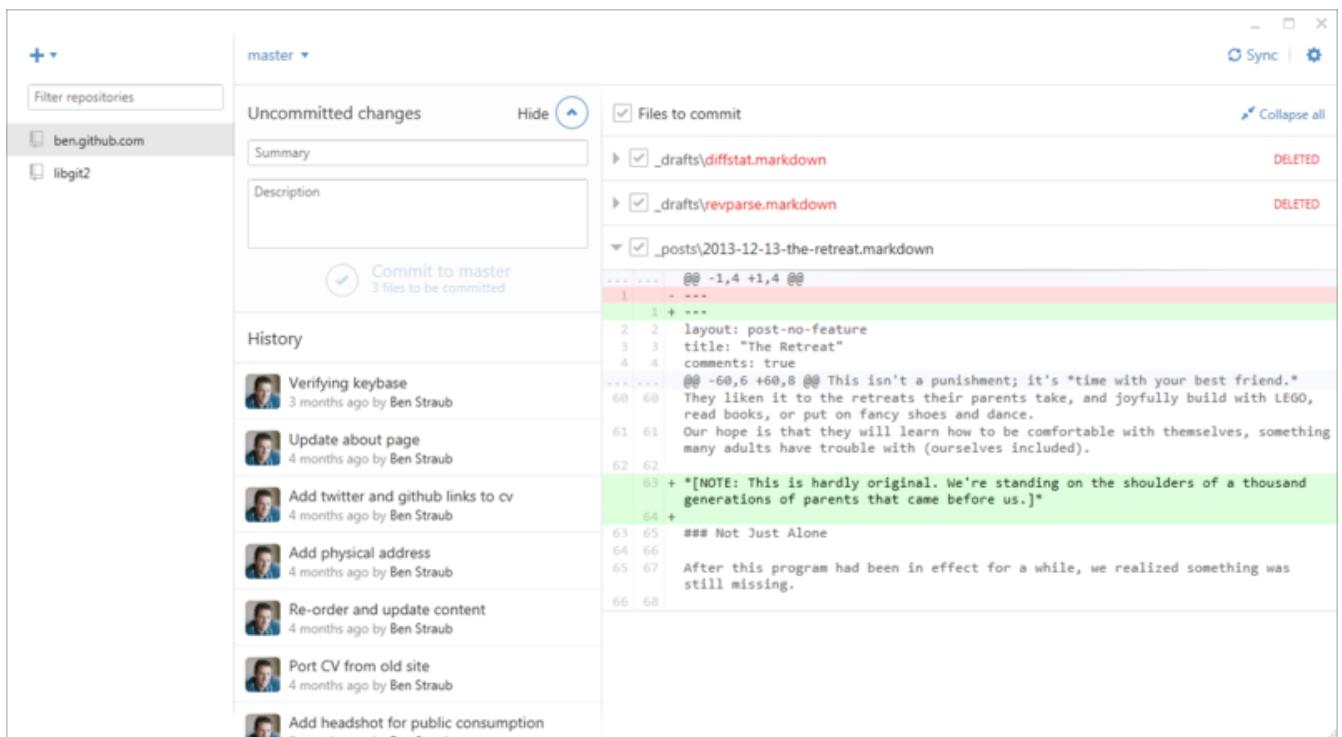
`gitk` and `git-gui` are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

GitHub for Mac and Windows

GitHub has created two workflow-oriented Git clients: one for Windows, and one for Mac. These clients are a good example of workflow-oriented tools – rather than expose *all* of Git’s functionality, they instead focus on a curated set of commonly-used features that work well together. They look like this:



圖表 154. GitHub for Mac.



圖表 155. GitHub for Windows.

They are designed to look and work very much alike, so we'll treat them like a single product in this chapter. We won't be doing a detailed rundown of these tools (they have their own documentation), but a quick tour of the "changes" view (which is where you'll spend most of your time) is in order.

- On the left is the list of repositories the client is tracking; you can add a repository (either by cloning or attaching locally) by clicking the "+" icon at the top of this area.
- In the center is a commit-input area, which lets you input a commit message, and select which files should be included. (On Windows, the commit history is displayed directly below this; on Mac, it's on a separate tab.)

- On the right is a diff view, which shows what’s changed in your working directory, or which changes were included in the selected commit.
- The last thing to notice is the “Sync” button at the top-right, which is the primary way you interact over the network.

筆記 You don’t need a GitHub account to use these tools. While they’re designed to highlight GitHub’s service and recommended workflow, they will happily work with any repository, and do network operations with any Git host.

Installation

GitHub for Windows can be downloaded from <https://windows.github.com>, and GitHub for Mac from <https://mac.github.com>. When the applications are first run, they walk you through all the first-time Git setup, such as configuring your name and email address, and both set up sane defaults for many common configuration options, such as credential caches and CRLF behavior.

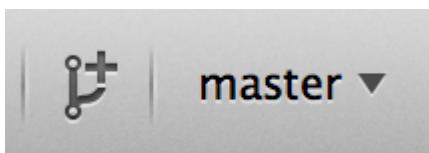
Both are “evergreen” – updates are downloaded and installed in the background while the applications are open. This helpfully includes a bundled version of Git, which means you probably won’t have to worry about manually updating it again. On Windows, the client includes a shortcut to launch Powershell with Posh-git, which we’ll talk more about later in this chapter.

The next step is to give the tool some repositories to work with. The client shows you a list of the repositories you have access to on GitHub, and can clone them in one step. If you already have a local repository, just drag its directory from the Finder or Windows Explorer into the GitHub client window, and it will be included in the list of repositories on the left.

Recommended Workflow

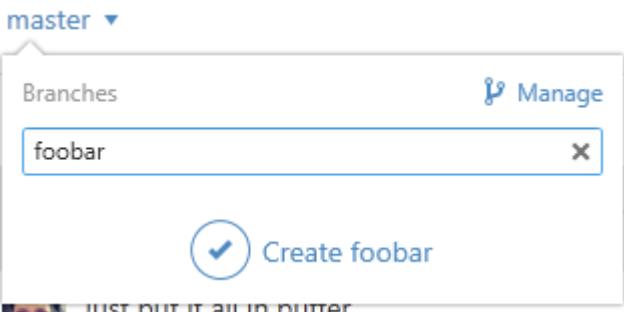
Once it’s installed and configured, you can use the GitHub client for many common Git tasks. The intended workflow for this tool is sometimes called the “GitHub Flow.” We cover this in more detail in [GitHub 流程](#), but the general gist is that (a) you’ll be committing to a branch, and (b) you’ll be syncing up with a remote repository fairly regularly.

Branch management is one of the areas where the two tools diverge. On Mac, there’s a button at the top of the window for creating a new branch:



圖表 156. “Create Branch” button on Mac.

On Windows, this is done by typing the new branch’s name in the branch-switching widget:



圖表 157. Creating a branch on Windows.

