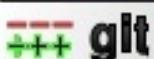


THE EXPERT'S VOICE® IN SOFTWARE DEVELOPMENT

Pro Git

*Everything you need to know about
the Git distributed source control tool*



Scott Chacon

Foreword by Junio C Hamano, Git project leader

Apress®

Table of Contents

Introduction	0
開始	1
關於版本控制	1.1
Git 的簡史	1.2
Git 基礎要點	1.3
安裝 Git	1.4
初次設定 Git	1.5
取得說明文件	1.6
總結	1.7
Git 基礎	2
取得 Git 儲存庫	2.1
提交更新到儲存庫	2.2
檢視提交的歷史記錄	2.3
復原	2.4
與遠端協同工作	2.5
標籤	2.6
提示和技巧	2.7
總結	2.8
Git 分支	3
何謂分支	3.1
分支的新建與合併	3.2
分支的管理	3.3
利用分支進行開發的工作流程	3.4
遠端分支	3.5
分支的衍合	3.6
小結	3.7
伺服器上的 Git	4
協議	4.1
在伺服器上部署 Git	4.2
生成 SSH 公開金鑰	4.3

架設伺服器	4.4
公共訪問	4.5
GitWeb	4.6
Gitosis	4.7
Gitolite	4.8
Git 守護進程	4.9
Git 託管服務	4.10
小結	4.11
分散式 Git	5
分散式工作流程	5.1
為專案作貢獻	5.2
專案的管理	5.3
小結	5.4
Git 工具	6
選擇修訂版本	6.1
互動式暫存	6.2
儲藏 (Stashing)	6.3
重寫歷史	6.4
使用 Git 做 Debug	6.5
子模組 (Submodules)	6.6
子樹合併	6.7
總結	6.8
Git 客製化	7
Git 配置	7.1
Git 屬性	7.2
Git Hooks	7.3
Git 強制策略實例	7.4
總結	7.5
Git 與其他系統	8
Git 與 Subversion	8.1
遷移到 Git	8.2
總結	8.3
Git 內部原理	9
底層命令 (Plumbing) 和高層命令 (Porcelain)	9.1

Git 物件	9.2
Git References	9.3
Packfiles	9.4
The Refspec	9.5
傳輸協議	9.6
維護及資料復原	9.7
總結	9.8

Learn Git

This is a GitBook version of the Scott Chacon's book: [Pro Git](#).

The entire Pro Git book, written by Scott Chacon and published by Apress, is available here. All content is licensed under the [Creative Commons Attribution Non Commercial Share Alike 3.0 license](#). Print versions of the book are available on [Amazon.com](#).

This book is also an example of a book that can be generated in multiple languages.

WARNING: This repo is automatically generated by [progit-to-gitbook](#). Please submit all pull requests to [Pro Git](#).

開始

本章介紹Git的相關知識。先從講解一些版本控制工具的背景知識開始，然後試著在讀者的系統將Git跑起來，最後則是設定它。在本章結束後，讀者應瞭解為什麼Git如此流行、為什麼讀者應該利用它、以及完成使用它的準備工作。

關於版本控制

什麼是版本控制？以及為什麼讀者會在意它？版本控制是一個能夠記錄一個或一組檔案在某一段時間的變更，使得讀者以後能收回特定版本的系統。在本書的範例中，讀者會學到如何對軟體的原始碼做版本控制。即使實際上讀者幾乎可以針對電腦上任意型態的檔案做版本控制。

若讀者是繪圖或網頁設計師且想要記錄每一版影像或版面配置（這也通常是讀者想要做的），採用版本控制系統（VCS）做這件事是非常明智的。它允許讀者將檔案復原到原本的狀態、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，等。使用版本控制系統一般也意謂著若讀者做了一些傻事或者遺失檔案，讀者能很容易地回復到原先的狀態。更進一步，僅需付出很小的代價即可得到這些優點。

本地端版本控制

許多讀者採用複製檔案到其它目錄的方式來做版本控制（若他們夠聰明的話，或許會是有記錄時間戳記的目錄）。因為它很簡單，這是個很常見的方法；但它也很容易出錯。讀者很容易就忘記在哪個目錄，並不小心地把錯誤的檔案寫入、或者複製到不想要的檔案。

為了解決這問題，程式設計師在很久以前開發了本地端的版本控制系統，具備簡單的資料庫，用來記載檔案的所有變更記錄（參考圖1-1）。

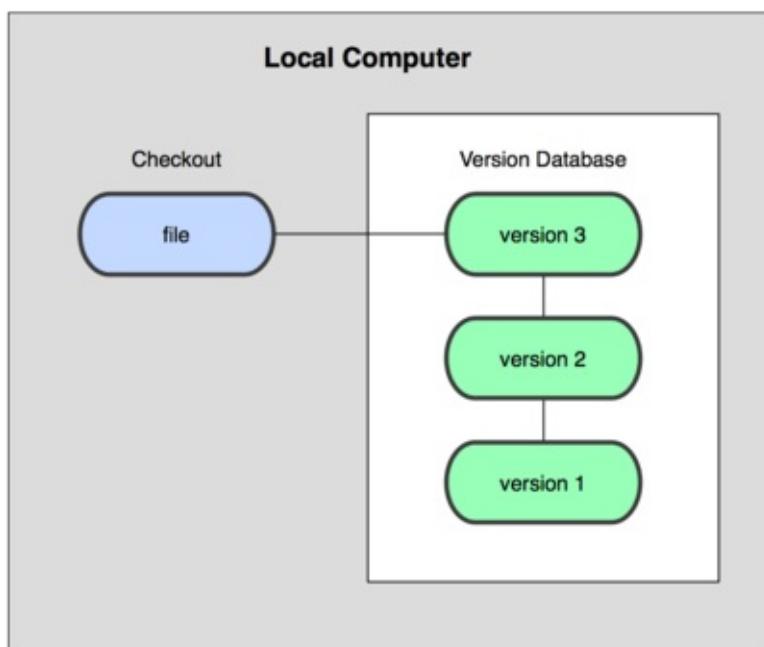


圖1-1. 本地端版本控制流程圖。

這種版本控制工具中最流行的是rcs，目前仍存在於許多電腦。即使是流行的Mac OS X作業系統，都會在讀者安裝開發工具時安裝rcs命令。此工具基本上以特殊的格式記錄修補集合(patch set，即檔案從一個版本變更到另一個版本所需資訊)，並儲存於磁碟上。它就可以藉由套用各修補集合產生各時間點的檔案內容。

集中式版本控制系统

接下來人們遇到的主要問題是需要在多種其它系統上的開發協同作業。為了解決此問題，集中式版本控制系统(Centralized Version Control Systems，簡稱CVCSs)被發展出來。此系統，如：CVS、Subversion及Perforce皆具備單一伺服器，記錄所有版本的檔案，且有多個客戶端從伺服器從伺服器取出檔案。在多年後，這已經是版本控制系統的標準（參考圖1-2）。

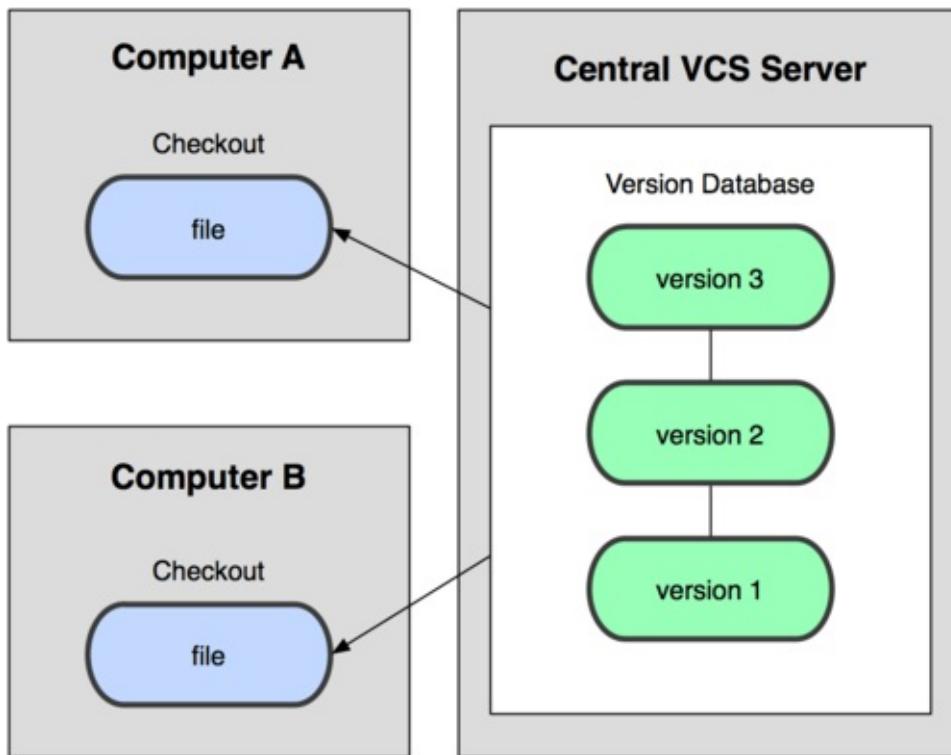


圖1-2. 集中式版本控制系统

這樣的配置提供了很多優點，特別是相較於本地端的版本控制系统來說。例如：每個人皆能得知其它人對此專案做了些什麼修改有一定程度的瞭解。管理員可調整存取權限，限制各使用者能做的事。而且維護集中式版本控制系统也比維護散落在各使用者端的資料庫來的容易。

然而，這樣的配置也有嚴重的缺點。最明顯的就是無法連上伺服器時。如果伺服器當機一個小時，在這段時間中沒有人能進行協同開發的工作或者將變更的部份傳遞給其他使用者。如果伺服器用來儲存資料庫的硬碟損毀，而且沒有相關的偏份資料。除了使用者已取到本地端電腦的版本外，包含該專案開發的歷史的所有資訊都會遺失。本地端版本控制系统也會有同樣的問題，只要使用者將整個專案的開發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

這就是分散式版本控制系統(Distributed Version Control Systems, 簡稱DVCSs)被引入的原因。在分散式版本控制系統，諸如：Git、Mercurial、Bazaar、Darcs。客戶端不只是取出最後一版的檔案，而是完整複製整個儲存庫。即使是整個系統賴以運作的電腦損毀，皆可將任何一個客戶端先前複製的資料還原到伺服器。每一次的取出動作實際上就是完整備份整個儲存庫。（參考圖1-3）

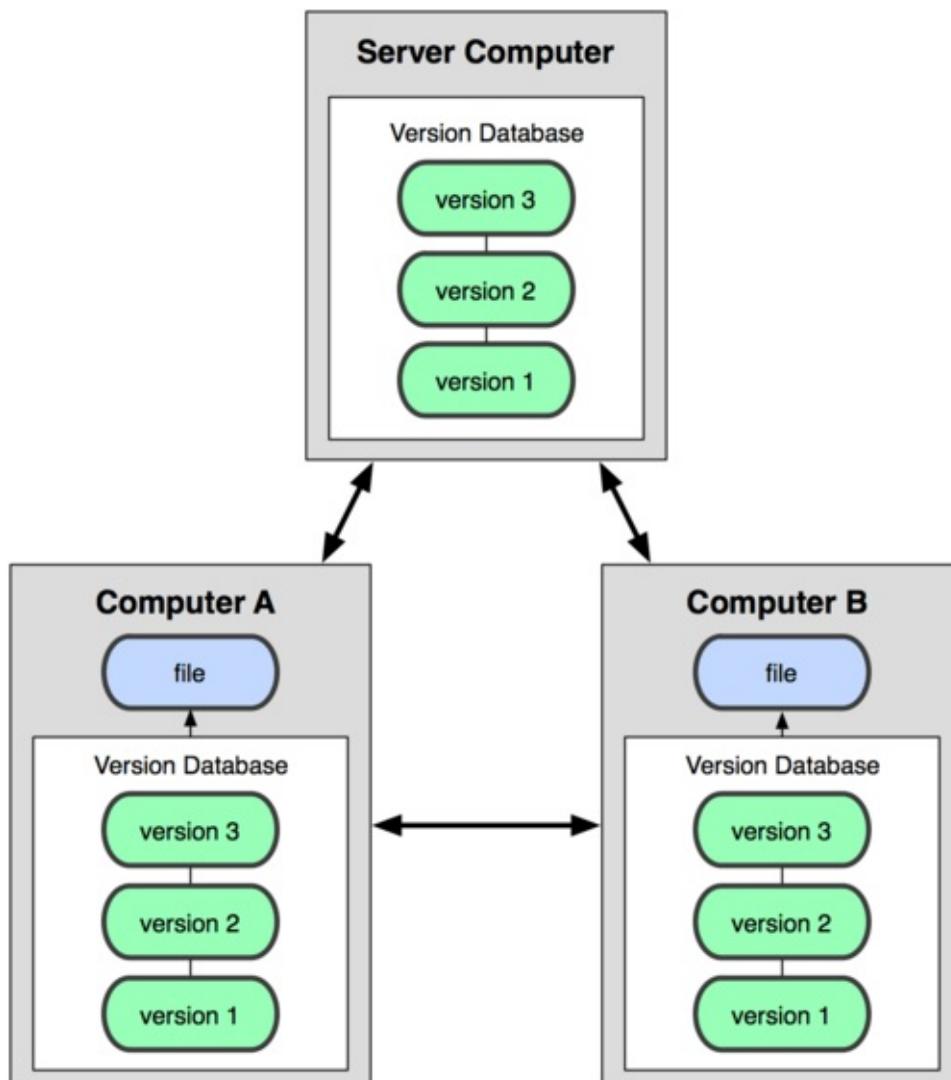


圖1-3. 分散式版本控制系統

更進一步來說，許多這類型的系統皆能同時與數個遠端的機器同時運作。因此讀者能同時與許多不同群組的人們協同開發同一個專案。這允許讀者設定多種集中式系統做不到的工作流程，如：階層式模式。

Git 的簡史

如同許多生命中美好的事物一樣，Git從有一點創造性的破壞及激烈的討論中誕生。Linux kernel 是開放原始碼中相當大的專案。在 Linux kernel 大部份的維護時間內（1991～2002），修改該軟體的方式通常以多個修補檔及壓縮檔流通。在2002年，Linux kernel 開始採用名為 BitKeeper 的商業分散式版本控制系統。

在 2005 年，開發 Linux kernel 的社群與開發 BitKeeper 的商業公司的關係走向決裂，也無法再免費使用該工具。這告訴了 Linux 社群及 Linux 之父 Linus Torvalds，該是基於使用 BitKeeper 得到的經驗，開發自有的工具的時候。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）

自從 2005 年誕生後，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。（參考第三章）

Git 基礎要點

那麼，簡單地說，Git是一個什麼樣的系統？這一章節是非常重要的。若讀者瞭解Git的本質以及運作的基礎，那麼使用起來就會很輕鬆且有效率。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。這將會幫助讀者使用此工具時發生不必要的誤會。Git儲存資料及運作它們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讀者更準確的使用此工具。

記錄檔案快照，而不是差異的部份

Git與其它版本控制系統（包含Subversion以及與它相關的）的差別是如何處理資料的方式。一般來說，大部份其它系統記錄資訊是一連串檔案更動的內容。如圖1-4所示。這些系統（CVS、Subversion、Perforce、Bazaar等等）儲存一組基本的檔案以及對應這些檔案隨時間遞增的更動資料。

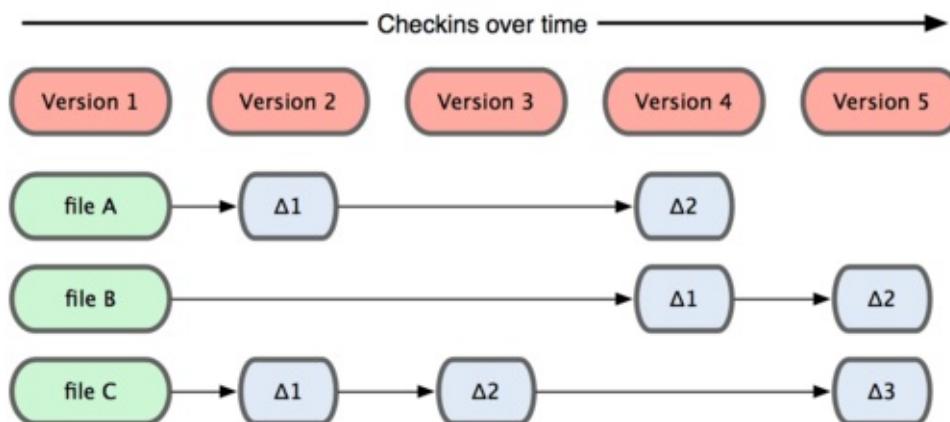


圖1-4. 其它系統傾向儲存每個檔案更動的資料。

Git並不以此種方式儲存資料。而是將其視為小型檔案系統的一組快照(Snapshot)。每一次讀者提交更新時、或者儲存目前專案的狀態到Git時。基本上它為當時的資料做一組快照並記錄參考到該快照的參考點。為了講求效率，只要檔案沒有變更，Git不會再度儲存該檔案，而是記錄到前一次的相同檔案的連結。Git的工作方式如圖1-5所示。

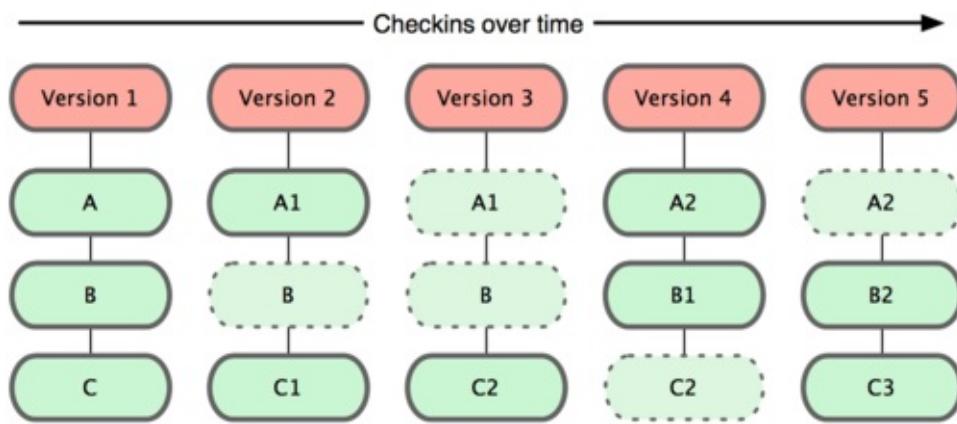


圖1-5. Git儲存每次專案更新時的快照。

這是Git與所有其它版本控制系統最重要的區別。它完全顛覆傳統版本控制的作法。這使用Git更像一個上層具備更強大工具的小型檔案系統，而不只是版本控制系統。我們將會在第三章介紹分支時，提到採用此種作法的優點。