Once your branch is created, making new commits is fairly straightforward. Make some changes in your working directory, and when you switch to the GitHub client window, it will show you which files changed. Enter a commit message, select the files you'd like to include, and click the “Commit” button (ctrl-enter or ⌘-enter).

The main way you interact with other repositories over the network is through the “Sync” feature. Git internally has separate operations for pushing, fetching, merging, and rebasing, but the GitHub clients collapse all of these into one multi-step feature. Here’s what happens when you click the Sync button:

1. `git pull --rebase`. If this fails because of a merge conflict, fall back to `git pull --no-rebase`.
2. `git push`.

This is the most common sequence of network commands when working in this style, so squashing them into one command saves a lot of time.

Summary

These tools are very well-suited for the workflow they’re designed for. Developers and non-developers alike can be collaborating on a project within minutes, and many of the best practices for this kind of workflow are baked into the tools. However, if your workflow is different, or you want more control over how and when network operations are done, we recommend you use another client or the command line.

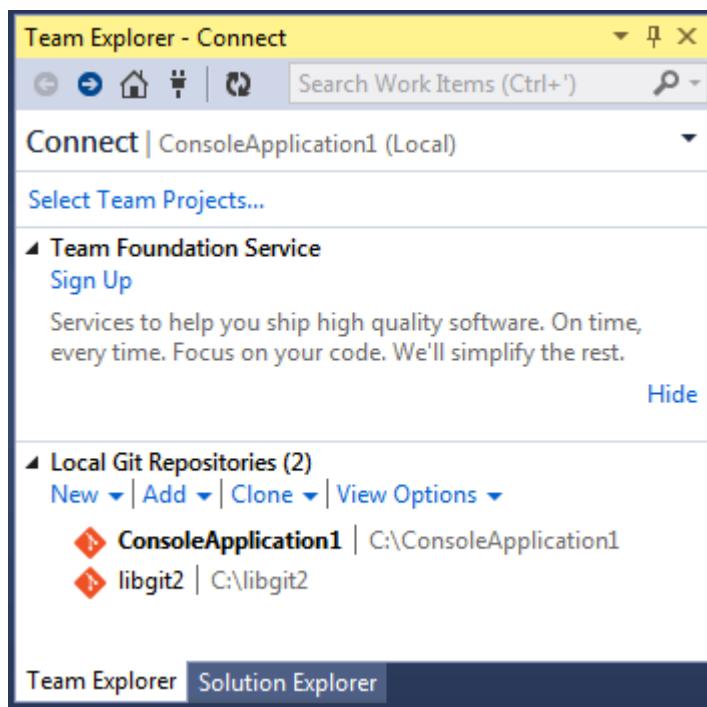
Other GUIs

There are a number of other graphical Git clients, and they run the gamut from specialized, single-purpose tools all the way to apps that try to expose everything Git can do. The official Git website has a curated list of the most popular clients at <http://git-scm.com/downloads/guis>. A more comprehensive list is available on the Git wiki site, at https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git in Visual Studio

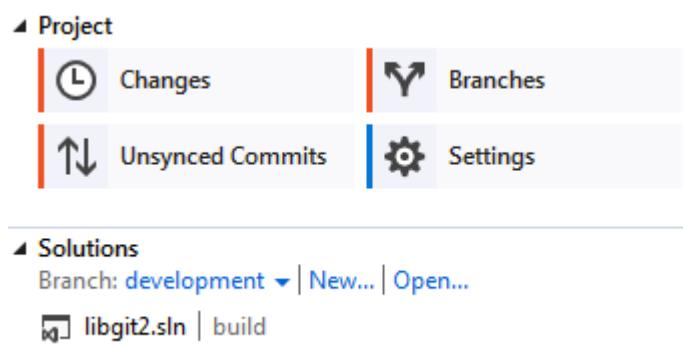
Starting with Visual Studio 2013 Update 1, Visual Studio users have a Git client built directly into their IDE. Visual Studio has had source-control integration features for quite some time, but they were oriented towards centralized, file-locking systems, and Git was not a good match for this workflow. Visual Studio 2013’s Git support has been separated from this older feature, and the result is a much better fit between Studio and Git.

To locate the feature, open a project that's controlled by Git (or just `git init` an existing project), and select View > Team Explorer from the menu. You'll see the "Connect" view, which looks a bit like this:



圖表 158. Connecting to a Git repository from Team Explorer.

Visual Studio remembers all of the projects you've opened that are Git-controlled, and they're available in the list at the bottom. If you don't see the one you want there, click the "Add" link and type in the path to the working directory. Double clicking on one of the local Git repositories leads you to the Home view, which looks like [The "Home" view for a Git repository in Visual Studio..](#) This is a hub for performing Git actions; when you're writing code, you'll probably spend most of your time in the "Changes" view, but when it comes time to pull down changes made by your teammates, you'll use the "Unsynced Commits" and "Branches" views.



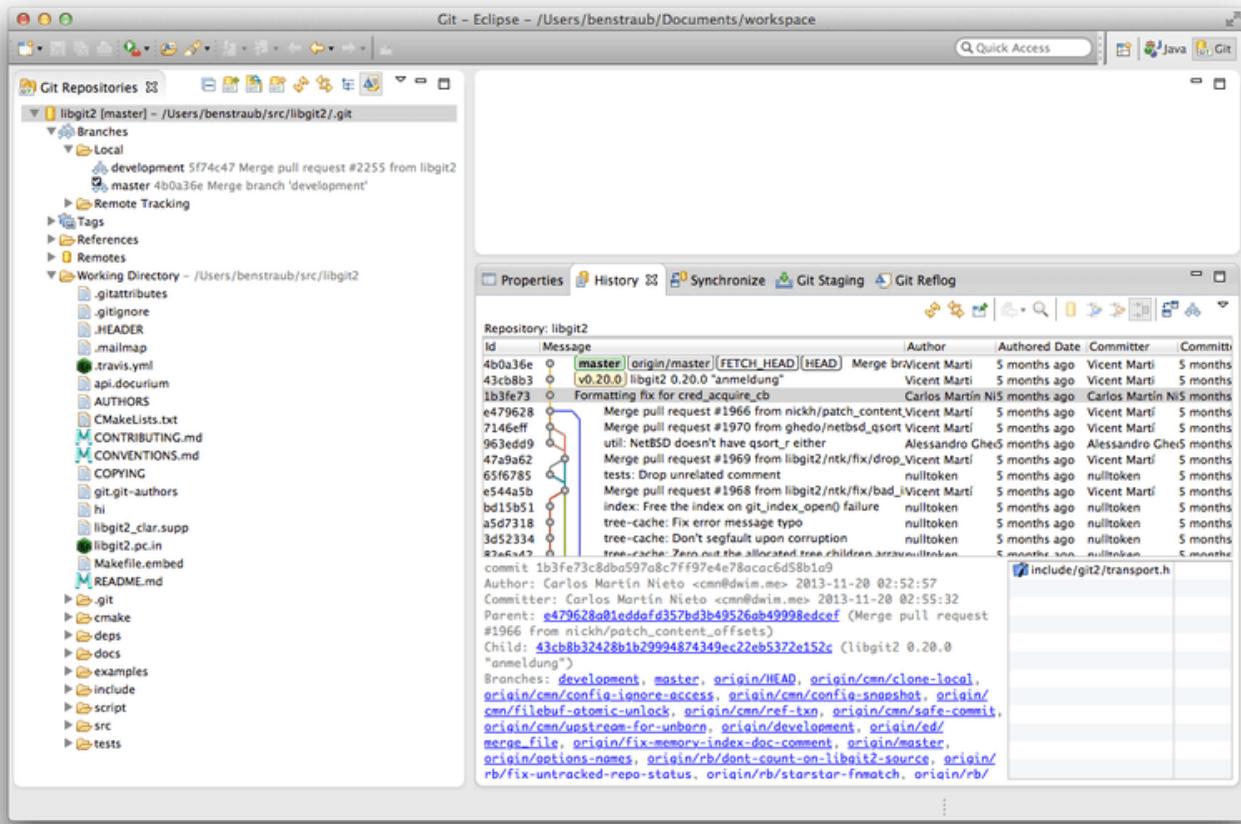
圖表 159. The "Home" view for a Git repository in Visual Studio.

Visual Studio now has a powerful task-focused UI for Git. It includes a linear history view, a diff viewer, remote commands, and many other capabilities. For complete documentation of this feature (which doesn't fit here), go to <http://msdn.microsoft.com/en-us/library/hh850437.aspx>.

Git in Eclipse

Eclipse ships with a plugin called Egit, which provides a fairly-complete interface to Git operations.

It's accessed by switching to the Git Perspective (Window > Open Perspective > Other...), and select "Git").



圖表 160. Eclipse's EGit environment.

EGit comes with plenty of great documentation, which you can find by going to Help > Help Contents, and choosing the "EGit Documentation" node from the contents listing.

Git in Bash

If you're a Bash user, you can tap into some of your shell's features to make your experience with Git a lot friendlier. Git actually ships with plugins for several shells, but it's not turned on by default.

First, you need to get a copy of the `contrib/completion/git-completion.bash` file out of the Git source code. Copy it somewhere handy, like your home directory, and add this to your `.bashrc`:

```
. ~/git-completion.bash
```

Once that's done, change your directory to a git repository, and type:

```
$ git chec<tab>
```

...and Bash will auto-complete to `git checkout`. This works with all of Git's subcommands, command-line parameters, and remotes and ref names where appropriate.

It's also useful to customize your prompt to show information about the current directory's Git

repository. This can be as simple or complex as you want, but there are generally a few key pieces of information that most people want, like the current branch, and the status of the working directory. To add these to your prompt, just copy the `contrib/completion/git-prompt.sh` file from Git's source repository to your home directory, add something like this to your `.bashrc`:

```
. ~/git-prompt.sh  
export GIT_PS1_SHOWDIRTYSTATE=1  
export PS1=' \w$(_git_ps1 " (%s)")\$ '
```

The `\w` means print the current working directory, the `\$` prints the `$` part of the prompt, and `_git_ps1 " (%s)"` calls the function provided by `git-prompt.sh` with a formatting argument. Now your bash prompt will look like this when you're anywhere inside a Git-controlled project:



圖表 161. Customized `bash` prompt.

Both of these scripts come with helpful documentation; take a look at the contents of `git-completion.bash` and `git-prompt.sh` for more information.

Git in Zsh

Zsh also ships with a tab-completion library for Git. To use it, simply run `autoload -Uz compinit && compinit` in your `.zshrc`. Zsh's interface is a bit more powerful than Bash's:

```
$ git che<tab>  
check-attr      -- display gitattributes information  
check-ref-format -- ensure that a reference name is well formed  
checkout        -- checkout branch or paths to working tree  
checkout-index   -- copy files from index to working directory  
cherry          -- find commits not merged upstream  
cherry-pick     -- apply changes introduced by some existing commits
```

Ambiguous tab-completions aren't just listed; they have helpful descriptions, and you can graphically navigate the list by repeatedly hitting tab. This works with Git commands, their arguments, and names of things inside the repository (like refs and remotes), as well as filenames and all the other things Zsh knows how to tab-complete.

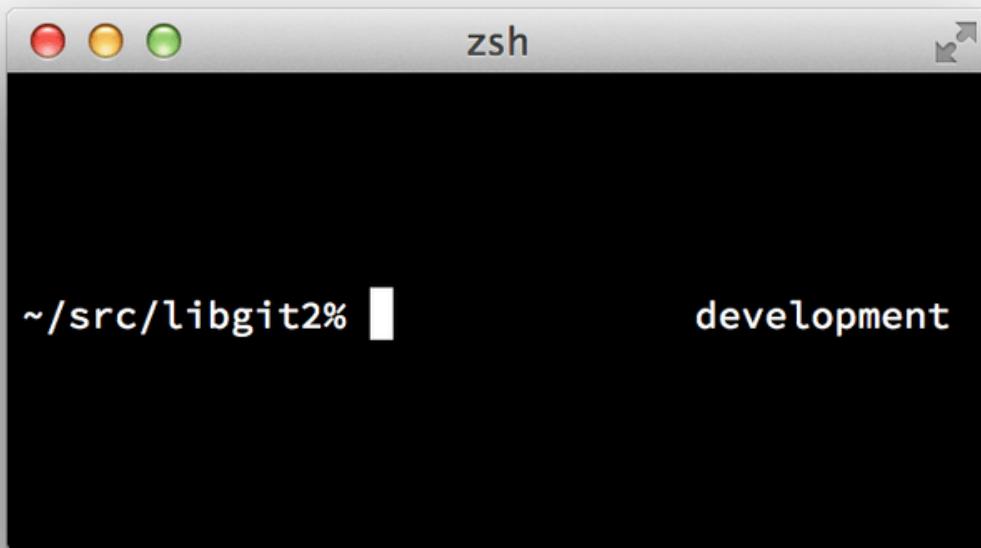
Zsh ships with a framework for getting information from version control systems, called `vcs_info`. To include the branch name in the prompt on the right side, add these lines to your `~/.zshrc` file:

```

autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_%#
zstyle ':vcs_info:git:*' formats '%b'

```

This results in a display of the current branch on the right-hand side of the terminal window, whenever your shell is inside a Git repository. (The left side is supported as well, of course; just uncomment the assignment to PROMPT.) It looks a bit like this:

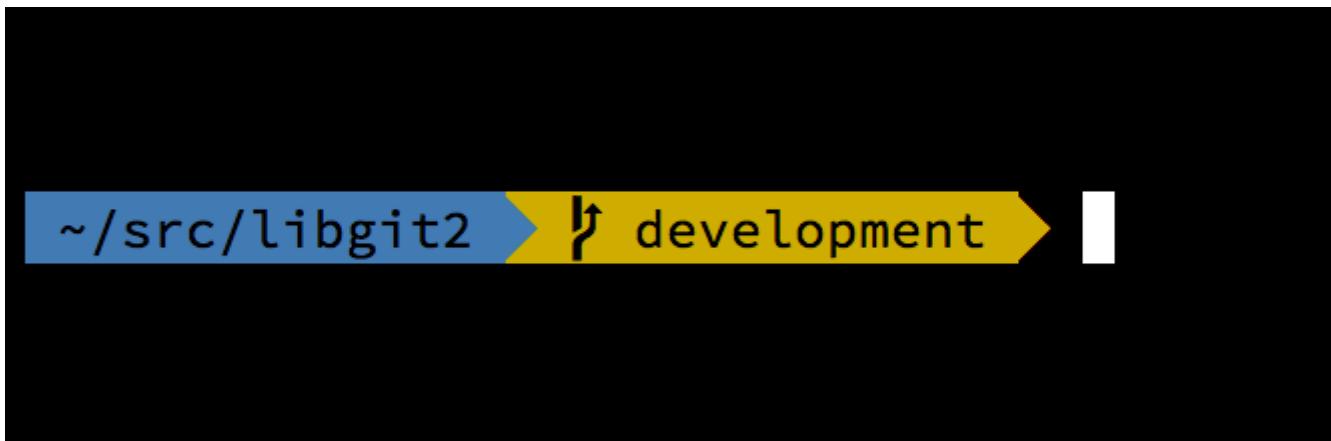


圖表 162. Customized zsh prompt.

For more information on vcs_info, check out its documentation in the [zshcontrib\(1\)](#) manual page, or online at <http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information>.

Instead of vcs_info, you might prefer the prompt customization script that ships with Git, called [git-prompt.sh](#); see <http://git-prompt.sh> for details. [git-prompt.sh](#) is compatible with both Bash and Zsh.

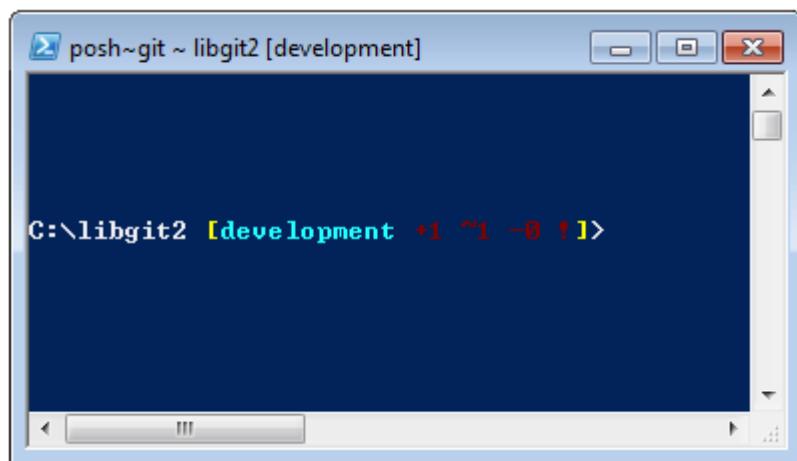
Zsh is powerful enough that there are entire frameworks dedicated to making it better. One of them is called "oh-my-zsh", and it can be found at <https://github.com/robbyrussell/oh-my-zsh>. oh-my-zsh's plugin system comes with powerful git tab-completion, and it has a variety of prompt "themes", many of which display version-control data. [An example of an oh-my-zsh theme](#). is just one example of what can be done with this system.



圖表 163. An example of an oh-my-zsh theme.

Git in Powershell

The standard command-line terminal on Windows (`cmd.exe`) isn't really capable of a customized Git experience, but if you're using Powershell, you're in luck. A package called Posh-Git (<https://github.com/dahlbyk/posh-git>) provides powerful tab-completion facilities, as well as an enhanced prompt to help you stay on top of your repository status. It looks like this:



圖表 164. Powershell with Posh-git.

If you've installed GitHub for Windows, Posh-Git is included by default, and all you have to do is add these lines to your `profile.ps1` (which is usually located in `C:\Users\<username>\Documents\WindowsPowerShell`):

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

If you're not a GitHub for Windows user, just download a Posh-Git release from (<https://github.com/dahlbyk/posh-git>), and uncompress it to the `WindowsPowerShell` directory. Then open a Powershell prompt as the administrator, and do this:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm  
> cd ~\Documents\WindowsPowerShell\posh-git  
> .\install.ps1
```

This will add the proper line to your `profile.ps1` file, and posh-git will be active the next time you open your prompt.

Summary

You've learned how to harness Git's power from inside the tools that you use during your everyday work, and also how to access Git repositories from your own programs.

附錄 B: Embedding Git in your Applications

If your application is for developers, chances are good that it could benefit from integration with source control. Even non-developer applications, such as document editors, could potentially benefit from version-control features, and Git's model works very well for many different scenarios.

If you need to integrate Git with your application, you have essentially three choices: spawning a shell and using the Git command-line tool; Libgit2; and JGit.

Command-line Git

One option is to spawn a shell process and use the Git command-line tool to do the work. This has the benefit of being canonical, and all of Git's features are supported. This also happens to be fairly easy, as most runtime environments have a relatively simple facility for invoking a process with command-line arguments. However, this approach does have some downsides.

One is that all the output is in plain text. This means that you'll have to parse Git's occasionally-changing output format to read progress and result information, which can be inefficient and error-prone.

Another is the lack of error recovery. If a repository is corrupted somehow, or the user has a malformed configuration value, Git will simply refuse to perform many operations.

Yet another is process management. Git requires you to maintain a shell environment on a separate process, which can add unwanted complexity. Trying to coordinate many of these processes (especially when potentially accessing the same repository from several processes) can be quite a challenge.

Libgit2

Another option at your disposal is to use Libgit2. Libgit2 is a dependency-free implementation of Git, with a focus on having a nice API for use within other programs. You can find it at <http://libgit2.github.com>.

First, let's take a look at what the C API looks like. Here's a whirlwind tour:

```

// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);

```

The first couple of lines open a Git repository. The `git_repository` type represents a handle to a repository with a cache in memory. This is the simplest method, for when you know the exact path to a repository’s working directory or `.git` folder. There’s also the `git_repository_open_ext` which includes options for searching, `git_clone` and friends for making a local clone of a remote repository, and `git_repository_init` for creating an entirely new repository.

The second chunk of code uses rev-parse syntax (see [Branch References](#) for more on this) to get the commit that HEAD eventually points to. The type returned is a `git_object` pointer, which represents something that exists in the Git object database for a repository. `git_object` is actually a “parent” type for several different kinds of objects; the memory layout for each of the “child” types is the same as for `git_object`, so you can safely cast to the right one. In this case, `git_object_type(commit)` would return `GIT_OBJ_COMMIT`, so it’s safe to cast to a `git_commit` pointer.

The next chunk shows how to access the commit’s properties. The last line here uses a `git_oid` type; this is Libgit2’s representation for a SHA-1 hash.

From this sample, a couple of patterns have started to emerge:

- If you declare a pointer and pass a reference to it into a Libgit2 call, that call will probably return an integer error code. A `0` value indicates success; anything less is an error.
- If Libgit2 populates a pointer for you, you’re responsible for freeing it.
- If Libgit2 returns a `const` pointer from a call, you don’t have to free it, but it will become invalid when the object it belongs to is freed.
- Writing C is a bit painful.

That last one means it isn’t very probable that you’ll be writing C when using Libgit2. Fortunately, there are a number of language-specific bindings available that make it fairly easy to work with Git repositories from your specific language and environment. Let’s take a look at the above example written using the Ruby bindings for Libgit2, which are named Rugged, and can be found at <https://github.com/libgit2/rugged>.

```

repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree

```

As you can see, the code is much less cluttered. Firstly, Rugged uses exceptions; it can raise things like [ConfigError](#) or [ObjectError](#) to signal error conditions. Secondly, there's no explicit freeing of resources, since Ruby is garbage-collected. Let's take a look at a slightly more complicated example: crafting a commit from scratch

```

blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
  :email => "bob@example.com",
  :name => "Bob User",
  :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
  :tree => index.write_tree(repo), ③
  :author => sig,
  :committer => sig, ④
  :message => "Add newfile.txt", ⑤
  :parents => repo.empty? ? [] : [repo.head.target].compact, ⑥
  :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧

```

- ① Create a new blob, which contains the contents of a new file.
- ② Populate the index with the head commit's tree, and add the new file at the path `newfile.txt`.
- ③ This creates a new tree in the ODB, and uses it for the new commit.
- ④ We use the same signature for both the author and committer fields.
- ⑤ The commit message.
- ⑥ When creating a commit, you have to specify the new commit's parents. This uses the tip of HEAD for the single parent.
- ⑦ Rugged (and Libgit2) can optionally update a reference when making a commit.
- ⑧ The return value is the SHA-1 hash of a new commit object, which you can then use to get a [Commit](#) object.

The Ruby code is nice and clean, but since Libgit2 is doing the heavy lifting, this code will run pretty fast, too. If you're not a rubyist, we touch on some other bindings in [Other Bindings](#).

Advanced Functionality

Libgit2 has a couple of capabilities that are outside the scope of core Git. One example is pluggability: Libgit2 allows you to provide custom “backends” for several types of operation, so you can store things in a different way than stock Git does. Libgit2 allows custom backends for configuration, ref storage, and the object database, among other things.

Let’s take a look at how this works. The code below is borrowed from the set of backend examples provided by the Libgit2 team (which can be found at <https://github.com/libgit2/libgit2-backends>). Here’s how a custom backend for the object database is set up:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*...*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(repo); ④
```

(Note that errors are captured, but not handled. We hope your code is better than ours.)

- ① Initialize an empty object database (ODB) “frontend,” which will act as a container for the “backends” which are the ones doing the real work.
- ② Initialize a custom ODB backend.
- ③ Add the backend to the frontend.
- ④ Open a repository, and set it to use our ODB to look up objects.

But what is this `git_odb_backend_mine` thing? Well, that’s the constructor for your own ODB implementation, and you can do whatever you want in there, so long as you fill in the `git_odb_backend` structure properly. Here’s what it *could* look like:

```

typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*...*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = ...;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // ...

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}

```

The subtlest constraint here is that `my_backend_struct`'s first member must be a `git_odb_backend` structure; this ensures that the memory layout is what the Libgit2 code expects it to be. The rest of it is arbitrary; this structure can be as large or small as you need it to be.

The initialization function allocates some memory for the structure, sets up the custom context, and then fills in the members of the `parent` structure that it supports. Take a look at the `include/git2/sys/odb_backend.h` file in the Libgit2 source for a complete set of call signatures; your particular use case will help determine which of these you'll want to support.

Other Bindings

Libgit2 has bindings for many languages. Here we show a small example using a few of the more complete bindings packages as of this writing; libraries exist for many other languages, including C++, Go, Node.js, Erlang, and the JVM, all in various stages of maturity. The official collection of bindings can be found by browsing the repositories at <https://github.com/libgit2>. The code we'll write will return the commit message from the commit eventually pointed to by HEAD (sort of like `git log -1`).

LibGit2Sharp

If you're writing a .NET or Mono application, LibGit2Sharp (<https://github.com/libgit2/libgit2sharp>) is what you're looking for. The bindings are written in C#, and great care has been taken to wrap the raw Libgit2 calls with native-feeling CLR APIs. Here's what our example program looks like:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

For desktop Windows applications, there's even a NuGet package that will help you get started quickly.

objective-git

If your application is running on an Apple platform, you're likely using Objective-C as your implementation language. Objective-Git (<https://github.com/libgit2/objective-git>) is the name of the Libgit2 bindings for that environment. The example program looks like this:

```
GTRepository *repo =  
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath:  
        @"/path/to/repo"] error:NULL];  
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget]  
    message];
```

Objective-git is fully interoperable with Swift, so don't fear if you've left Objective-C behind.

pygit2

The bindings for Libgit2 in Python are called Pygit2, and can be found at <http://www.pygit2.org/>. Our example program:

```
pygit2.Repository("/path/to/repo") # open repository  
    .head                         # get the current branch  
    .peel(pygit2.Commit)           # walk down to the commit  
    .message                       # read the message
```

Further Reading

Of course, a full treatment of Libgit2's capabilities is outside the scope of this book. If you want more information on Libgit2 itself, there's API documentation at <https://libgit2.github.com/libgit2>, and a set of guides at <https://libgit2.github.com/docs>. For the other bindings, check the bundled README and tests; there are often small tutorials and pointers to further reading there.

JGit

If you want to use Git from within a Java program, there is a fully featured Git library called JGit. JGit is a relatively full-featured implementation of Git written natively in Java, and is widely used in the Java community. The JGit project is under the Eclipse umbrella, and its home can be found at <http://www.eclipse.org/jgit>.

Getting Set Up

There are a number of ways to connect your project with JGit and start writing code against it. Probably the easiest is to use Maven – the integration is accomplished by adding the following snippet to the `<dependencies>` tag in your pom.xml file:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

The `version` will most likely have advanced by the time you read this; check <http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit> for updated repository information. Once this step is done, Maven will automatically acquire and use the JGit libraries that you'll need.