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。一般來說並需要到網路上其它電腦提取的資訊。若讀者使用集中式版本控制系統，大部份的動作皆包含網路延遲的成本。這項特點讓你覺得Git處理資料的速度飛快。因為整個專案的歷史皆存在你的硬碟中，大部份的運作看起來幾乎都是馬上完成。

例如：想要瀏覽專案的歷史時，Git不需要到伺服器下載歷史，而是從本地端的資料庫讀取並顯示即可。這意謂著讀者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及現在版本的差別，Git可在本地端找出一個月前的檔案並在比對兩者的差異，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並在本地端比對。

這意謂著即使讀者已離線，或者切斷VPN連線後，也很少有讀者無法執行的動作。若讀者在飛機或火車上，並想要做一些工作，讀者在取得可上傳的網路前仍可很快樂地提交更新。若讀者回到家且無法讓VPN連線程式正常運作，讀者仍然可繼續工作。在許多其它系統幾乎是無法做這些事或者必須付出很大代價。以Perforce為例，在無法連到伺服器時讀者做不了多少事。以Subversion及CVS為例，雖然讀者能編輯檔案，但因為資料庫此時是離線的，讀者無法提交更新到資料庫。這看起來可能還不是什麼大問題，但讀者可能驚訝Git有這麼大的不同。

Git能檢查完整性

在Git中所有的物件在儲存前都會被計算查核碼(`checksum`)並以查核碼檢索物件。這意謂著Git不可能不清楚任何檔案或目錄的內容已被更動。此功能內建在Git底層並整合到它的設計哲學。Git能夠馬上察覺傳輸時的遺失或是檔案的毀損。

Git用來計算查核碼的機制稱為SHA-1雜湊法。它由40個十六進制的字母(0–9 and a–f)組成的字串組成，基於Git的檔案內容或者目錄結構計算。查核碼看起來如下所示：

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

讀者會在Git中到處都看到雜湊值，因為它到處被使用。事實上Git以檔案內容的雜湊值定址出檔案在資料庫的位址，而不是以檔案的名稱定址。

Git 通常只增加資料

當讀者使用Git，幾乎所有的動作只是增加資料到Git的資料庫。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，讀者有可能會遺失或者搞混尚未提交的更新。但是在提交快照到Git後，很少會有遺失的情況，特別是讀者定期將資料庫更新到其它儲存庫。

這讓使用Git可輕鬆地像在玩一樣，因為我們知道我們可以進行任何實驗而不會破壞任何東西。在第九章的“底層細節”中，我們會進一步討論Git如何儲存資料，以及讀者如何復原看似遺失的資料。

三種狀態

現在，注意。若讀者希望接下來的學習過程順利些，這是關於Git的重要且需記住的事項。Git有三種表達檔案的狀態：已提交(committed)、已修改(modified)及已暫存(staged)。已提交意謂著資料已安全地存在讀者的本地端資料庫。已修改代表著讀者已修改檔案但尚未提交到資料庫。已暫存意謂著讀者標記已修改檔案目前的版本到下一次提供的快照。

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

Local Operations

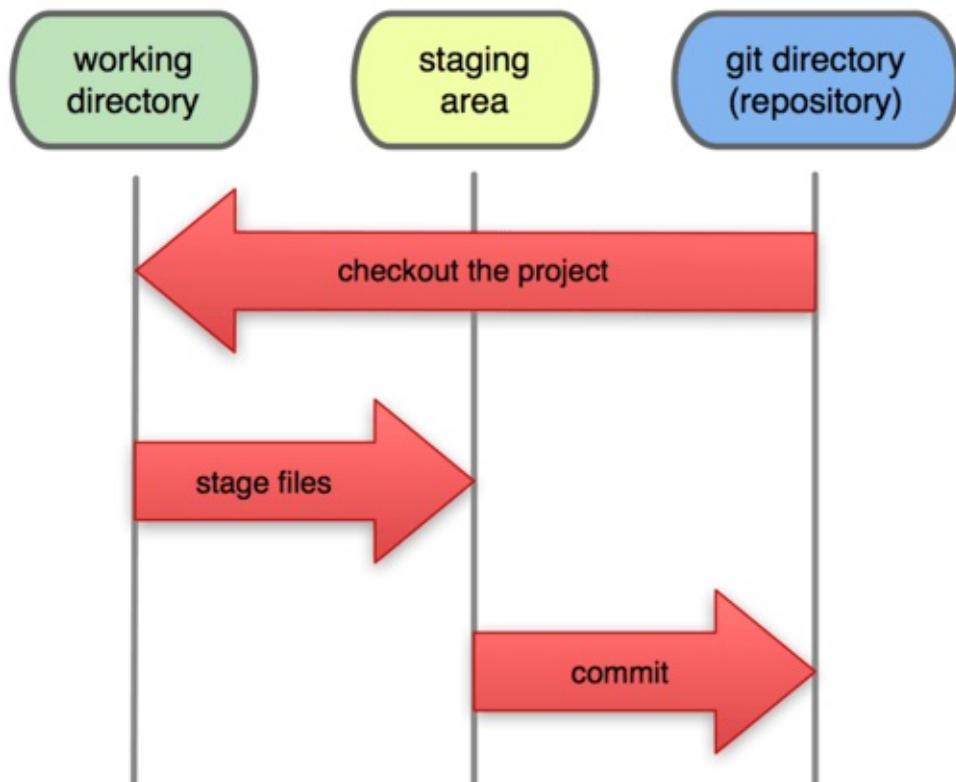


圖1-6. 工作目錄、暫存區域及git目錄。

Git目錄是Git用來儲存讀者的專案的元數據及物件資料庫。這是Git最重要的部份，而且它是當讀者從其它電腦複製儲存庫時會複製過來的。

工作目錄是專案被取出的某一個版本。這些檔案從Git目錄內被壓縮過的資料庫中拉出來並放在磁碟機供讀者使用或修改。

暫存區域是一個單純的檔案，一般來說放在Git目錄，儲存關於下一個提交的資訊。有時稱為索引，但現在將它稱為暫存區域已開始成為標準。

基本Git工作流程大致如下：

1. 讀者修改工作目錄內的檔案。
2. 暫存檔案，將檔案的快照新增到暫存區域。
3. 做提交的動作，這會讓存在暫存區域的檔案快照永久地儲存在Git目錄。

在Git目錄內特定版本的檔案被認定為已提交。若檔案被修改且被增加到暫存區域，稱為被暫存。若檔案被取出後有被修改，但未被暫存，稱為被修改。在第二章讀者會學到更多關於這些狀態以及如何利用它們的優點或者整個略過暫存步驟。

安裝Git

讓我們開始使用Git。首先讀者要做的事是安裝Git。讀者有很多取得它們的方法。主要的兩種分別是從原始碼安裝或者從讀者使用平台現存的套件安裝。

從原始碼安裝

若讀者有能力的話，從原始碼安裝是非常有用的。因為讀者能取得最新版本。每一版Git通常都會包含有用的UI改善。因此取得最新版本通常是最好的，只要讀者覺得編譯軟體的原始碼是很容易的。許多Linux發行套件通常都是附上非常舊的套件。除非讀者使用的發行套件非常新或者使用向後相容的移植版本。從原始碼安裝通常是最好的選擇。

要安裝Git，讀者需要先安裝它需要的程式庫：curl、zlib、openssl、expat及libiconv。例如：若讀者的系統有yum（如：Fedora）或apt-get（如：以Debian為基礎的系統），讀者可使用下列任一命令安裝所有需要的程式庫：

```
$ yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
libbz-dev libssl-dev
```

當讀者安裝所有必要的程式庫，讀者可到Git的網站取得最新版本：

```
http://git-scm.com/download
```

接著，編譯及安裝：

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

在這些工作完成後，讀者也可以使用Git取得Git的更新版：

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

在Linux系統安裝

若讀者想使用二進位安裝程式安裝Git到Linux，一般來說讀者可經由發行套件提供的套件管理工具完成此工作。若讀者使用Fedora，可使用 yum：

```
$ yum install git-core
```

若讀者在以Debian為基礎的發行套件，如：Ubuntu。試試 apt-get：

```
$ apt-get install git
```

在Mac系統安裝

有兩種很容易將Git安裝到Mac的方法。最簡單的是使用圖形化界面的Git安裝程式，可從Google Code下載（圖1-7）：

```
http://code.google.com/p/git-osx-installer
```

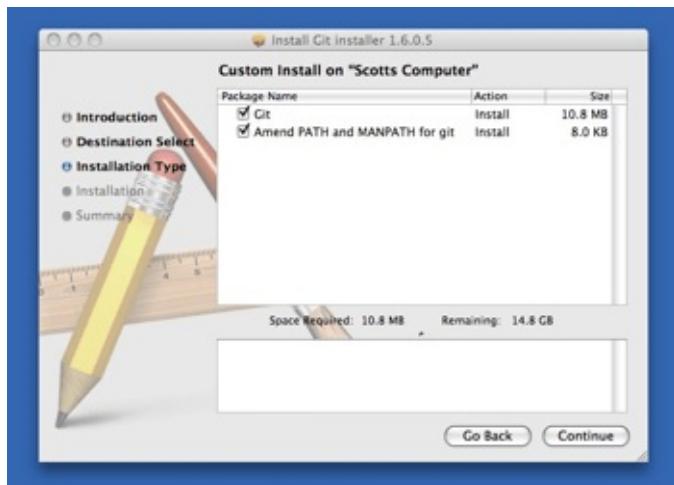


圖1-7. Git OS X 安裝程式。

藉由MacPorts安裝Git是另一種主要的方法。若讀者已安裝MacPorts，使用下列命令安裝Git

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

讀者完全不需要安裝所有的額外套件，但讀者可能會想要加上 +svn 參數，以利於使用Git讀寫Subversion儲存庫（參考第8章）。

在Windows系統安裝

在Windows系統安裝Git相當地容易。msysGit專案已提供相當容易安裝的程序。只要從Google Code網頁下載安裝程式並執行即可：

```
http://msysgit.github.com/
```

在安裝完畢後，讀者同時會有命令列版本（包含SSH客戶端程式）及標準的圖形界面版本。

Note on Windows usage: you should use Git with the provided msysGit shell (Unix style), it allows to use the complex lines of command given in this book. If you need, for some reason, to use the native Windows shell / command line console, you have to use double quotes instead of simple quotes (for parameters with spaces in them) and you must quote the parameters ending with the circumflex accent (^) if they are last on the line, as it is a continuation symbol in Windows.

初次設定Git

現在讀者的系統已安裝了Git，讀者可能想要做一些客製化的動作。讀者應只需要做這些工作一次。這些設定在更新版本時會被保留下來。讀者可藉由再度執行命令的方式再度修改這些設定。

Git附帶名為 `git config` 的工具，允許讀者取得及設定組態參數，可用來決定Git外觀及運作。這些參數可存放在以下三個地方：

- 檔案 `/etc/gitconfig`：包含給該系統所有使用者的儲存庫使用的數值。只要讀者傳遞`--system`參數給 `git config`，它就會讀取或者寫入參數到這個檔案
- 檔案 `~/.gitconfig`：給讀者自己的帳號使用。傳遞`--global`參數給 `git config`，它就會讀取或者寫入參數到這個檔案。
- 儲存庫內的設定檔，也就是 `.git/config`：僅給所在的儲存庫使用。每個階級的設定會覆寫上一層的。因此，`git/config` 內的設定值的優先權高過 `/etc/config`。

在Windows系統，Git在 `$HOME` 目錄(對大部份使用者來說是 `c:\Documents and Settings\$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。

設定識別資料

讀者安裝Git後首先應該做的事是指定使用者名稱及電子郵件帳號。這一點非常重要，因為每次Git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
```

再說一次，若讀者有指定 `--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。若讀者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。

指定編輯器

現在讀者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需要讀者輸入訊息時會叫用它。預設情況下，Git會使用系統預設的編輯器，一般來說是Vi或Vim。若讀者想指定不同的編輯器，例如：Emacs。可執行下列指令：

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

指定合併工具

另外一個對讀者來說有用的選項是設定解決合併失敗時，讀者慣用的合併工具。假設讀者想使用vimdiff：

```
$ git config --global merge.tool vimdiff
```

Git能接受kdiff3、tkdiff、meld、xxdiff、emerge、vimdiff、gvimdiff、ecmerge及opendiff做為合併工具。讀者可設定自訂的工具。詳情參考第七章。

檢查讀者的設定

若讀者想確認設定值，可使用 `git config --list` 命令列出所有Git能找到的設定值：

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

讀者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。

使用者也可以下列命令 `git config` 設定名稱，檢視Git認為該設定名稱的設定值：

```
$ git config user.name
Scott Chacon
```

取得說明文件

若讀者在使用 Git 時需要幫助，有三種方法取得任何 Git 命令的手冊：

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

例如：讀者可以下列命令取得 config 命令的手冊

```
$ git help config
```

這些命令對讀者是很有幫助的，因為讀者可在任意地方取得它們，即使已離線。若手冊及這本書不足以幫助讀者，且讀者需要更進一步的協助。讀者可試著進入 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。

總結

目前讀者應該對於Git有一些基本的瞭解，而且知道它與其它集中式版本控制系統的不同，其中有些可能是讀者正在使用的。讀者的系統現在也應該有一套可動作的Git且已設定好讀者個人的識別資料。現在正是學習一些Git基本操作的好時機。

Git 基礎

若讀者只需要閱讀一個章節即可開始使用 Git，這章就是你所需要的。本章節涵蓋讀者大部份用到 Git 時需要使用的所有基本命令。在讀完本章節後，讀者應該有能力組態及初始化一個儲存庫、開始及停止追蹤檔案、暫存及提交更新。還會提到如何讓 Git 忽略某些檔案、如何輕鬆且很快地救回失誤、如何瀏覽讀者的專案歷史及觀看各個已提交的更新之間的變更、以及如何從遠端儲存庫 拉 更新下來或將更新 推 上去。

取得Git儲存庫

讀者可使用兩種主要的方法取得一個Git儲存庫。第一種是將現有的專案或者目錄匯入Git。第二種從其它伺服器複製一份已存在的Git儲存庫。

在現有目錄初始化儲存庫

若讀者要開始使用 Git 追蹤現有的專案，只需要進入該專案的目錄並執行：

```
$ git init
```

這個命令會建立名為 `.git` 的子目錄，該目錄包含一個Git儲存庫架構必要的所有檔案。目前來說，專案內任何檔案都還沒有被追蹤。（關於`.git`目錄內有些什麼檔案，可參考第九章）

若讀者想要開始對現有的檔案開始做版本控制（除了空的目錄以外），讀者也許應該開始追蹤這些檔案並做第一次的提交。讀者能以少數的`git add`命令指定要追蹤的檔案，並將它們提交：

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

這些命令執行完畢大約只需要一分鐘。現在，讀者已經有個追蹤部份檔案及第一次提交內容的Git儲存庫。

複製現有的儲存庫

若讀者想要取得現有的Git儲存庫的複本（例如：讀者想要散佈的），那需要使用的命令是 `git clone`。若讀者熟悉其它版本控制系統，例如：**Subversion**，讀者應該注意這個命令是複製(clone)，而不是取出特定版本(checkout)。這一點非常重要，Git取得的是大部份伺服器端所有的資料複本。該專案歷史中所有檔案的所有版本都在讀者執行過 `git clone` 後拉回來。事實上，若伺服器的磁碟機損毀，讀者可使用任何一個客戶端的複本還原伺服器為當初取得該複本的狀態（讀者可能會遺失一些僅存於伺服器的攔截程式，不過所有版本的資料都健在），參考第四章取得更多資訊。

讀者可以 `git clone [url]`，複製一個儲存庫。例如：若讀者想複製名為Grit的Ruby Git程式庫，可以執行下列命令：

```
$ git clone git://github.com/schacon/grit.git
```

接下來會有個名為 `grit` 的目錄被建立，並在其下初始化名為 `.git` 的目錄。拉下所有存在該儲存庫的所有資料，並取出最新版本為工作複本。若讀者進入新建立的 `grit` 目錄，會看到專案的檔案都在這兒，且可使用。若讀者想複製儲存庫到 `grit` 以外其它名字的目錄，只需要在下一個參數指定即可：