If you would rather manage the binary dependencies yourself, pre-built JGit binaries are available from <http://www.eclipse.org/jgit/download>. You can build them into your project by running a command like this:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Plumbing

JGit has two basic levels of API: plumbing and porcelain. The terminology for these comes from Git itself, and JGit is divided into roughly the same kinds of areas: porcelain APIs are a friendly front-end for common user-level actions (the sorts of things a normal user would use the Git command-line tool for), while the plumbing APIs are for interacting with low-level repository objects directly.

The starting point for most JGit sessions is the `Repository` class, and the first thing you'll want to do is create an instance of it. For a filesystem-based repository (yes, JGit allows for other storage models), this is accomplished using `FileRepositoryBuilder`:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

The builder has a fluent API for providing all the things it needs to find a Git repository, whether or not your program knows exactly where it's located. It can use environment variables (`.readEnvironment()`), start from a place in the working directory and search (`.setWorkTree(...).findGitDir()`), or just open a known `.git` directory as above.

Once you have a `Repository` instance, you can do all sorts of things with it. Here's a quick sampling:

```

// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");

```

There's quite a bit going on here, so let's go through it one section at a time.

The first line gets a pointer to the `master` reference. JGit automatically grabs the *actual* master ref, which lives at `refs/heads/master`, and returns an object that lets you fetch information about the reference. You can get the name (`.getName()`), and either the target object of a direct reference (`.getObjectId()`) or the reference pointed to by a symbolic ref (`.getTarget()`). Ref objects are also used to represent tag refs and objects, so you can ask if the tag is “peeled,” meaning that it points to the final target of a (potentially long) string of tag objects.

The second line gets the target of the `master` reference, which is returned as an `ObjectId` instance. `ObjectId` represents the SHA-1 hash of an object, which might or might not exist in Git's object database. The third line is similar, but shows how JGit handles the rev-parse syntax (for more on this, see [Branch References](#)); you can pass any object specifier that Git understands, and JGit will return either a valid `ObjectId` for that object, or `null`.

The next two lines show how to load the raw contents of an object. In this example, we call `ObjectLoader.copyTo()` to stream the contents of the object directly to stdout, but `ObjectLoader` also has methods to read the type and size of an object, as well as return it as a byte array. For large objects (where `.isLarge()` returns `true`), you can call `.openStream()` to get an `InputStream`-like object that can read the raw object data without pulling it all into memory at once.

The next few lines show what it takes to create a new branch. We create a `RefUpdate` instance, configure some parameters, and call `.update()` to trigger the change. Directly following this is the code to delete that same branch. Note that `.setForceUpdate(true)` is required for this to work;

otherwise the `.delete()` call will return `REJECTED`, and nothing will happen.

The last example shows how to fetch the `user.name` value from the Git configuration files. This Config instance uses the repository we opened earlier for local configuration, but will automatically detect the global and system configuration files and read values from them as well.

This is only a small sampling of the full plumbing API; there are many more methods and classes available. Also not shown here is the way JGit handles errors, which is through the use of exceptions. JGit APIs sometimes throw standard Java exceptions (such as `IOException`), but there are a host of JGit-specific exception types that are provided as well (such as `NoRemoteRepositoryException`, `CorruptObjectException`, and `NoMergeBaseException`).

Porcelain

The plumbing APIs are rather complete, but it can be cumbersome to string them together to achieve common goals, like adding a file to the index, or making a new commit. JGit provides a higher-level set of APIs to help out with this, and the entry point to these APIs is the `Git` class:

```
Repository repo;
// construct repo...
Git git = new Git(repo);
```

The `Git` class has a nice set of high-level *builder-style* methods that can be used to construct some pretty complex behavior. Let's take a look at an example – doing something like `git ls-remote`:

```
CredentialsProvider cp = new
UsernamePasswordCredentialsProvider("username", "p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
.setCredentialsProvider(cp)
.setRemote("origin")
.setTags(true)
.setHeads(false)
.call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " +
ref.getObjectId().name());
}
```

This is a common pattern with the `Git` class; the methods return a command object that lets you chain method calls to set parameters, which are executed when you call `.call()`. In this case, we're asking the `origin` remote for tags, but not heads. Also notice the use of a `CredentialsProvider` object for authentication.

Many other commands are available through the `Git` class, including but not limited to `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, and `reset`.

Further Reading

This is only a small sampling of JGit's full capabilities. If you're interested and want to learn more,

here's where to look for information and inspiration:

- The official JGit API documentation is available online at <http://download.eclipse.org/jgit/docs/latest/apidocs>. These are standard Javadoc, so your favorite JVM IDE will be able to install them locally, as well.
- The JGit Cookbook at <https://github.com/centic9/jgit-cookbook> has many examples of how to do specific tasks with JGit.
- There are several good resources pointed out at <http://stackoverflow.com/questions/6861881>.

附錄 C: Git Commands

Throughout the book we have introduced dozens of Git commands and have tried hard to introduce them within something of a narrative, adding more commands to the story slowly. However, this leaves us with examples of usage of the commands somewhat scattered throughout the whole book.

In this appendix, we'll go through all the Git commands we addressed throughout the book, grouped roughly by what they're used for. We'll talk about what each command very generally does and then point out where in the book you can find us having used it.

Setup and Config

There are two commands that are used quite a lot, from the first invocations of Git to common everyday tweaking and referencing, the `config` and `help` commands.

git config

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories.

The `git config` command has been used in nearly every chapter of the book.

In [初次設定 Git](#) we used it to specify our name, email address and editor preference before we even got started using Git.

In [Git Aliases](#) we showed how you could use it to create shorthand commands that expand to long option sequences so you don't have to type them every time.

In [衍合](#) we used it to make `--rebase` the default when you run `git pull`.

In [Credential Storage](#) we used it to set up a default store for your HTTP passwords.

In [Keyword Expansion](#) we showed how to set up smudge and clean filters on content coming in and out of Git.

Finally, basically the entirety of [Git Configuration](#) is dedicated to the command.

git help

The `git help` command is used to show you all the documentation shipped with Git about any command. While we're giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run `git help <command>`.

We introduced the `git help` command in [取得說明文件](#) and showed you how to use it to find more information about the `git shell` in [設定伺服器](#).

Getting and Creating Projects

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

git init

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run `git init`.

We first introduce this in [取得一個 Git 倉儲](#), where we show creating a brand new repository to start working with.

We talk briefly about how you can change the default branch from “master” in [遠端分支](#).

We use this command to create an empty bare repository for a server in [把 Bare Repository 放到伺服器上](#).