```
$ git clone git://github.com/schacon/grit.git mygrit
```

這個命令做的事大致如同上一個命令，只不過目的目錄名為 `mygrit`。

Git 提供很多種協定給讀者使用。上一個範例採用 `git://` 協定，讀者可能會看過 `http(s)://` 或者 `user@server:/path.git` 等使用 SSH 傳輸的協定。在第四章會介紹設定存取伺服器上的 Git 儲存庫的所有可用的選項，以及它們的優點及缺點。

提交更新到儲存庫

讀者現在有一個貨真價實的Git儲存庫，而且有一份已放到工作複本的該專案的檔案。讀者需要做一些修改並提交這些更動的快照到儲存庫，當這些修改到達讀者想要記錄狀態的情況。

記住工作目錄內的每個檔案可能為兩種狀態的任一種：追蹤或者尚未被追蹤。被追蹤的檔案是最近的快照；它們可被復原、修改，或者暫存。未被追蹤的檔案則是其它未在最近快照也未被暫存的任何檔案。當讀者第一次複製儲存器時，讀者所有檔案都是被追蹤且未被修改的。因為讀者剛取出它們而且尚未更改做任何修改。

只要讀者編輯任何已被追蹤的檔案。Git將它們視為被更動的，因為讀者將它們改成與最後一次提交不同。讀者暫存這些已更動檔案並提供所有被暫存的更新，並重複此週期。此生命週期如圖2-1所示。

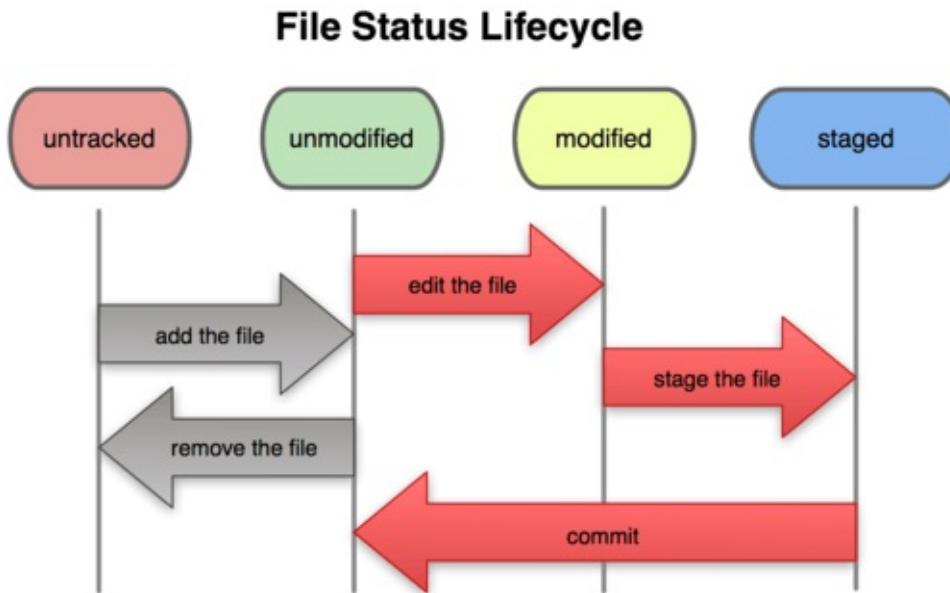


圖2-1. 檔案狀態的生命週期。

檢視檔案的狀態

主要給讀者用來檢視檔案的狀態是 `git status` 命令。若讀者在複製完複本後馬上執行此命令，會看到如下的文字：

```
$ git status
On branch master
nothing to commit, working directory clean
```

這意謂著讀者有一份乾淨的工作目錄（換句話說，沒有未被追蹤或已被修改的檔案）。Git未看到任何未被追蹤的檔案，否則會將它們列出。最後，這個命令告訴讀者目前在哪一個分支。到目前為止，一直都是master，這是預設的。目前讀者不用考慮它。下一個章節會詳細介紹分支。

假設讀者新增一些檔案到專案，如 README。若該檔案先前並不存在，執行 git status 命令後，讀者會看到未被追蹤的檔案，如下：

```
$ vim README
$ git status
On branch 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)
```

讀者可看到新增的 README 尚未被追蹤，因為它被列在輸出訊息的 Untracked files 下方。除非讀者明確指定要將該檔案加入提交的快照，Git不會主動將它加入。這樣就不會突然地將一些二進位格式的檔案或其它讀者並不想加入的檔案含入。讀者的確是要新增 README 檔案，因此讓我們開始追蹤該檔案。

追蹤新檔案

要追蹤新增的檔案，讀者可使用 git add 命令。欲追蹤 README 檔案，讀者可執行：

```
$ git add README
```

若讀者再度檢查目前狀態，可看到 README 檔案已被列入追蹤並且已被暫存：

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

    new file:   README
```

因為它被放在Changes to be committed文字下方，讀者可得知它已被暫存起來。若讀者此時提交更新，剛才執行 git add 加進來的檔案就會被記錄在歷史的快照。讀者可能可回想一下先前執行 git init 後也有執行過 git add，開始追蹤目錄內的檔案。git add 命令可接受檔名或者目錄名。若是目錄名，會遞迴將整個目錄下所有檔案及子目錄都加進來。

暫存已修改檔案

讓我們修改已被追蹤的檔案。若讀者修改先前已被追蹤的檔案，名為 `benchmarks.rb`，並檢查目前儲存庫的狀態。讀者會看到類似以下文字：

```
$ git status
On branch 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:  benchmarks.rb
```

`benchmarks.rb` 檔案出現在 “Changes not staged for commit” 下方，代表著這個檔案已被追蹤，而且位於工作目錄的該檔案已被修改，但尚未暫存。要暫存該檔案，可執行 `git add` 命令（這是一個多重用途的指令）。現在，讀者使用 `git add` 將 `benchmarks.rb` 檔案暫存起來，並再度執行 `git status`：

```
$ git add benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:  benchmarks.rb
```

這兩個檔案目前都被暫存起來，而且會進入下一次的提交。假設讀者記得仍需要對 `benchmarks.rb` 做一點修改後才要提交，可再度開啟並編輯該檔案。然而，當我們再度執行 `git status`：

```
$ vim benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:  benchmarks.rb

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:  benchmarks.rb
```

到底發生了什麼事？現在 `benchmarks.rb` 同時被列在已被暫存及未被暫存。這怎麼可能？這表示Git的確在讀者執行 `git add` 命令後，將檔案暫存起來。若讀者現在提交更新，最近一次執行 `git add` 命令時暫存的 `benchmarks.rb` 會被提交。若讀者在 `git add` 後修改檔案，需要再度執行 `git add` 將最新版的檔案暫存起來：

```
$ git add benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:  benchmarks.rb
```

忽略某些檔案

通常讀者會有一類不想讓Git自動新增，也不希望它們被列入未被追蹤的檔案。這些通常是自動產生的檔案，例如：記錄檔或者編譯系統產生的檔案。在這情況下，讀者可建立一個名為 `.gitignore` 檔案，列出符合這些檔案檔名的特徵。以下是一個範例：

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

第一列告訴Git忽略任何檔名為 `.o` 或 `.a` 結尾的檔案，它們是可能是編譯系統建置讀者的程式碼時產生的目的檔及程式庫。第二列告訴Git忽略所有檔名為`~`結尾的檔案，通常被很多文書編輯器，如：`Emacs`，使用的暫存檔案。讀者可能會想一併將`log`、`tmp`、`pid`目錄及自動產生的文件等也一併加進來。依據類推。在讀者要開始開發之前將 `.gitignore` 設定好，通常是一個不錯的點子。這樣子讀者不會意外地將真的不想追蹤的檔案提交到Git儲存庫。

編寫 `.gitignore` 檔案的規則如下：

- 空白列或者以#開頭的列會被忽略。
- 可使用標準的Glob pattern。
- 可以/結尾，代表是目錄。
- 可使用!符號將特徵反過來使用。

Glob pattern就像是shell使用的簡化版正規運算式。星號（ * ）匹配零個或多個字元； [abc] 匹配中括弧內的任一字元（此例為 a 、 b 、 c ）；問號（ ? ）匹配單一個字元；中括弧內的字以連字符連接（如： [0-9] ），用來匹配任何符合該範圍的字（此例為0到9）。

以下是另一個 `.gitignore` 的範例檔案：

```
# 註解，會被忽略。
# 不要追蹤檔名為 .a 結尾的檔案
*.a
# 但是要追蹤 lib.a，即使上方已指定忽略所有的 .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
# ignore all .txt files in the doc/ directory
doc/**/*.*
```

A `**/` pattern is available in Git since version 1.8.2.

檢視已暫存及尚未暫存的更動

若 `git status` 命令仍無法清楚告訴讀者想要的資訊（讀者想知道的是更動了哪些內容，而不是哪些檔案）。可使用 `git diff` 命令。稍後我們會更詳盡講解該命令。讀者使用它時通常會是為了瞭解兩個問題：目前已做的修改但尚未暫存的內容是哪些？以及將被提交的暫存資料有哪些？雖然 `git status` 一般來說即可回答這些問題。`git diff` 可精確的顯示哪些列被加入或刪除，以修補檔方式表達。

假設讀者編輯並暫存 `README`，接者修改 `benchmarks.rb` 檔案，卻未暫存。若讀者檢視目前的狀況，會看到類似下方文字：

```
$ git status
On branch 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:  benchmarks.rb
```

想瞭解尚未暫存的修改，執行 `git diff`，不用帶任何參數：

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
      @commit.parents[0].parents[0].parents[0]
    end

+
+    run_code(x, 'commits 1') do
+      git.commits.size
+    end
+
+    run_code(x, 'commits 2') do
+      log = git.commits('master', 15)
+      log.size
```

這命令比對目前工作目錄及暫存區域後告訴讀者哪些變更尚未被暫存。

若讀者想知道將被提交的暫存資料，使用 `git diff --cached`（在Git 1.6.1及更新版本，也可以使用較易記憶的 `git diff --staged` 命令）。這命令比對暫存區域及最後一個提交。

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

很重要的一點是 `git diff` 不會顯示最後一次commit後的所有變更；只會顯示尚未暫存的變更。這一點可能會混淆，若讀者已暫存所有的變更，`git diff` 不會顯示任何資訊。

舉其它例子，若讀者暫存 `benchmarks.rb` 檔案後又編輯，可使用 `git diff` 看已暫存的版本與工作目錄內版本尚未暫存的變更：

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified: benchmarks.rb

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: benchmarks.rb
```

現在讀者可使用 `git diff` 檢視哪些部份尚未被暫存：

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
main()

##pp Grit:::GitRuby.cache_client.stats
## test line
```

以及使用 `git diff --cached` 檢視目前已暫存的變更：

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
end

+
+    run_code(x, 'commits 1') do
+        git.commits.size
+    end
+
+    run_code(x, 'commits 2') do
+        log = git.commits('master', 15)
+        log.size
```

提交修改

現在讀者的暫存區域已被更新為讀者想畏的，可開始提交變更的部份。要記得任何尚未被暫存的新建檔案或已被修改但尚未使用 `git add` 暫存的檔案將不會被記錄在本次的提交中。它們仍會以被修改的檔案的身份存在磁碟中。在這情況下，最後一次執行 `git status`，讀者會看到所有已被暫存的檔案，讀者也準備好要提交修改。最簡單的提交是執行 `git commit`：

```
$ git commit
```

執行此命令會叫出讀者指定的編輯器。（由讀者 shell 的 `$EDITOR` 環境變數指定，通常是 `vim` 或 `emacs`。讀者也可以如同第 1 章介紹的，使用 `git config --global core.editor` 命令指定）

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
# Changes to be committed:
#       new file:   README
#       modified:  benchmarks.rb
#
~  
~  
~  
".git/COMMIT_EDITMSG" 10L, 283C
```

讀者可看到預設的提交訊息包含最近一次 `git status` 的輸出以註解方式呈現，以及螢幕最上方有一列空白列。讀者可移除這些註解後再輸入提交的訊息，或者保留它們，提醒你現在正在進行提交。（若想知道更動的內容，可傳遞 `-v` 參數給 `git commit`。如此一來連比對的結果也會一併顯示在編輯器內，方便讀者明確看到有什麼變更。）當讀者離開編輯器，Git 會利用這些提交訊息產生新的提交（註解及比對的結果會先被濾除）。

另一種方式則是在 `commit` 命令後方以 `-m` 參數指定提交訊息，如下：

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Fix benchmarks for speed
 2 files changed, 3 insertions(+)
 create mode 100644 README
```

現在讀者已建立第一個提交！讀者可從輸出的訊息看到此提交、放到哪個分支（`master`）、SHA-1查核碼（`463dc4f`）、有多少檔案被更動，以及統計此提交有多少列被新增及移除。

記得提交記錄讀者放在暫存區的快照。任何讀者未暫存的仍然保持在已被修改狀態；讀者可進行其它的提交，將它增加到歷史。每一次讀者執行提供，都是記錄專案的快照，而且以後可用來比對或者復原。

跳過暫存區域

雖然優秀好用的暫存區域能很有技巧且精確的提交讀者想記錄的資訊，有時候暫存區域也比讀者實際需要的工作流程繁瑣。若讀者想跳過暫存區域，Git 提供了簡易的使用方式。在 `git commit` 命令後方加上 `-a` 參數，Git 自動將所有已被追蹤且被修改的檔案送到暫存區域並開始提交程序，讓讀者略過 `git add` 的步驟：

```
$ git status
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:   benchmarks.rb

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 files changed, 5 insertions(+)
```

留意本次的提交之前，讀者並不需要執行 `git add` 將 `benchmarks.rb` 檔案加入。

刪除檔案

要從Git刪除檔案，讀者需要將它從已被追蹤檔案中移除（更精確的來說，是從暫存區域移除），並且提交。 `git rm` 命令除了完成此工作外，也會將該檔案從工作目錄移除。因此讀者以後不會在未被追蹤檔案列表看到它。

若讀者僅僅是將檔案從工作目錄移除，那麼在 `git status` 的輸出，可看見該檔案將會被視為“已被變更且尚未被更新”（也就是尚未存到暫存區域）：

```
$ rm grit.gemspec
$ git status
On branch 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:    grit.gemspec

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

接著，若執行 `git rm`，則會將暫存區域內的該檔案移除：

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    grit.gemspec
```

下一次提交時，該檔案將會消失而且不再被追蹤。若已更動過該檔案且將它記錄到暫存區域。必須使用 `-f` 參數才能將它強制移除。這是為了避免已被記錄的快照意外被移除且再也無法使用Git復原。

其它有用的技巧是保留工作目錄內的檔案，但從暫存區域移除。換句話說，或許讀者想在磁碟機上的檔案且不希望Git繼續追蹤它。這在讀者忘記將某些檔案記錄到 `.gitignore` 且不小心將它增加到暫存區域時特別有用。比如說：巨大的記錄檔、或大量在編譯時期產生的 `.a` 檔案。欲使用此功能，加上 `--cached` 參數：

```
$ git rm --cached readme.txt
```

除了檔名、目錄名以外，還可以指定簡化的正規運算式給 `git rm` 命令。這意謂著可執行類似下列指令：

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

注意倒斜線（\）前方的星號（*）。這是必須的，因為Git會在shell以上執行檔案的擴展。此命令移除log目錄下所有檔名以.log結尾的檔案。讀者也可以執行類似下列命令：

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

此命令移除所有檔名以~結尾的檔案。

搬動檔案

Git並不像其它檔案控制系統一樣，明確地追蹤檔案的移動。若將被Git追蹤的檔名更名，並沒有任何元數據儲存在Git中去標示此更名動作。然而Git能很聰明地指出這一點。稍後會介紹關於偵測檔案的搬動。

因此Git存在mv這個指令會造成一點混淆。若想要在Git中更名某個檔案，可執行以下命令：

```
$ git mv file_from file_to
```

而且這命令可正常工作。事實上，在執行完更名的動作後檢視一下狀態。可看到Git認為該檔案被更名：

```
$ git mv README.txt README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed: README.txt -> README
```

不過，這就相當於執行下列命令：

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

Git會在背後判斷檔案是否被更名，因此不管是用上述方法還是'mv'命令都沒有差別。實際上唯一不同的是'mv'是一個命令，而不是三個。使用上較方便。更重要的是讀者可使用任何慣用的工具更名，再使用add/rm，接著才提交。

檢視提交的歷史記錄

在提交數個更新，或者複製已有一些歷史記錄的儲存庫後。或許會想希望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。

以下採用非常簡單，名為 `simplegit` 的專案做展示。欲取得此專案，執行以下命令：

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

在此專案目錄內執行 `git log`，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 code

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

    first commit
```

在未加任何參數情況下，`git log` 以新到舊的順序列出儲存庫的提交的歷史記錄。也就是說最新的更新會先被列出來。同時也會列出每個更新的 SHA-1 查核值、作者大名及電子郵件地址、及提交時輸入的訊息。

`git log` 命令有很多樣化的選項，供讀者精確指出想搜尋的結果。接下來會介紹一些常用的選項。

最常用的選項之一為 `-p`，用來顯示每個更新之間差別的內容。另外還可以加上 `-2` 參數，限制為只輸出最後兩個更新。

```
$ 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
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      = "simplegit"
-    s.version   = "0.1.0"
+    s.version   = "0.1.1"
     s.author    = "Scott Chacon"
     s.email     = "schacon@gee-mail.com"

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

    removed unnecessary test code

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外，還另外附上每個更新的差異。這對於重新檢視或者快速的瀏覽協同工作伙伴新增的更新非常有幫助。

有時候用 **word level** 的方式比 **line level** 更容易看懂變化。在 `git log -p` 後面附加 `--word-diff` 選項，就可以取代預設的 **line level** 模式。當你在看原始碼的時候 **word level** 還挺有用的，還有一些大型文字檔，如書籍或論文就派上用場了，範例如下：

```
$ git log -U1 --word-diff
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
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      = "simplegit"
    s.version   = ["0.1.0"] { +"0.1.1" +}
    s.author    = "Scott Chacon"
```

如你所見，輸出範例中沒有列出新增與刪除的行，變動的地方用內嵌的方式顯示，你可以看到新增的字被包括在 `{+ +}` 內，而刪除的則包括在 `{- -}` 內，如果你想再減少顯示的資訊，將上述的三行再減少到只顯示變動的那行。你可以用 `-U1` 選項，就像上述的範例中那樣。

另外也可以使用 `git log` 提供的一系統摘要選項。例如：若想檢視每個更新的簡略統計資訊，可使用 `--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 code

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(+)
```

如以上所示，`--stat` 選項在每個更新項目的下方列出被更動的檔案、有多少檔案被更動，以及有多行列被加入或移出該檔案。也會在最後印出摘要的訊息。其它實用的選項是 `--pretty`。這個選項改變原本預設輸出的格式。有數個內建的選項供讀者選用。其中 `oneline` 選項將每一個更新印到單獨一行，對於檢視很多更新時很有用。更進一步，`short`、`full`、`fuller` 選項輸出的格式大致相同，但會少一些或者多一些資訊。

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

最有趣的選項是 `format`，允許讀者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, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

表格2-1列出一些 `format` 支援的選項。

選項	選項的說明
%H	該更新的SHA1雜湊值
%h	該更新的簡短SHA1雜湊值
%T	存放該更新的根目錄的Tree物件的SHA1雜湊值
%t	存放該更新的根目錄的Tree物件的簡短SHA1雜湊值
%P	該更新的父更新的SHA1雜湊值
%p	該更新的父更新的簡短SHA1雜湊值
%an	作者名字
%ae	作者電子郵件
%ad	作者的日期 (格式依據 date 選項而不同)
%ar	相對於目前時間的作者的日期
%cn	提交者的名字
%ce	提交者的電子郵件
%cd	提交的日期
%cr	相對於目前時間的提交的日期
%s	標題

讀者可能會好奇作者與提交者之間的差別。作者是完成該工作的人，而提交者則是最後將該工作提交出來的人。因此，若讀者將某個專案的修補檔送出，而且該專案的核心成員中一員套用該更新，則讀者與該成員皆會被列入該更新。讀者即作者，而該成員則是提交者。在第五章會提到較多之間的差別。

`oneline` 及 `format` 選項對於另一個名為 `--graph` 的選項特別有用。該選項以 ASCII 畫出分支的分歧及合併的歷史。可參考我們的 Grit 的儲存庫：

```
$ 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 xmlschema
* 11d191e Merge branch 'defunkt' into local
```

這些只是一些簡單的 `git log` 的選項，還有許多其它的。表格2-2列出目前我們涵蓋的及一些可能有用的格式選項，以及它們如何更動 `log` 命令的輸出格式。

選項	選項的說明
<code>-p</code>	顯示每個更新與上一個的差異。
<code>--word-diff</code>	使用 word diff 格式顯示 patch 內容。
<code>--stat</code>	顯示每個更新更動的檔案的統計及摘要資訊。
<code>--shortstat</code>	僅顯示 <code>--stat</code> 提供的訊息中關於更動、插入、刪除的文字。
<code>--name-only</code>	在更新的訊息後方顯示更動的檔案列表。
<code>--name-status</code>	顯示新增、更動、刪除的檔案列表。
<code>--abbrev-commit</code>	僅顯示SHA1查核值的前幾位數，而不是顯示全部的40位數。
<code>--relative-date</code>	以相對於目前時間方式顯示日期（例如：“2 weeks ago”），而不是完整的日期格式。
<code>--graph</code>	以 ASCII 在 log 輸出旁邊畫出分支的分歧及合併。
<code>--pretty</code>	以其它格式顯示更新。可用的選項包含 <code>oneline</code> 、 <code>short</code> 、 <code>full</code> 、 <code>fuller</code> 及可自訂格式的 <code>format</code> 。
<code>--oneline</code>	<code>--pretty=oneline --abbrev-commit</code> 的簡短用法。

限制 `log` 的輸出範圍

除了輸出格式的選項，`git log` 也接受一些好用的選項。也就是指定只顯示某一個子集合的更新。先前已介紹過僅顯示最後兩筆更新的 `-2` 選項。實際上可指定 `-<n>`，而 `n` 是任何整數，用來顯示最後的 `n` 個更新。不過讀者可能不太會常用此選項，因為 Git 預設將所有的輸出導到分頁程式，故一次只會看到一頁。

然而，像 `--since` 及 `--until` 限制時間的選項就很有用。例如，以下命令列出最近兩週的更新：

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

此命令支援多種格式。可指定特定日期（如：“2008-01-15”）或相對的日期，如：“2 years 1 day 3 minutes ago”。

使用者也可以過濾出符合某些搜尋條件的更新。`--author` 選項允許使用者過濾出特定作者，而 `--grep` 選項允許以關鍵字搜尋提交的訊息。（注意：若希望同時符合作者名字及字串比對，需要再加上 `--all-match`；否則預設為列出符合任一條件的更新）

最後一個有用的選項是過濾路徑。若指定目錄或檔案名稱，可僅印出更動到這些檔案的更新。這選項永遠放在最後，而且一般來說會在前方加上 `--` 以資區別。

在表格2-3，我們列出這些選項以及少數其它常見選項以供參考。

選項	選項的說明文字
<code>-(n)</code>	僅顯示最後 n 個更新
<code>--since, --after</code>	列出特定日期後的更新。
<code>--until, --before</code>	列出特定日期前的更新。
<code>--author</code>	列出作者名稱符合指定字串的更新。
<code>--committer</code>	列出提交者名稱符合指定字串的更新。

例如：若想檢視 Git 的原始碼中，Junio Hamano 在 2008 年十月提交且不是合併用的更新。可執行以下命令：

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
--before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Git 原始碼的更新歷史接近二萬筆更新，本命令顯示符合條件的六筆更新。

使用圖形界面檢視歷史

若讀者較偏向使用圖形界面檢視歷史，或與會想看一下隨著 Git 發佈的，名為 gitk 的 Tcl/Tk 程式。Gitk 基本上就是 git log 的圖形界面版本，而且幾乎接受所有 git log 支援的過濾用選項。若在專案所在目錄下執行 gitk 命令，將會看到如圖2-2的畫面。

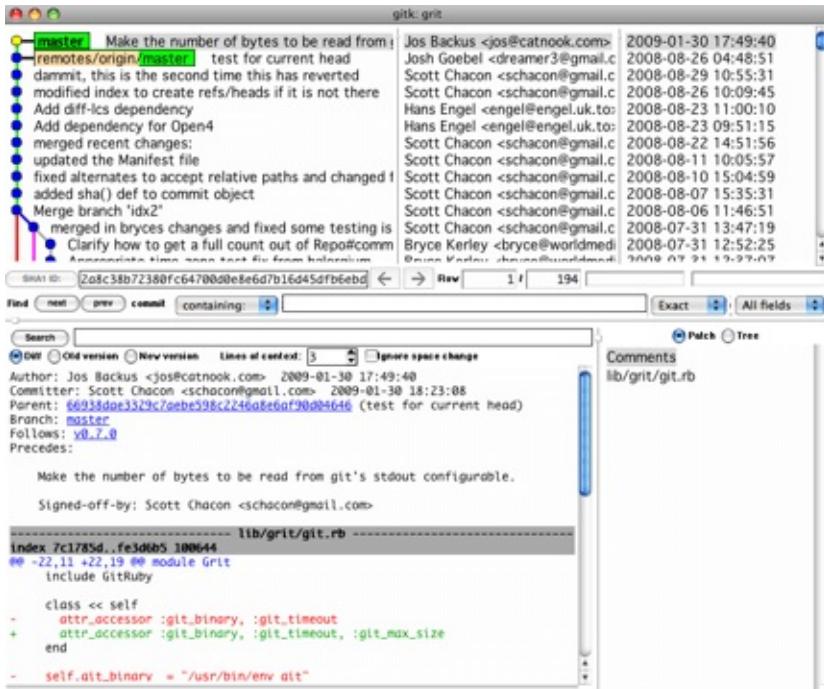


圖2-2。gitk檢視歷史程式。

在上圖中可看到視窗的上半部顯示相當棒的更新歷史圖。視窗下半部則顯示當時被點選的更新引入的變更。

復原

在任何時間點，或許讀者會想要復原一些事情。接下來我們會介紹一些基本的復原方式。但要注意，由於有些復原動作所做的變更無法再被還原。這是少數在使用 Git 時，執行錯誤的動作會遺失資料的情況。

更動最後一筆更新

最常見的復原發生在太早提交更新，也許忘了加入某些檔案、或者搞砸了提交的訊息。若想要試著重新提交，可試著加上 `--amend` 選項：

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

此命令取出暫存區資料並用來做本次的提交。只要在最後一次提交後沒有做過任何修改（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，唯一有更動的是提交時的訊息。

同一個文書編輯器被帶出來，並且已包含先前提交的更新內的訊息。讀者可像往常一樣編輯這些訊息，差別在於它們會覆蓋上一次提交。

如下例，若提交了更新後發現忘了一併提交某些檔案，可執行最後一個命令：

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

這些命令的僅僅會提交一個更新，第二個被提交的更新會取代第一個。

取消已被暫存的檔案

接下來兩節展示如何應付暫存區及工作目錄的復原。用來判斷這兩個區域狀態的命令也以相當好的方式提示如何復原。比如說已經修改兩個檔案，並想要以兩個不同的更新提交它們，不過不小心執行 `git add *` 將它們同時都加入暫存區。應該如何將其中一個移出暫存區？

`git status` 命令已附上相關的提示：

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

    modified:   README.txt
    modified:   benchmarks.rb
```

在“Changes to be committed”文字下方，註明著使用“`git reset HEAD <file>...`”，將 file 移出暫存區”。因此，讓我們依循該建議將 `benchmarks.rb` 檔案移出暫存區：

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

    modified:   README.txt

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:   benchmarks.rb
```

這個命令看起來有點奇怪，不過它的確可行。`benchmarks.rb` 檔案被移出暫存區了。

復原已被更動的檔案

若讀者發現其者並不需要保留 `benchmarks.rb` 檔案被更動部份，應該如何做才能很容易的復原為最後一次提交的狀態（或者最被複製儲存庫時、或放到工作目錄時的版本）？很幸運的，`git status` 同樣也告訴讀者如何做。在最近一次檢視狀態時，暫存區看起來應如下所示：

```
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:   benchmarks.rb
```

在這訊息中已很明確的說明如何拋棄所做的修改（至少需升級為 Git 1.6.1 或更新版本。若讀者使用的是舊版，強烈建議升級，以取得更好用的功能。）讓我們依據命令執行：

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

    modified:   README.txt
```

在上述文字可看到該變更已被復原。讀者應該瞭解這是危險的命令，任何對該檔案做的修改將不復存在，就好像複製別的檔案將它覆蓋。除非很清楚真的不需要該檔案，絕不要使用此檔案。若需要將這些修改排除，我們在下一章節會介紹備份及分支。一般來說會比此方法來的好。

切記，任何在 Git 提交的更新幾乎都是可復原的。即使是分支中的更新被刪除或被 `--amend` 覆寫，皆能被復原。（參考第九章關於資料的復原）然而，未被提交的則幾乎無法救回。

與遠端協同工作

想要在任何 Git 控管的專案協同作業，需要瞭解如何管理遠端的儲存庫。遠端儲存庫是置放在網際網路或網路其它地方中的專案版本。讀者可設定多個遠端儲存庫，具備唯讀或可讀寫的權限。與他人協同作業時，需要管理這些遠端儲存庫，並在需要分享工作時上傳或下載資料。管理遠端儲存庫包含瞭解如何新增遠端儲存庫、移除已失效的儲存庫、管理許多分支及定義是否要追蹤它們等等。本節包含如何遠端管理的技巧。

顯示所有的遠端儲存庫

欲瞭解目前已加進來的遠端儲存庫，可執行 `git remote` 命令。它會列出當初加入遠端儲存庫時指定的名稱。若目前所在儲存庫是從其它儲存庫複製過來的，至少應該看到 `origin`，也就是 Git 複製儲存庫時預設名字：

```
$ git clone git://github.com/schacon/ticgit.git
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 | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

也可以再加上 `-v` 參數，將會在名稱後方顯示其 URL：

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

若有一個以上遠端儲存庫，此命令會全部列出。例如：我的 Grit 儲存庫包含以下遠端儲存庫。

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

這意謂著我們可很容易從這些伙伴的儲存庫取得最新的更新。要留意的是只有 origin 遠端的 URL 是 SSH。因此它是唯一我們能上傳的遠端的儲存庫。（關於這部份將在第四章介紹）

新增遠端儲存庫

在先前章節已提到並示範如何新增遠端儲存庫，這邊會很明確的說明如何做這項工作。欲新增遠端儲存庫並取一個簡短的名字，執行 `git remote add [shortname] [url]`：

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb    git://github.com/paulboone/ticgit.git
```

現在可看到命令列中的 `pb` 字串取代了整個 URL。例如，若想取得 Paul 上傳的且本地端儲存庫沒有的更新，可執行 `git fetch pb`：

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master      -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

現在可在本地端使用 `pb/master` 存取 Paul 的 `master` 分支。讀者可將它合併到本地端任一分支、或者建立一個本地端的分支指向它，如果讀者想監看它。

從遠端儲存庫擷取或合併

如剛才所示，欲從遠端擷取資料，可執行：

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

此命令到該遠端專案將所有本地端沒有的資料拉下來。在執行此動作後，讀者應該有參考到該遠端專案所有分支的參考點，可在任何時間點用來合併或監看。（在第三章將會提及更多關於如何使用分支的細節）

若複製了一個儲存庫，會自動將該遠端儲存庫命令為 `origin`。因此 `git fetch origin` 取出所有在複製或最後一下擷取後被上傳到該儲存庫的更新。需留意的是 `fetch` 命令僅僅將資料拉到本地端的儲存庫，並未自動將它合併進來，也沒有修改任何目前工作的項目。讀者得在必要時將它們手動合併進來。

若讀者設定一個會追蹤遠端分支的分支（參考下一節及第三章，取得更多資料），可使用 `git pull` 命令自動擷取及合併遠端分支到目錄的分支。這對讀者來說或許是較合適的工作流程。而且 `git clone` 命令預設情況下會自動設定本地端的 `master` 分支追蹤被複製的遠端儲存庫的 `master` 分支。（假設該儲存庫有 `master` 分支）執行 `git pull` 一般來說會從當初複製時的來源儲存庫擷取資料並自動試著合併到目前工作的版本。

上傳到遠端儲存庫

當讀者有想分享出去的專案，可將更新上傳到上游。執行此動作的命令很簡單：`git push [remote-name] [branch-name]`。若想要上傳 `master` 分支到 `origin` 伺服器（再說一次，複製時通常自動設定此名字），接著執行以下命令即可上傳到伺服器：

```
$ git push origin master
```

此命令只有在被複製的伺服器開放寫入權限給使用者，而且同一時間內沒有其它人在上傳。若讀者在其它同樣複製該伺服器的使用者上傳一些更新後上傳到上游，該上傳動作將會被拒絕。讀者必須先將其它使用者上傳的資料拉下來並整合進來後才能上傳。參考第三章瞭解如何上傳到遠端儲存庫的細節。

監看遠端儲存庫

若讀者想取得遠端儲存庫某部份更詳盡的資料，可執行 `git remote show [remote-name]`。若執行此命令時加上特定的遠端名字，比如說：`origin`。會看到類似以下輸出：

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

它將同時列出遠端儲存庫的URL位置和追蹤分支資訊。特別是告訴你如果你在 `master` 分支時用 `git pull` 時，會去自動抓取數據合併到本地的 `master` 分支。它也列出所有曾經被抓取過的遠端分支。

當你使用 Git 更頻繁之後，你或許會想利用 `git remote show` 去看到更多的資訊。

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

這個指令顯示當你執行 `git push` 會自動推送的哪個分支(最後兩行)。它也顯示哪些遠端分支還沒被同步到本地端(在這個例子是caching)，哪些已同步到本地的遠端分支在遠端已被刪除(libwalker和walker2)，以及當執行 `git pull` 時會自動被合併的分支。

移除或更名遠端儲存庫

在新版 Git 中可以用 `git remote rename` 命令修改某個遠端儲存庫在本地的簡稱，舉例而言，想把 `pb` 改成 `paul`，可以執行下列指令：

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

值得留意的是這也改變了遠端分支的名稱，原來的 `pb/master` 分支現在變成 `paul/master`。

當你為了種種原因想要移除某個遠端，像是換伺服器或是已不再使用某個特別的鏡像，又或是某個貢獻者已不再貢獻時。你可以使用 `git remote rm`：

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

標籤

就像大多數的版本管理系統，Git具備在特定時間點加入標籤去註明其重要性的功能。一般而言，我們會使用這個功能去標記出發行版本(如V1.0等等)。這個小節中，你將會學到如何列出既有的標籤、建立新標籤以及各種不同標籤間的差異。

列出標籤

在Git中列出既有的標籤是非常簡單的。直接輸入 `git tag`：

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

這個指令將以字母順序列出標籤；所以這個順序並不代表其重要性。

你也可以用特定的字串規則去搜尋標籤。以Git本身的儲存庫為例，其中包含超過240個標籤。當你只對1.4.2感興趣時，你可以執行以下指令：

```
$ git tag -l 'v1.4.2.*'  
v1.4.2.1  
v1.4.2.2  
v1.4.2.3  
v1.4.2.4
```

建立標簽

Git使用兩大類的標籤：輕量級(*lightweight*)和含附註(*annotated*)。輕量級標籤就像是沒有更動的分支，實際上它僅是指到特定commit的指標。然而，含附註的標籤則是實際存在Git資料庫上的完整物件。它具備檢查碼、e-mail和日期，也包含標籤訊息，並可以被GNU Privacy Guard (GPG)簽署和驗證。一般而言，我們都建議使用含附註的標籤以便保留相關訊息；但如果只是臨時加註標籤或不需要保留其他訊息，就是使用輕量級標籤的時機。

含附註的標籤

建立一個含附註的標籤很簡單。最容易的方法是加入 `-a` 到 `tag` 指令上：

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

而 `-m` 選項用來設定標籤訊息。如果你沒有設定該訊息，Git 會啓動文字編輯器讓你輸入。

透過 `git show` 可看到指定標籤的資料與對應的 commit。

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

在列出 commit 資訊前，我們可以看到這個標籤的設定者資訊，下標籤時間與附註訊息。

簽署標籤

假設你有私鑰(private key)，你也可以用 GPG 簽署在標籤上。只要用 `-s` 取代 `-a`：

```
$ 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@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

再對這個標籤執行 `git show`，你就能看到你的 GPG 簽章已經附在裡面：

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgjSN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAj82/
=wryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

稍後你會看到如何驗證已簽署的標籤。

輕量級的標簽

另一種則是輕量級的標簽。基本上就是只保存commit檢查碼的文件。要建立這樣的標籤，不必下任何選項，直接設定標籤名稱即可。

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

這樣一來，當執行 `git show` 查看這個標籤時，你不會看到其他標籤資訊，只會顯示對應的 commit：

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

驗證標籤

想要驗證已簽署的標籤，需要使用 `git tag -v [tag-name]`。這個指令透過GPG去驗證簽章。而且在你的keyring中需要有簽署者的公鑰才能進行驗證：

```
$ 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
```

如果沒有簽署者的公鑰，則會看到下列訊息：