Finally, we go through some of the details of what it actually does behind the scenes in [Plumbing and Porcelain](#).

git clone

The `git clone` command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs `git init` to make it an empty Git repository, adds a remote (`git remote add`) to the URL that you pass it (by default named `origin`), runs a `git fetch` from that remote repository and then checks out the latest commit into your working directory with `git checkout`.

The `git clone` command is used in dozens of places throughout the book, but we’ll just list a few interesting places.

It’s basically introduced and explained in [克隆現有的倉儲](#), where we go through a few examples.

In [在伺服器上佈署 Git](#) we look at using the `--bare` option to create a copy of a Git repository with no working directory.

In [Bundling](#) we use it to unbundle a bundled Git repository.

Finally, in [Cloning a Project with Submodules](#) we learn the `--recursive` option to make cloning a repository with submodules a little simpler.

Though it’s used in many other places through the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

Basic Snapshotting

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

git add

The `git add` command adds content from the working directory into the staging area (or “index”) for the next commit. When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We’ll quickly cover some of the unique uses that can be found.

We first introduce and explain `git add` in detail in [追蹤新的檔案](#).

We mention how to use it to resolve merge conflicts in [合併衝突的基本解法](#).

We go over using it to interactively stage only specific parts of a modified file in [Interactive Staging](#).

Finally, we emulate it at a low level in [Tree Objects](#), so you can get an idea of what it’s doing behind the scenes.

git status

The `git status` command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In its normal form, it also will show you some basic hints on how to move files between these stages.

We first cover `status` in [檢查你的檔案狀態](#), both in its basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the `git status` command is covered there.

git diff

The `git diff` command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (`git diff` by itself), between your staging area and your last commit (`git diff --staged`), or between two commits (`git diff master branchB`).

We first look at the basic uses of `git diff` in [檢視已預存及未預存的檔案](#), where we show how to see what changes are staged and which are not yet staged.

We use it to look for possible whitespace issues before committing with the `--check` option in [提交指南](#).

We see how to check the differences between branches more effectively with the `git diff A...B` syntax in [決定要提到哪些資訊](#).

We use it to filter out whitespace differences with `-b` and how to compare different stages of conflicted files with `--theirs`, `--ours` and `--base` in [Advanced Merging](#).

Finally, we use it to effectively compare submodule changes with `--submodule` in [Starting with Submodules](#).

git difftool

The `git difftool` command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in `git diff` command.

We only briefly mention this in [檢視已預存及未預存的檔案](#).

git commit

The `git commit` command takes all the file contents that have been staged with `git add` and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in [提交你的修改](#). There we also demonstrate how to use the `-a` flag to skip the `git add` step in daily workflows and how to use the `-m` flag to pass a commit message in on the command line instead of firing up an editor.

In [復原](#) we cover using the `--amend` option to redo the most recent commit.

In [簡述分支](#), we go into much more detail about what `git commit` does and why it does it like that.

We looked at how to sign commits cryptographically with the `-S` flag in [Signing Commits](#).

Finally, we take a look at what the `git commit` command does in the background and how it's actually implemented in [Commit Objects](#).

git reset

The `git reset` command is primarily used to undo things, as you can possibly tell by the verb. It moves around the `HEAD` pointer and optionally changes the `index` or staging area and can also optionally change the working directory if you use `--hard`. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of `git reset` in [將已預存的檔案移出預存區](#), where we use it to unstage a file we had run `git add` on.

We then cover it in quite some detail in [Reset Demystified](#), which is entirely devoted to explaining this command.

We use `git reset --hard` to abort a merge in [Aborting a Merge](#), where we also use `git merge --abort`, which is a bit of a wrapper for the `git reset` command.

git rm

The `git rm` command is used to remove files from the staging area and working directory for Git. It is similar to `git add` in that it stages a removal of a file for the next commit.

We cover the `git rm` command in some detail in [移除檔案](#), including recursively removing files and only removing files from the staging area but leaving them in the working directory with `--cached`.

The only other differing use of `git rm` in the book is in [Removing Objects](#) where we briefly use and explain the `--ignore-unmatch` when running `git filter-branch`, which simply makes it not error

out when the file we are trying to remove doesn't exist. This can be useful for scripting purposes.

git mv

The `git mv` command is a thin convenience command to move a file and then run `git add` on the new file and `git rm` on the old file.

We only briefly mention this command in [移動檔案](#).

git clean

The `git clean` command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files.

We cover many of the options and scenarios in which you might use the clean command in [Cleaning your Working Directory](#).

Branching and Merging

There are just a handful of commands that implement most of the branching and merging functionality in Git.

git branch

The `git branch` command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

Most of [使用 Git 分支](#) is dedicated to the `branch` command and it's used throughout the entire chapter. We first introduce it in [建立一個新的分支](#) and we go through most of its other features (listing and deleting) in [分支管理](#).

In [Tracking Branches](#) we use the `git branch -u` option to set up a tracking branch.

Finally, we go through some of what it does in the background in [Git References](#).

git checkout

The `git checkout` command is used to switch branches and check content out into your working directory.

We first encounter the command in [在分支之間切換](#) along with the `git branch` command.

We see how to use it to start tracking branches with the `--track` flag in [Tracking Branches](#).

We use it to reintroduce file conflicts with `--conflict=diff3` in [Checking Out Conflicts](#).

We go into closer detail on its relationship with `git reset` in [Reset Demystified](#).

Finally, we go into some implementation detail in [The HEAD](#).

git merge

The `git merge` tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

The `git merge` command was first introduced in [分支的基本用法](#). Though it is used in various places in the book, there are very few variations of the `merge` command—generally just `git merge <branch>` with the name of the single branch you want to merge in.

We covered how to do a squashed merge (where Git merges the work but pretends like it's just a new commit without recording the history of the branch you're merging in) at the very end of [Fork 公眾專案](#).

We went over a lot about the merge process and command, including the `-Xignore-space-change` command and the `--abort` flag to abort a problem merge in [Advanced Merging](#).

We learned how to verify signatures before merging if your project is using GPG signing in [Signing Commits](#).

Finally, we learned about Subtree merging in [Subtree Merging](#).

git mergetool

The `git mergetool` command simply launches an external merge helper in case you have issues with a merge in Git.

We mention it quickly in [合併衝突的基本解法](#) and go into detail on how to implement your own external merge tool in [External Merge and Diff Tools](#).

git log

The `git log` command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you're currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

This command is used in nearly every chapter of the book to demonstrate the history of a project.

We introduce the command and cover it in some depth in [檢視提交的歷史記錄](#). There we look at the `-p` and `--stat` option to get an idea of what was introduced in each commit and the `--pretty` and `--oneline` options to view the history more concisely, along with some simple date and author filtering options.

In [建立一個新的分支](#) we use it with the `--decorate` option to easily visualize where our branch pointers are located and we also use the `--graph` option to see what divergent histories look like.