```
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'
```

追加標籤

你也可以對過去的commit上加入標籤。假設你的commit歷史如下：

```
$ 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
9fcceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

如果你之前忘了將"updated rakefile"這個commit加入v1.2標籤。仍然可在事後設定。要完成這個動作，你必須加入該次commit的檢查碼(或前幾碼即可)到以下指令：

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

你可以看到標籤已經補上：

```
$ 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 9fce02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

updated rakefile
...
```

分享標籤

在預設的情況下，`git push` 指令並不會將標籤傳到遠端伺服器上。當建立新標籤後，你必須特別下指令才會將它推送到遠端儲存庫上。類似將分支推送到遠端的過程，透過 `git push origin [tagname]` 指令。

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]           v1.5 -> v1.5
```

如果你有很多標籤需要一次推送上去，你也可以加入 `--tags` 選項到 `git push` 指令中。這將會傳送所有尚未在遠端伺服器上的標籤。

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]      v0.1 -> v0.1
 * [new tag]      v1.2 -> v1.2
 * [new tag]      v1.4 -> v1.4
 * [new tag]      v1.4-lw -> v1.4-lw
 * [new tag]      v1.5 -> v1.5
```

現在，當其他使用者clone或pull你的儲存庫時，他們也同時會取得所有你的標籤。

提示和技巧

在結束Git基礎這個章節前，我們將介紹有一些將會使你的Git使用經驗更簡單、方便和親切的提示和技巧。或許很多人從未運用過這些技巧，我們也不會假設你在本書的後續章節會使用它們。但你也許會想知道如何使用它們。

自動補齊

如果你用的是 Bash shell，你可以啓動Git本身寫好的自動補齊腳本。下載Git原始碼，切到 contrib/completion 目錄；可以看到檔案名為 git-completion.bash。將它複製到你的家目錄，並加入以下指令到你的 .bashrc 檔案裡：

```
source ~/git-completion.bash
```

如果你想為所有使用者都自動設置Bash shell的補齊功能，在Mac系統上將這個腳本複製到 /opt/local/etc/bash_completion.d 目錄，若你使用Linux系統複製到 /etc/bash_completion.d/ 目錄。這兩個目錄中的腳本，都會在 Bash 啓動時自動載入。

如果你在Windows使用Git Bash，也就是利用Windows with msysGit安裝Git，自動補齊功能已預先設定好，可以直接使用。

在你輸入Git指令時，只要按下Tab鍵，便會列出所有合適的指令建議：

```
$ git co<tab><tab>
commit config
```

然後按下Tab鍵兩次，便會提示commit和config這些可用指令。當再輸入 m<tab> 便會自動補齊 git commit 。

指令的選項也可以自動補齊，這或許是更實用的功能。舉例而言，當你下 git log 指令時，若忘記該輸入哪個選項，只要輸入開頭字元然後按下Tab去看看可能的選項：

```
$ git log --s<tab>
--shortstat  --since=  --src-prefix=  --stat    --summary
```

這是個好用的小技巧，或許可以省下許多輸入和查文件的時間

Git 命令別名

如果僅輸入命令的部份字元，Git並不會幫你推論出你想要下的完整命令。如果你想偷懶，不想輸入Git命令的所有字元，你可以輕易地利用 `git config` 設定別名(alias)。你也許會想要設定以下這幾個範例：

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

這些例子顯示出，你可以只輸入 `git ci`，取代輸入 `git commit`。隨著你深入使用Git，將會發現某些命令用得頻繁；這時不妨建立新的別名提高使用效率。

利用這個技術將有助於創造出你認為應該存在的命令。舉例而言，為了提高取消暫存檔案的便利性，你可以加入以下命令：

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

這將使得下列兩個命令完全相等：

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

使用別名看起來更清楚。另外，加入 `last` 別名也是很常用的技巧：

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

如此一來，將可更簡單地看到最新的提交訊息：

```
$ 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>
```

你可以發現，Git只是簡單地在命令中替換你設定的別名。然而，你不僅希望執行Git的子命令，而想執行外部命令。在這個情形中，你可以加入 `!` 字元在所要執行的命令前。這將有助於設計運作於Git儲存庫的自製工具。這個範例藉由設定 `git visual` 別名去執行 `gitk`：

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


總結

至此，讀者已具備所有Git的本地端操作，包括：創建和副本儲存庫、建立修改、暫存和提交這些修改，以及檢視在儲存庫中所有修改歷史。接下來，我們將觸及Git的殺手級特性，也就是他的分支模型。

Git 分支

幾乎每一種版本控制系統都以某種形式支援分支。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同時繼續工作。在很多版本控制系統中，這是個昂貴的過程，常常需要創建一個原始程式碼目錄的完整副本，對大型項目來說會花費很長時間。

有人把 Git 的分支模型稱為“必殺技特性”，而正是因為它，將 Git 從版本控制系統家族裡區分出來。Git 有何特別之處呢？Git 的分支可謂是難以置信的羽量級，它的新建操作幾乎可以在瞬間完成，並且在不同分支間切換起來也差不多一樣快。和許多其他版本控制系統不同，Git 鼓勵在工作流程中頻繁使用分支與合併，哪怕一天之內進行許多次都沒有關係。理解分支的概念並熟練運用後，你才會意識到為什麼 Git 是一個如此強大而獨特的工具，並從此真正改變你的開發方式。

何謂分支

為了理解 Git 分支的實現方式，我們需要回顧一下 Git 是如何儲存資料的。或許你還記得第一章的內容，Git 保存的不是檔差異或者變化量，而只是一系列檔快照。

在 Git 中提交時，會保存一個提交（commit）物件，該物件包含一個指向暫存內容快照的指標，包含本次提交的作者等相關附屬資訊，包含零個或多個指向該提交物件的父物件指標：首次提交是沒有直接祖先的，普通提交有一個祖先，由兩個或多個分支合併產生的提交則有多個祖先。

為直觀起見，我們假設在工作目錄中有三個檔，準備將它們暫存後提交。暫存操作會對每一個檔計算校驗和（即第一章中提到的 SHA-1 雜湊字串），然後把當前版本的檔快照保存到 Git 倉庫中（Git 使用 blob 類型的物件存儲這些快照），並將校驗和加入暫存區域：

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

當使用 `git commit` 新建一個提交物件前，Git 會先計算每一個子目錄（本例中就是專案根目錄）的校驗和，然後在 Git 倉庫中將這些目錄保存為樹（tree）物件。之後 Git 創建的提交物件，除了包含相關提交資訊以外，還包含著指向這個樹物件（專案根目錄）的指標，如此它就可以在將來需要的時候，重現此次快照的內容了。

現在，Git 倉庫中有五個物件：三個表示檔快照內容的 blob 物件；一個記錄著目錄樹內容及其中各個檔對應 blob 物件索引的 tree 物件；以及一個包含指向 tree 物件（根目錄）的索引和其他提交資訊中繼資料的 commit 物件。概念上來說，倉庫中的各個物件保存的資料和相互關係看起來如圖 3-1 所示：

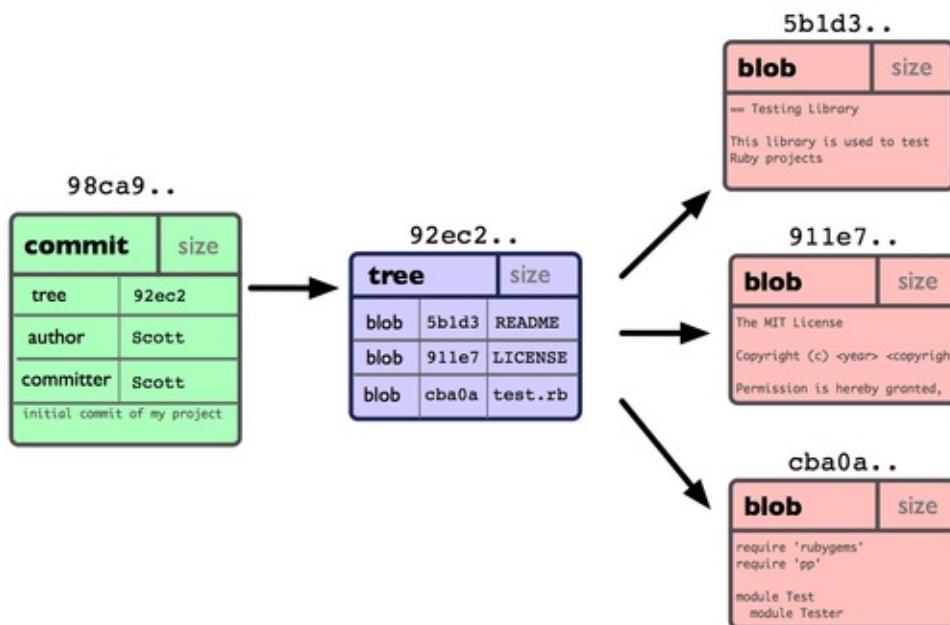


圖 3-1. 單個提交物件在倉庫中的資料結構

作些修改後再次提交，那麼這次的提交物件會包含一個指向上次提交物件的指標（譯注：即下圖中的 `parent` 物件）。兩次提交後，倉庫歷史會變成圖 3-2 的樣子：

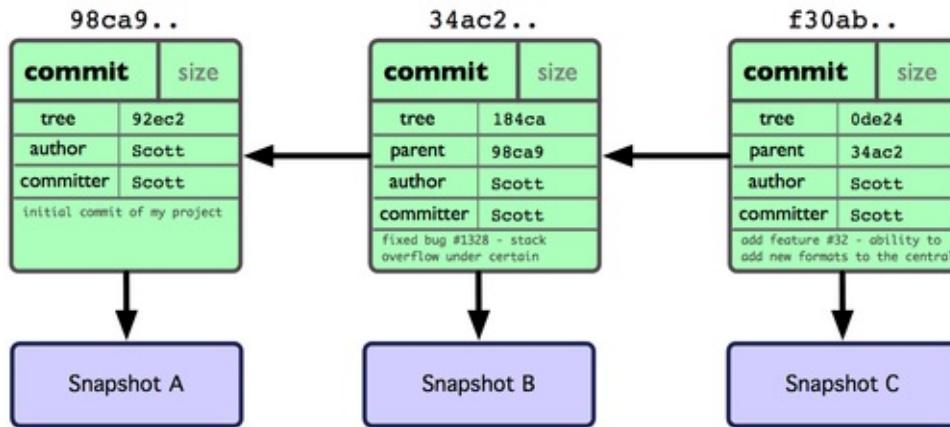


圖 3-2. 多個提交物件之間的連結關係

現在來談分支。Git 中的分支，其實本質上僅僅是個指向 `commit` 物件的可變指標。Git 會使用 `master` 作為分支的預設名字。在若干次提交後，你其實已經有了一個指向最後一次提交物件的 `master` 分支，它在每次提交的時候都會自動向前移動。

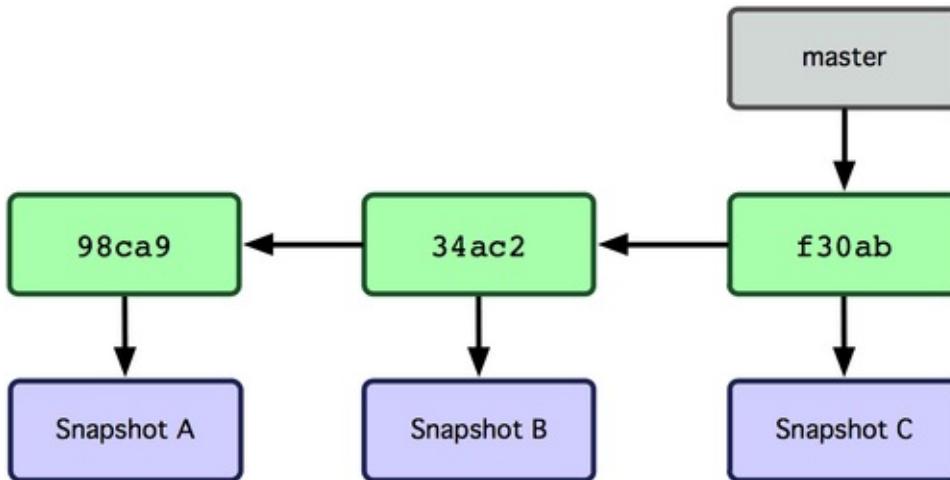


圖 3-3. 分支其實就是從某個提交物件往回看的歷史

那麼，Git 又是如何創建一個新的分支的呢？答案很簡單，創建一個新的分支指標。比如新建一個 `testing` 分支，可以使用 `git branch testing` 命令：

```
$ git branch testing
```

這會在當前 `commit` 物件上新建一個分支指標（見圖 3-4）。

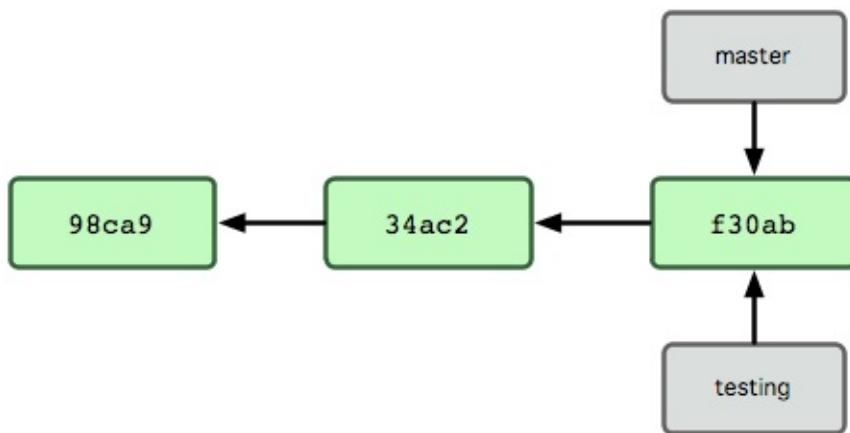


圖 3-4. 多個分支指向提交資料的歷史

那麼，Git 是如何知道你當前在哪個分支上工作的呢？其實答案也很簡單，它保存著一個名為 HEAD 的特別指標。請注意它和你熟知的許多其他版本控制系統（比如 Subversion 或 CVS）裡的 HEAD 概念大不相同。在 Git 中，它是一個指向你正在工作中的本地分支的指標（譯注：將 HEAD 想像為當前分支的別名。）。運行 `git branch` 命令，僅僅是建立了一個新的分支，但不會自動切換到這個分支中去，所以在這個例子中，我們依然還在 master 分支裡工作（參考圖 3-5）。

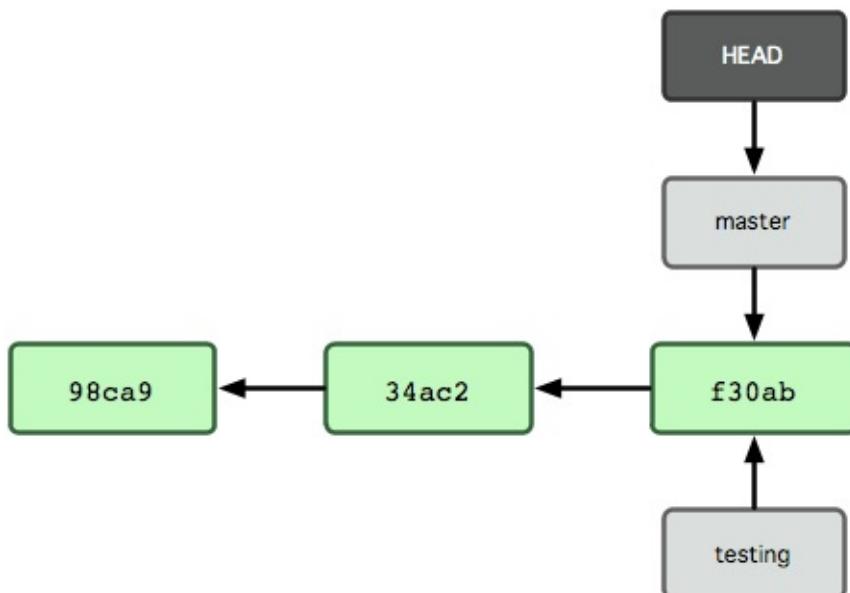


圖 3-5. HEAD 指向當前所在的分支

要切換到其他分支，可以執行 `git checkout` 命令。我們現在轉換到新建的 testing 分支：

```
$ git checkout testing
```

這樣 HEAD 就指向了 testing 分支（見圖 3-6）。

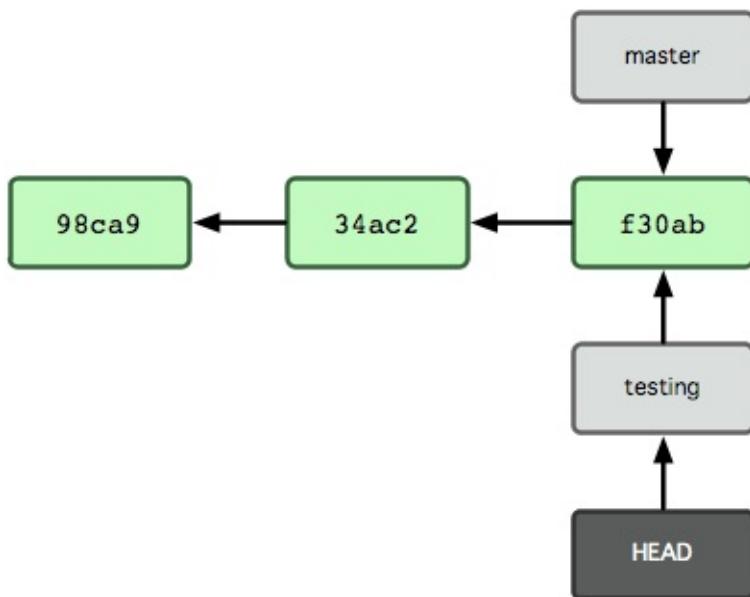


圖 3-6. HEAD 在你轉換分支時指向新的分支

這樣的實現方式會給我們帶來什麼好處呢？好吧，現在不妨再提交一次：

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

圖 3-7 展示了提交後的結果。

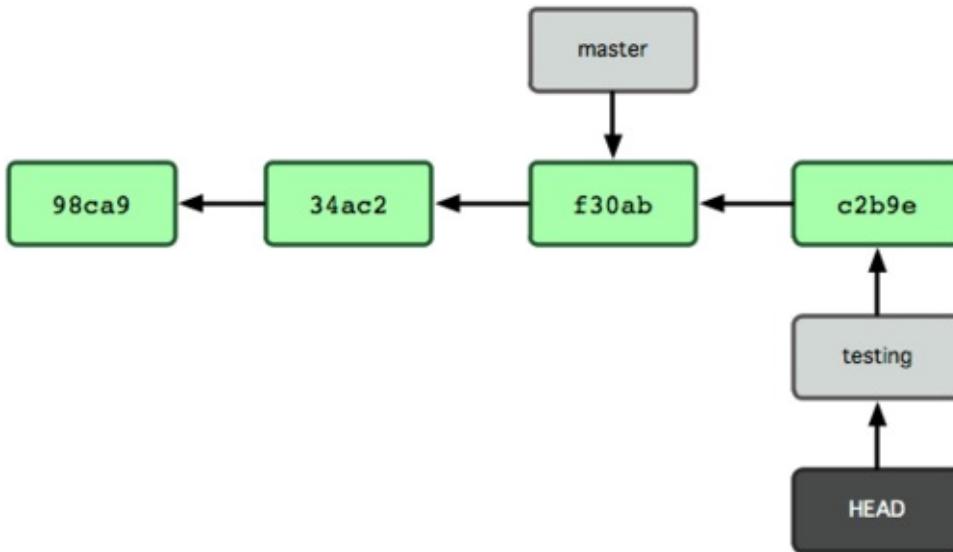


圖 3-7. 每次提交後 HEAD 隨著分支一起向前移動

非常有趣，現在 testing 分支向前移動了一格，而 master 分支仍然指向原先 `git checkout` 時所在的 commit 物件。現在我們回到 master 分支看看：

```
$ git checkout master
```

圖 3-8 顯示了結果。

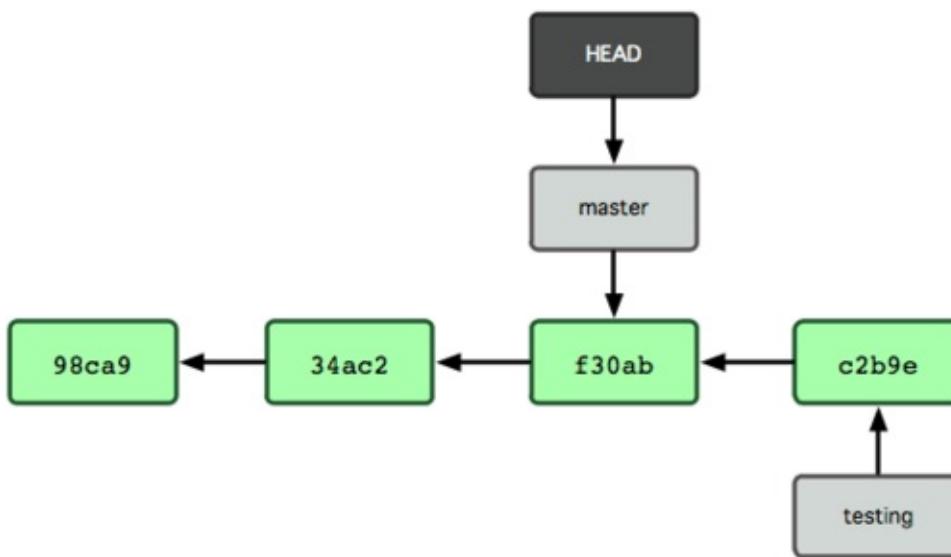


圖 3-8. HEAD 在一次 checkout 之後移動到了另一個分支

這條命令做了兩件事。它把 HEAD 指標移回到 master 分支，並把工作目錄中的檔換成了 master 分支所指向的快照內容。也就是說，現在開始所做的改動，將始於本專案中一個較老的版本。它的主要作用是將 testing 分支裡作出的修改暫時取消，這樣你就可以向另一個方向進行開發。

我們作些修改後再次提交：

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

現在我們的項目提交歷史產生了分叉（如圖 3-9 所示），因為剛才我們創建了一個分支，轉換到其中進行了一些工作，然後又回到原來的主分支進行了另外一些工作。這些改變分別孤立在不同的分支裡：我們可以在不同分支裡反復切換，並在時機成熟時把它們合併到一起。而所有這些工作，僅僅需要 `branch` 和 `checkout` 這兩條命令就可以完成。

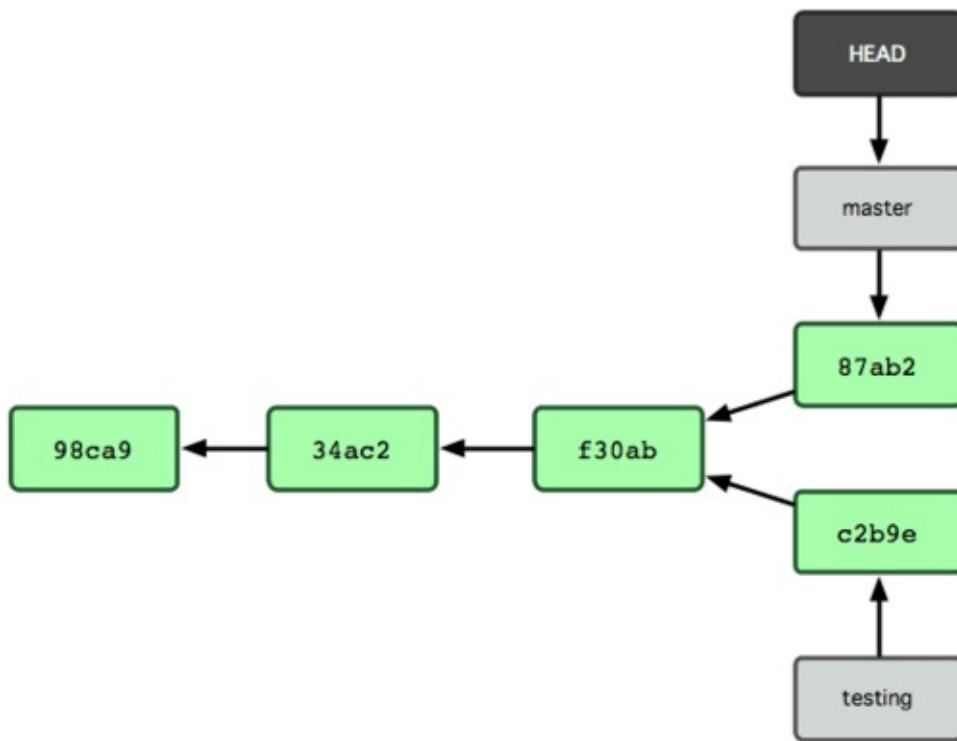


圖 3-9. 不同流向的分支歷史

由於 Git 中的分支實際上僅是一個包含所指物件校驗和（40 個字元長度 SHA-1 字串）的檔，所以創建和銷毀一個分支就變得非常廉價。說白了，新建一個分支就是向一個檔寫入 41 個位元組（外加一個分行符號）那麼簡單，當然也就很快了。

這和大多數版本控制系統形成了鮮明對比，它們管理分支大多採取備份所有專案檔案到特定目錄的方式，所以根據專案檔案數量和大小不同，可能花費的時間也會有相當大的差別，快則幾秒，慢則數分鐘。而 Git 的實現與項目複雜度無關，它永遠可以在幾毫秒的時間內完成分支的創建和切換。同時，因為每次提交時都記錄了祖先資訊（譯注：即 `parent` 物件），將來要合併分支時，尋找恰當的合併基礎（譯注：即共同祖先）的工作其實已經自然而然地擺在那裡了，所以實現起來非常容易。Git 鼓勵開發者頻繁使用分支，正是因為有著這些特性作保障。

接下來看看，我們為什麼應該頻繁使用分支。

分支的新建與合併

現在讓我們來看一個簡單的分支與合併的例子，實際工作中大體也會用到這樣的工作流程：

1. 開發某個網站。
2. 為實現某個新的需求，創建一個分支。
3. 在這個分支上開展工作。

假設此時，你突然接到一個電話說有個很嚴重的問題需要緊急修補，那麼可以按照下面的方式處理：

1. 返回到原先已經發佈到生產伺服器上的分支。
2. 為這次緊急修補建立一個新分支，並在其中修復問題。
3. 通過測試後，回到生產伺服器所在的分支，將修補分支合併進來，然後再推送到生產伺服器上。
4. 切換到之前實現新需求的分支，繼續工作。

分支的新建與切換

首先，我們假設你正在專案中愉快地工作，並且已經提交了幾次更新（見圖 3-10）。

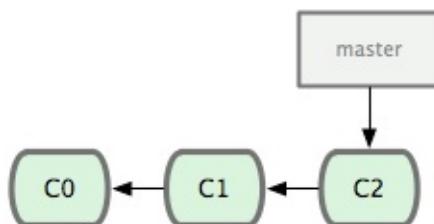


圖 3-10. 一個簡短的提交歷史

現在，你決定要修補問題追蹤系統上的 #53 問題。順帶說明下，Git 並不同任何特定的問題追蹤系統打交道。這裡為了說明要解決的問題，才把新建的分支取名為 iss53。要新建並切換到該分支，運行 `git checkout` 並加上 `-b` 參數：

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

這相當於執行下面這兩條命令：

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

圖 3-11 示意該命令的執行結果。

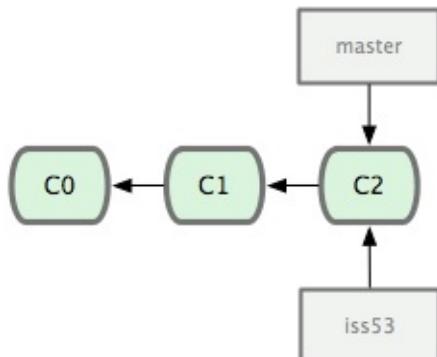
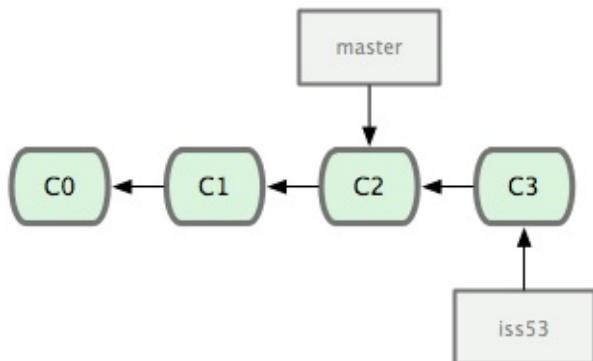


圖 3-11. 創建了一個新分支的指標

接著你開始嘗試修復問題，在提交了若干次更新後，`iss53` 分支的指標也會隨著向前推進，因為它就是當前分支（換句話說，當前的 `HEAD` 指標正指向 `iss53`，見圖 3-12）：

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

圖 3-12. `iss53` 分支隨工作進展向前推進

現在你就接到了那個網站問題的緊急電話，需要馬上修補。有了 Git，我們就不需要同時發佈這個補丁和 `iss53` 裡作出的修改，也不需要在創建和發佈該補丁到伺服器之前花費大力氣來復原這些修改。唯一需要的僅僅是切換回 `master` 分支。

不過在此之前，留心你的暫存區或者工作目錄裡，那些還沒有提交的修改，它會和你即將檢出的分支產生衝突從而阻止 Git 為你切換分支。切換分支的時候最好保持一個清潔的工作區域。稍後會介紹幾個繞過這種問題的辦法（分別叫做 `stashing` 和 `commit amending`）。目前已經提交了所有的修改，所以接下來可以正常轉換到 `master` 分支：

```
$ 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 之前一模一樣，你可以集中精力進行緊急修補。這一點值得牢記：Git 會把工作目錄的內容恢復為檢出某分支時它所指向的那個提交物件的快照。它會自動添加、刪除和修改檔以確保目錄的內容和你當時提交時完全一樣。

接下來，你得進行緊急修補。我們創建一個緊急修補分支 `hotfix` 來開展工作，直到搞定（見圖 3-13）：

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 3a0874c] fixed the broken email address
 1 files changed, 1 deletion(-)
```

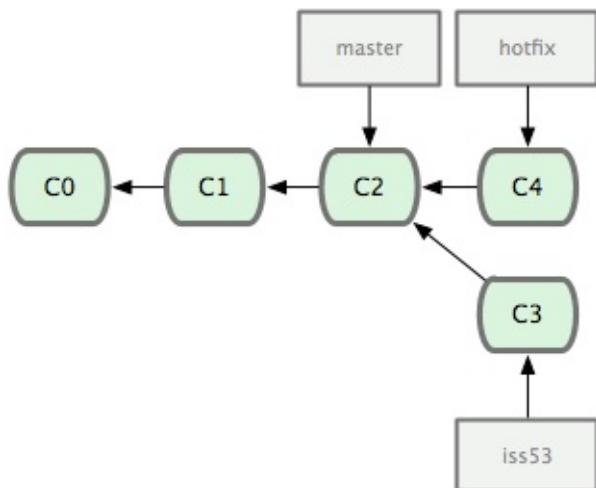


圖 3-13. `hotfix` 分支是從 `master` 分支所在點分化出來的

有必要作些測試，確保修補是成功的，然後回到 `master` 分支並把它合併進來，然後發佈到生產伺服器。用 `git merge` 命令來進行合併：

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

請注意，合併時出現了“Fast forward”的提示。由於當前 `master` 分支所在的提交物件是要併入的 `hotfix` 分支的直接上游，Git 只需把 `master` 分支指標直接右移。換句話說，如果順著一個分支走下去可以到達另一個分支的話，那麼 Git 在合併兩者時，只會簡單地把指標右移，因為這種單線的歷史分支不存在任何需要解決的分歧，所以這種合併過程可以稱為快進（Fast forward）。

現在最新的修改已經在當前 `master` 分支所指向的提交物件中了，可以部署到生產伺服器上去了（見圖 3-14）。

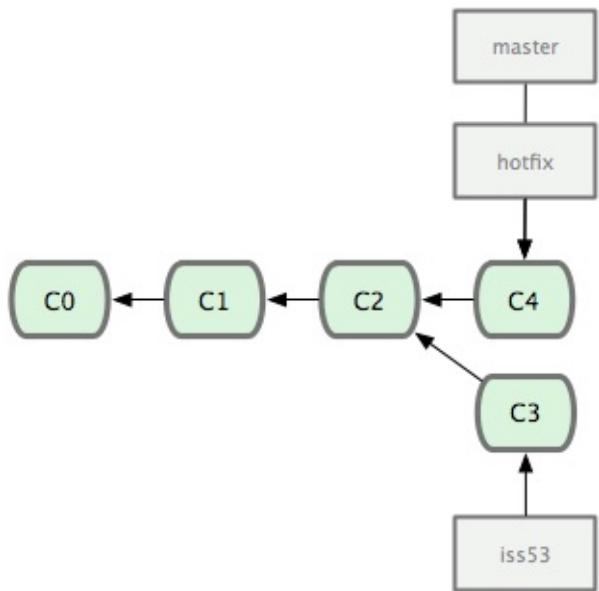


圖 3-14. 合併之後，`master` 分支和 `hotfix` 分支指向同一位置。

在那個超級重要的修補發佈以後，你想要回到被打擾之前的工作。由於當前 `hotfix` 分支和 `master` 都指向相同的提交物件，所以 `hotfix` 已經完成了歷史使命，可以刪掉了。使用 `git branch` 的 `-d` 選項執行刪除操作：

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

現在回到之前未完成的 #53 問題修復分支上繼續工作（圖 3-15）：

```
$ 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(+)
```

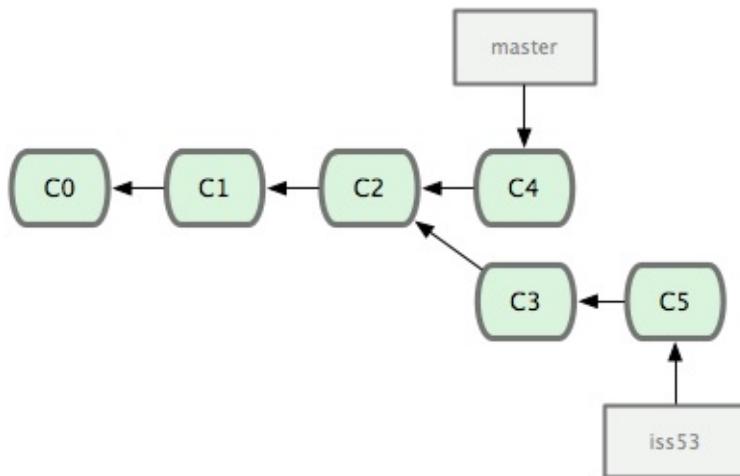


圖 3-15. iss53 分支可以不受影響繼續推進。

值得注意的是之前 `hotfix` 分支的修改內容尚未包含到 `iss53` 中來。如果需要納入此次修補，可以用 `git merge master` 把 `master` 分支合併到 `iss53`；或者等 `iss53` 完成之後，再將 `iss53` 分支中的更新併入 `master`。