In [私有的小團隊](#) and [Commit Ranges](#) we cover the `branchA..branchB` syntax to use the `git log` command to see what commits are unique to a branch relative to another branch. In [Commit Ranges](#) we go through this fairly extensively.

In [Merge Log](#) and [Triple Dot](#) we cover using the `branchA...branchB` format and the `--left-right` syntax to see what is in one branch or the other but not in both. In [Merge Log](#) we also look at how to

use the `--merge` option to help with merge conflict debugging as well as using the `--cc` option to look at merge commit conflicts in your history.

In [RefLog Shortnames](#) we use the `-g` option to view the Git reflog through this tool instead of doing branch traversal.

In [Searching](#) we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In [Signing Commits](#) we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch.

This is basically entirely covered in [Stashing and Cleaning](#).

git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

This command is introduced and covered in detail in [標籤](#) and we use it in practice in [為釋出的版本加上標籤](#).

We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in [Signing Your Work](#).

Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

We first look at this command in [從你的遠端獲取或拉取](#) and we continue to see examples of its use in [遠端分支](#).

We also use it in several of the examples in [對專案進行貢獻](#).

We use it to fetch a single specific reference that is outside of the default space in [Pull Request 參照](#) and we see how to fetch from a bundle in [Bundling](#).

We set up highly custom refsspecs in order to make `git fetch` do something a little different than the default in [The Refspec](#).

git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you’re on.

We introduce it quickly in [從你的遠端獲取或拉取](#) and show how to see what it will merge if you run it in [檢視遠端](#).

We also see how to use it to help with rebasing difficulties in [Rebase When You Rebase](#).

We show how to use it with a URL to pull in changes in a one-off fashion in [切換到遠端分支](#).

Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in [Signing Commits](#).

git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

We first look at the `git push` command in [推送到你的遠端](#). Here we cover the basics of pushing a branch to a remote repository. In [Pushing](#) we go a little deeper into pushing specific branches and in [Tracking Branches](#) we see how to set up tracking branches to automatically push to. In [刪除遠端分支](#) we use the `--delete` flag to delete a branch on the server with `git push`.

Throughout [對專案進行貢獻](#) we see several examples of using `git push` to share work on branches through multiple remotes.

We see how to use it to share tags that you have made with the `--tags` option in [分享標籤](#).

In [Publishing Submodule Changes](#) we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

In [Other Client Hooks](#) we talk briefly about the `pre-push` hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.

Finally, in [Pushing Refspecs](#) we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as “origin” so you don’t have to type them out all the time. You can have several of these and the `git remote` command is used to add, change and delete them.

This command is covered in detail in [與遠端協同工作](#), including listing, adding, removing and renaming them.

It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote`

add <name> <url> format.

git archive

The `git archive` command is used to create an archive file of a specific snapshot of the project.

We use `git archive` to create a tarball of a project for sharing in [準備釋出一個版本](#).

git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The `submodule` command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources.

This command is only mentioned and entirely covered in [Submodules](#).

Inspection and Comparison

git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

We first use it to show annotated tag information in [有註解的標籤](#).

Later we use it quite a bit in [Revision Selection](#) to show the commits that our various revision selections resolve to.

One of the more interesting things we do with `git show` is in [Manual File Re-merging](#) to extract specific file contents of various stages during a merge conflict.

git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

We showed how to use it to create a nice changelog in [簡短的日誌](#).

git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It's a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.

We use `git describe` in [產生一個建置編號](#) and [準備釋出一個版本](#) to get a string to name our release file after.

Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from

figuring out where something was introduced to figuring out who introduced it.

git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

It is fully covered in [Binary Search](#) and is only mentioned in that section.

git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

It is covered in [File Annotation](#) and is only mentioned in that section.

git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

It is covered in [Git Grep](#) and is only mentioned in that section.

Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as though the commit series is a series of patches. These commands help you manage your branches in this manner.

git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you’re currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

Cherry picking is described and demonstrated in [行合與挑揀的工作流程](#).

git rebase

The `git rebase` command is basically an automated `cherry-pick`. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

Rebasing is covered in detail in [行合](#), including covering the collaborative issues involved with rebasing branches that are already public.

We use it in practice during an example of splitting your history into two separate repositories in [Replace](#), using the `--onto` flag as well.

We go through running into a merge conflict during rebasing in [Rerere](#).

We also use it in an interactive scripting mode with the `-i` option in [Changing Multiple Commit Messages](#).

git revert

The `git revert` command is essentially a reverse `git cherry-pick`. It creates a new commit that applies the exact opposite of the change introduced in the commit you’re targeting, essentially undoing or reverting it.

We use this in [Reverse the commit](#) to undo a merge commit.

Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

git apply

The `git apply` command applies a patch created with the `git diff` or even GNU diff command. It is similar to what the `patch` command might do with a few small differences.

We demonstrate using it and the circumstances in which you might do so in [套用從電子郵件來的補丁](#).

git am

The `git am` command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

We covered usage and workflow around `git am` in [使用 am 命令套用補丁](#) including using the `--resolved`, `-i` and `-3` options.

There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in [Email Workflow Hooks](#).

We also use it to apply patch formatted GitHub Pull Request changes in [電郵通知](#).

git format-patch

The `git format-patch` command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted.

We go through an example of contributing to a project using the `git format-patch` tool in [透過電子郵件貢獻到公眾專案](#).

git imap-send

The `git imap-send` command uploads a mailbox generated with `git format-patch` into an IMAP drafts folder.

We go through an example of contributing to a project by sending patches with the `git imap-send` tool in [透過電子郵件貢獻到公眾專案](#).

git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email.

We go through an example of contributing to a project by sending patches with the `git send-email` tool in [透過電子郵件貢獻到公眾專案](#).

git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in [Fork 公眾專案](#).

External Systems

Git comes with a few commands to integrate with other version control systems.

git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

This command is covered in depth in [Git and Subversion](#).

git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record.

This command is covered in depth in [A Custom Importer](#).

Administration

If you’re administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

git gc

The `git gc` command runs “garbage collection” on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in [Maintenance](#).

git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies.

We only quickly use this once in [Data Recovery](#) to search for dangling objects.

git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in [RefLog Shortnames](#), where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.

We also go through a practical example of recovering such a lost branch in [Data Recovery](#).

git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

In [Removing a File from Every Commit](#) we explain the command and explore several different options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.

In [Git-p4](#) and [TFS](#) we use it to fix up imported external repositories.

Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

The first one we encounter is `ls-remote` in [Pull Request 參照](#) which we use to look at the raw references on the server.

We use `ls-files` in [Manual File Re-merging](#), [Rerere](#) and [The Index](#) to take a more raw look at what your staging area looks like.

We also mention `rev-parse` in [Branch References](#) to take just about any string and turn it into an object SHA-1.

However, most of the low level plumbing commands we cover are in [Git Internals](#), which is more or less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.

Index