分支的合併

在問題 #53 相關的工作完成之後，可以合併回 `master` 分支。實際操作同前面合併 `hotfix` 分支差不多，只需回到 `master` 分支，運行 `git merge` 命令指定要合併進來的分支：

```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

請注意，這次合併操作的底層實現，並不同於之前 `hotfix` 的併入方式。因為這次你的開發歷史是從更早的地方開始分叉的。由於當前 `master` 分支所指向的提交物件（C4）並不是 `iss53` 分支的直接祖先，Git 不得不進行一些額外處理。就此例而言，Git 會用兩個分支的末端（C4 和 C5）以及它們的共同祖先（C2）進行一次簡單的三方合併計算。圖 3-16 用紅框標出了 Git 用於合併的三個提交對象：

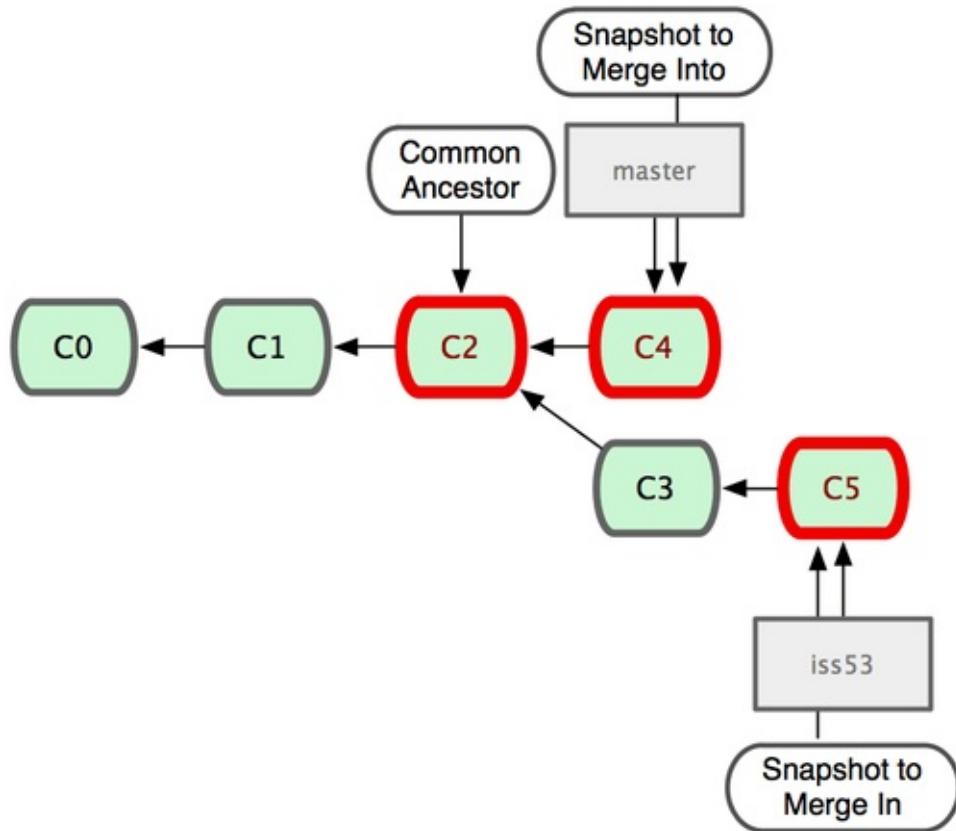


圖 3-16. Git 為分支合併自動識別出最佳的同源合併點。

這次，Git 沒有簡單地把分支指標右移，而是對三方合併後的結果重新做一個新的快照，並自動創建一個指向它的提交物件（C6）（見圖 3-17）。這個提交物件比較特殊，它有兩個祖先（C4 和 C5）。

值得一提的是 Git 可以自己裁決哪個共同祖先才是最佳合併基礎；這和 CVS 或 Subversion（1.5 以後的版本）不同，它們需要開發者手工指定合併基礎。所以此特性讓 Git 的合併操作比其他系統都要簡單不少。

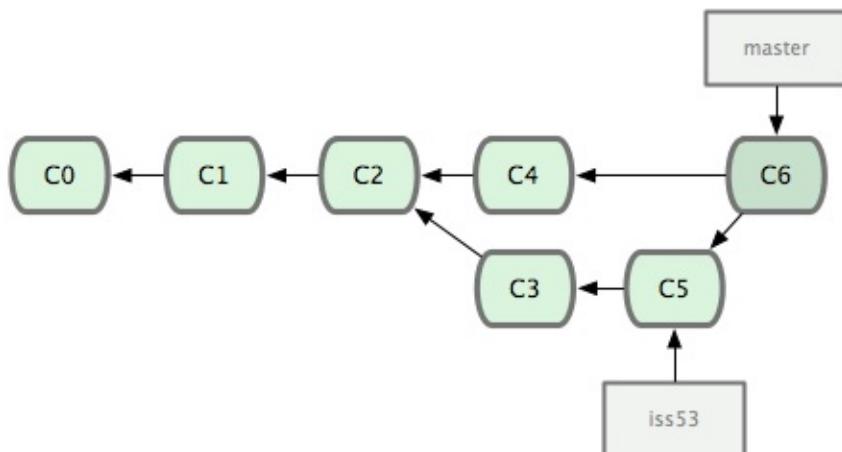


圖 3-17. Git 自動創建了一個包含了合併結果的提交物件。

既然之前的工作成果已經合併到 `master` 了，那麼 `iss53` 也就沒用了。你可以就此刪除它，並在問題追蹤系統裡關閉該問題。

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

遇到衝突時的分支合併

有時候合併操作並不會如此順利。如果在不同的分支中都修改了同一個檔的同一部分，Git 就無法乾淨地把兩者合到一起（譯注：邏輯上說，這種問題只能由人來裁決。）。如果你在解決問題 #53 的過程中修改了 `hotfix` 中修改的部分，將得到類似下面的結果：

```
$ 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 作了合併，但沒有提交，它會停下來等你解決衝突。要看看哪些檔在合併時發生衝突，可以用 `git status` 查閱：

```
$ 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")
```

任何包含未解決衝突的檔都會以未合併（unmerged）的狀態列出。Git 會在有衝突的檔裡加入標準的衝突解決標記，可以通過它們來手工定位並解決這些衝突。可以看到此檔包含類似下面這樣的部分：

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

可以看到 `=====` 隔開的上半部分，是 `HEAD`（即 `master` 分支，在運行 `merge` 命令時所切換到的分支）中的內容，下半部分是在 `iss53` 分支中的內容。解決衝突的辦法無非是二者選其一或者由你親自整合到一起。比如你可以通過把這段內容替換為下面這樣來解決：

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

這個解決方案各採納了兩個分支中的一部分內容，而且我還刪除了 <<<<< 和 ====== 這些行。在解決了所有檔裡的所有衝突後，運行 git add 將把它們標記為已解決狀態（譯注：實際上就是來一次快照保存到暫存區域。）。因為一旦暫存，就表示衝突已經解決。如果你想用一個有圖形介面的工具來解決這些問題，不妨運行 git mergetool，它會調用一個視覺化的合併工具並引導你解決所有衝突：

```
$ 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 tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge
Merging:
index.html

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

如果不想用默認的合併工具（Git 為我默認選擇了 opendiff，因為我在 Mac 上運行了該命令），你可以在上方"merge tool candidates"裡找到可用的合併工具列表，輸入你想用的工具名。我們將在第七章討論怎樣改變環境中的預設值。

退出合併工具以後，Git 會詢問你合併是否成功。如果回答是，它會為你把相關檔暫存起來，以表明狀態為已解決。

再運行一次 git status 來確認所有衝突都已解決：

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

modified:   index.html
```

如果覺得滿意了，並且確認所有衝突都已解決，也就是進入了暫存區，就可以用 git commit 來完成這次合併提交。提交的記錄差不多是這樣：

```
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.
#
```

如果想給將來看這次合併的人一些方便，可以修改該資訊，提供更多合併細節。比如你都作了哪些改動，以及這麼做的原因。有時候裁決衝突的理由並不直接或明顯，有必要略加注解。

分支的管理

到目前為止，你已經學會了如何創建、合併和刪除分支。除此之外，我們還需要學習如何管理分支，在日後的常規工作中會經常用到下面介紹的管理命令。

`git branch` 命令不僅僅能創建和刪除分支，如果不加任何參數，它會給出當前所有分支的清單：

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

注意看 `master` 分支前的 `*` 字元：它表示當前所在的分支。也就是說，如果現在提交更新，`master` 分支將隨著開發進度前移。若要查看各個分支最後一個提交物件的資訊，運行 `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
```

要從該清單中篩選出你已經（或尚未）與當前分支合併的分支，可以用 `--merged` 和 `--no-merged` 選項（Git 1.5.6 以上版本）。比如用 `git branch --merged` 查看哪些分支已被併入當前分支（譯注：也就是說哪些分支是當前分支的直接上游。）：

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

之前我們已經合併了 `iss53`，所以在這裡會看到它。一般來說，清單中沒有 `*` 的分支通常都可以用 `git branch -d` 來刪掉。原因很簡單，既然已經把它們所包含的工作整合到了其他分支，刪掉也不會損失什麼。

另外可以用 `git branch --no-merged` 查看尚未合併的工作：

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

它會顯示還未合併進來的分支。由於這些分支中還包含著尚未合併進來的工作成果，所以簡單地用 `git branch -d` 刪除該分支會提示錯誤，因為那樣做會丟失資料：

```
$ 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'.
```

不過，如果你確實想要刪除該分支上的改動，可以用大寫的刪除選項 `-D` 強制執行，就像上面提示資訊中給出的那樣。

利用分支進行開發的工作流程

現在我們已經學會了新建分支和合併分支，可以（或應該）用它來做點什麼呢？在本節，我們會介紹一些利用分支進行開發的工作流程。而正是由於分支管理的便捷，才衍生出了這類典型的工作模式，你可以根據專案的實際情況選擇一種用用看。

長期分支

由於 Git 使用簡單的三方合併，所以就算在較長一段時間內，反復多次把某個分支合併到另一分支，也不是什麼難事。也就是說，你可以同時擁有多個開放的分支，每個分支用於完成特定的任務，隨著開發的推進，你可以隨時把某個特性分支的成果並到其他分支中。

許多使用 Git 的開發者都喜歡用這種方式來開展工作，比如僅在 `master` 分支中保留完全穩定的代碼，即已經發佈或即將發佈的代碼。與此同時，他們還有一個名為 `develop` 或 `next` 的平行分支，專門用於後續的開發，或僅用於穩定性測試 — 當然並不是說一定要絕對穩定，不過一旦進入某種穩定狀態，便可以把它合併到 `master` 裡。這樣，在確保這些已完成的特性分支（短期分支，比如之前的 `iss53` 分支）能夠通過所有測試，並且不會引入更多錯誤之後，就可以並到主幹分支中，等待下一次的發佈。

本質上我們剛才談論的，是隨著提交物件不斷右移的指標。穩定分支的指標總是在提交歷史中落後一大截，而前沿分支總是比較靠前（見圖 3-18）。

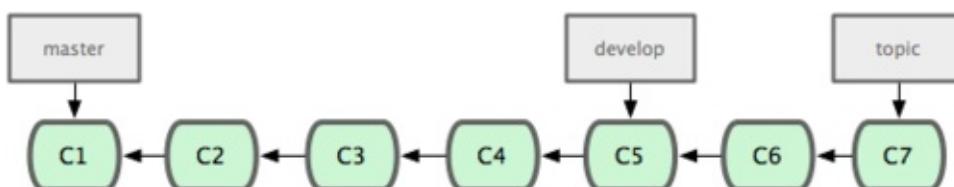


圖 3-18. 穩定分支總是比較老舊。

或者把它們想像成工作流水線，或許更好理解一些，經過測試的提交物件集合被遴選到更穩定的流水線（見圖 3-19）。

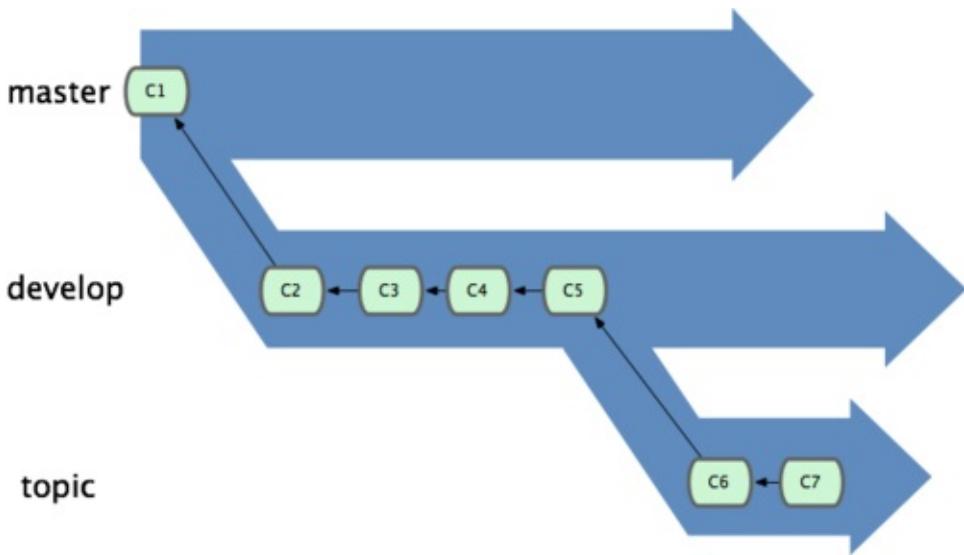


圖 3-19. 想像成流水線可能會容易點。

你可以用這招維護不同層次的穩定性。某些大專案還會有個 `proposed` (建議) 或 `pu` (`proposed updates`, 建議更新) 分支，它包含著那些可能還沒有成熟到進入 `next` 或 `master` 的內容。這麼做的目的是擁有不同層次的穩定性：當這些分支進入到更穩定的水準時，再把它們合併到更高層分支中去。再次說明下，使用多個長期分支的做法並非必需，不過一般來說，對於特大型項目或特複雜的專案，這麼做確實更容易管理。

特性分支

在任何規模的專案中都可以使用特性 (Topic) 分支。一個特性分支是指一個短期的，用來實現單一特性或與其相關工作的分支。可能你在以前的版本控制系統裡從未做過類似這樣的事情，因為通常創建與合併分支消耗太大。然而在 Git 中，一天之內建立、使用、合併再刪除多個分支是常見的事。

我們在上節的例子裡已經見過這種用法了。我們創建了 `iss53` 和 `hotfix` 這兩個特性分支，在提交了若干更新後，把它們合併到主幹分支，然後刪除。該技術允許你迅速且完全的進行語境切換 — 因為你的工作分散在不同的流水線裡，每個分支裡的改變都和它的目標特性相關，流覽代碼之類的事情因而變得更簡單了。你可以把作出的改變保持在特性分支中幾分鐘，幾天甚至幾個月，等它們成熟以後再合併，而不用在乎它們建立的順序或者進度。

現在我們來看一個實際的例子。請看圖 3-20，由下往上，起先我們在 `master` 工作到 C1，然後開始一個新分支 `iss91` 嘗試修復 91 號缺陷，提交到 C6 的時候，又冒出一個解決該問題的新辦法，於是從之前 C4 的地方又分出一個分支 `iss91v2`，幹到 C8 的時候，又回到主幹 `master` 中提交了 C9 和 C10，再回到 `iss91v2` 繼續工作，提交 C11，接著，又冒出個不太確定的想法，從 `master` 的最新提交 C10 處開了個新的分支 `dumbidea` 做些試驗。

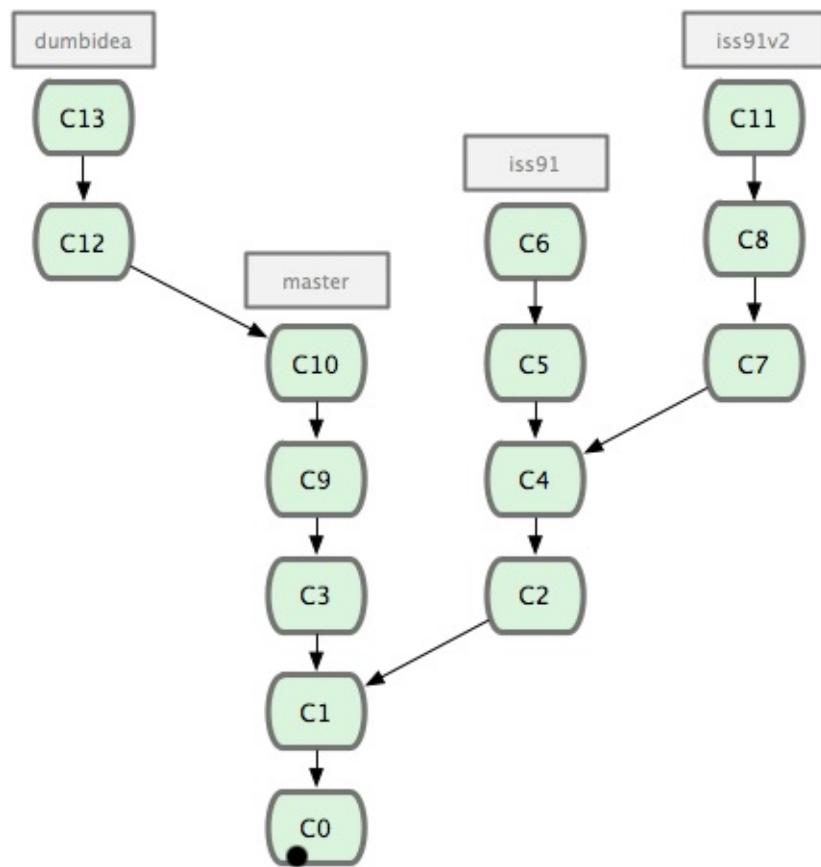


圖 3-20. 擁有多個特性分支的提交歷史。

現在，假定兩件事情：我們最終決定使用第二個解決方案，即 `iss91v2` 中的辦法；另外，我們把 `dumbidea` 分支拿給同事們看了以後，發現它竟然是個天才之作。所以接下來，我們準備拋棄原來的 `iss91` 分支（實際上會丟棄 `C5` 和 `C6`），直接在主幹中併入另外兩個分支。最終的提交歷史將變成圖 3-21 這樣：

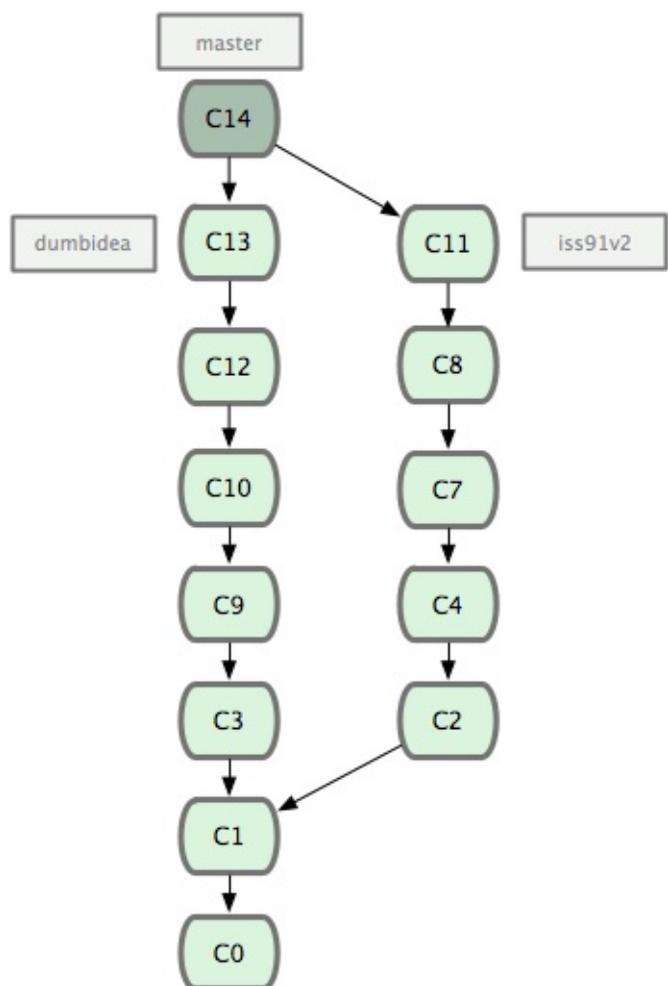


圖 3-21. 合併了 `dumbidea` 和 `iss91v2` 後的分支歷史。

請務必牢記這些分支全部都是本地分支，這一點很重要。當你在使用分支及合併的時候，一切都是在你自己的 Git 倉庫中進行的 — 完全不涉及與伺服器的交互。

遠端分支

遠端分支（remote branch）是對遠端倉庫中的分支的索引。它們是一些無法移動的本地分支；只有在 Git 進行網路交互時才會更新。遠端分支就像是書簽，提醒著你上次連接遠端倉庫時上面各分支的位置。

我們用 `(遠端倉庫名)/(分支名)` 這樣的形式表示遠端分支。比如我們想看看上次同 `origin` 倉庫通訊時 `master` 分支的樣子，就應該查看 `origin/master` 分支。如果你和同伴一起修復某個問題，但他們先推送了一個 `iss53` 分支到遠端倉庫，雖然你可能也有一個本地的 `iss53` 分支，但指向伺服器上最新更新的卻應該是 `origin/iss53` 分支。

可能有點亂，我們不妨舉例說明。假設你們團隊有個地址為 `git.ourcompany.com` 的 Git 伺服器。如果你從這裡克隆，Git 會自動為你將此遠端倉庫命名為 `origin`，並下載其中所有的資料，建立一個指向它的 `master` 分支的指標，在本地命名為 `origin/master`，但你無法在本地更改其資料。接著，Git 建立一個屬於你自己的本地 `master` 分支，始於 `origin` 上 `master` 分支相同的位置，你可以就此開始工作（見圖 3-22）：

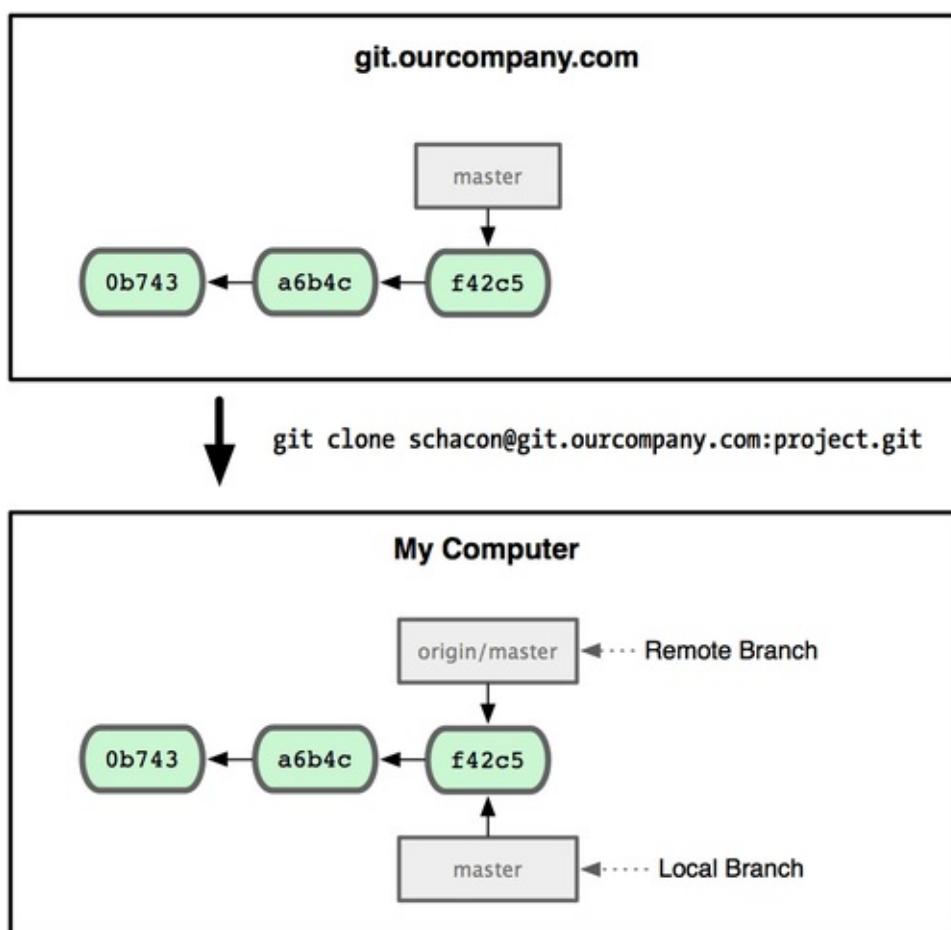


圖 3-22. 一次 Git 克隆會建立你自己的本地分支 `master` 和遠端分支 `origin/master`，並且將它們都指向 `origin` 上的 `master` 分支。

如果你在本地 `master` 分支做了些改動，與此同時，其他人向 `git.ourcompany.com` 推送了他們的更新，那麼伺服器上的 `master` 分支就會向前推進，而於此同時，你在本地的提交歷史正朝向不同方向發展。不過只要你不和伺服器通訊，你的 `origin/master` 指標仍然保持原位不會移動（見圖 3-23）。

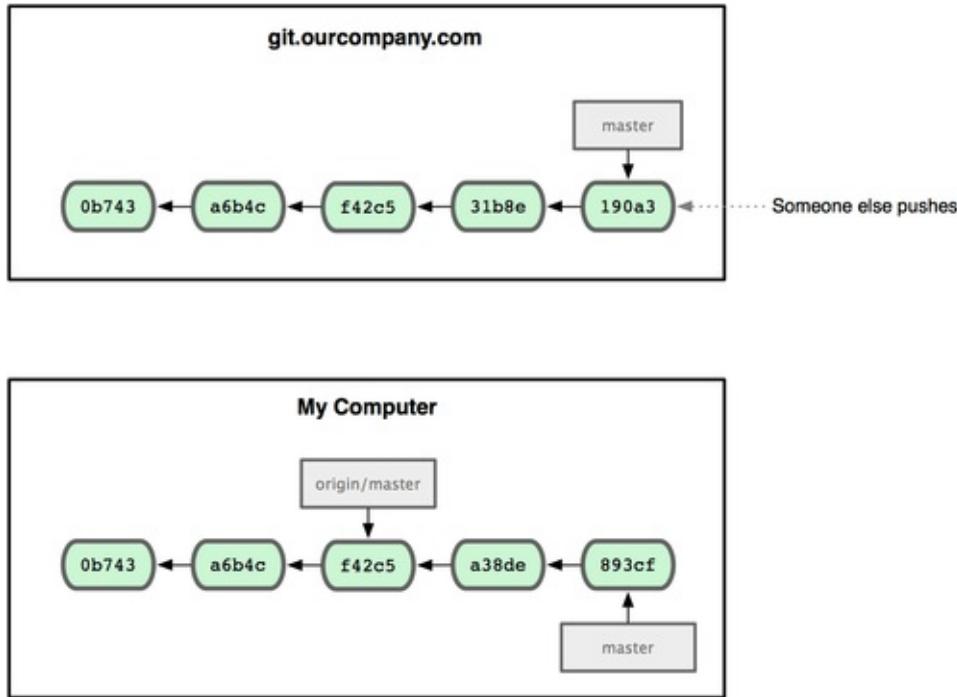


圖 3-23. 在本地工作的同時有人向遠端倉庫推送內容會讓提交歷史開始分流。

可以運行 `git fetch origin` 來同步遠端伺服器上的資料到本地。該命令首先找到 `origin` 是哪個伺服器（本例為 `git.ourcompany.com`），從上面獲取你尚未擁有的資料，更新你本地的資料庫，然後把 `origin/master` 的指針移到它最新的位置上（見圖 3-24）。

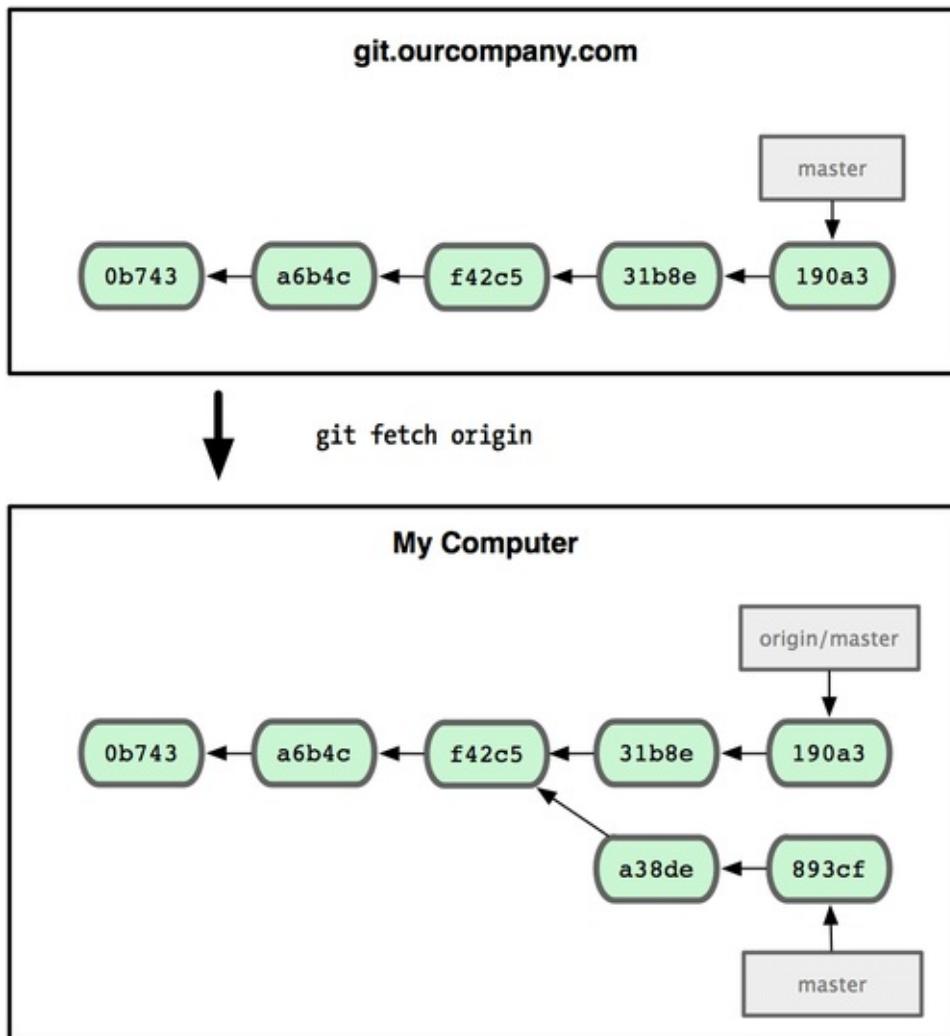


圖 3-24. `git fetch` 命令會更新 remote 索引。

為了演示擁有多個遠端分支（在不同的遠端伺服器上）的專案是如何工作的，我們假設你還有另一個僅供你的敏捷開發小組使用的內部伺服器 `git.team1.ourcompany.com`。可以用第二章中提到的 `git remote add` 命令把它加為當前專案的遠端分支之一。我們把它命名為 `teamone`，以便代替完整的 Git URL 以方便使用（見圖 3-25）。

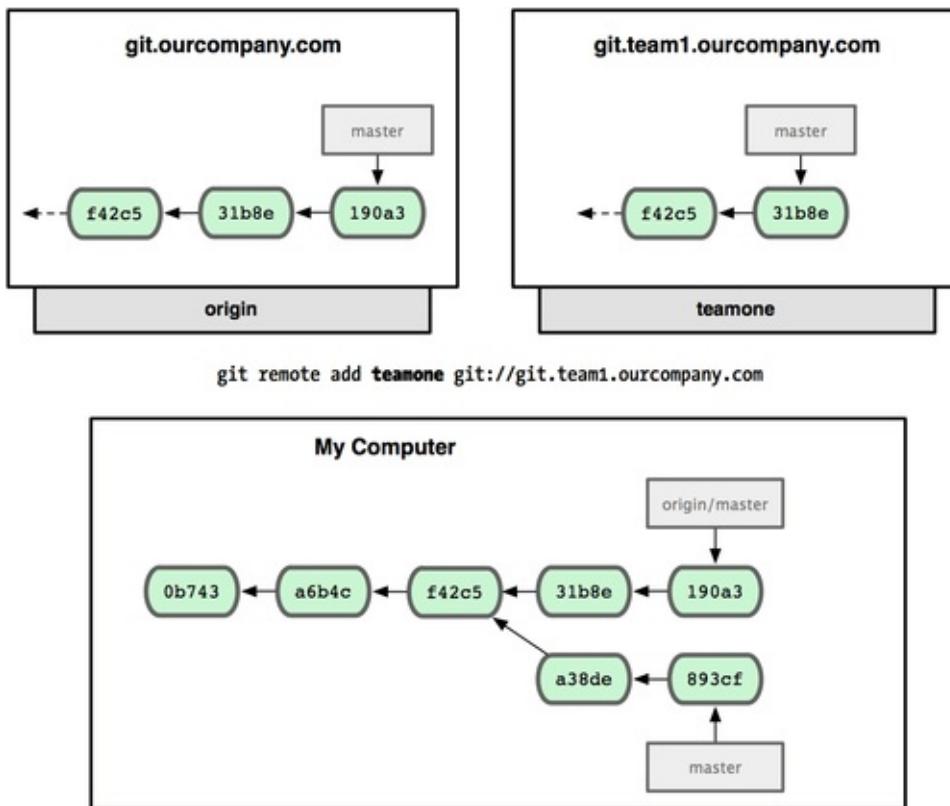


圖 3-25. 把另一個伺服器加為遠端倉庫

現在你可以用 `git fetch teamone` 來獲取小組伺服器上你還沒有的資料了。由於當前該伺服器上的內容是你 `origin` 伺服器上的子集，Git 不會下載任何資料，而只是簡單地創建一個名為 `teamone/master` 的遠端分支，指向 `teamone` 伺服器上 `master` 分支所在的提交物件 `31b8e`（見圖 3-26）。

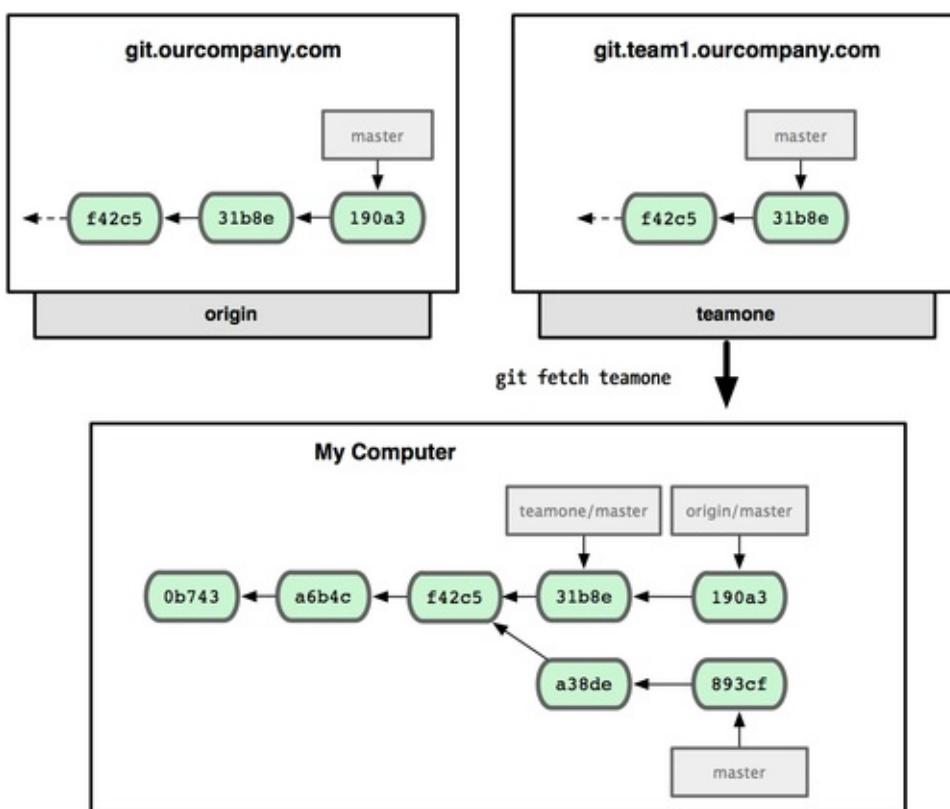


圖 3-26. 你在本地有了一個指向 teamone 伺服器上 master 分支的索引。

推送本地分支

要想和其他人分享某個本地分支，你需要把它推送到一個你擁有寫許可權的遠端倉庫。你創建的本地分支不會因為你的寫入操作而被自動同步到你引入的遠端伺服器上，你需要明確地執行推送分支的操作。換句話說，對於無意分享的分支，你儘管保留為私人分支好了，而只推送那些協同工作要用到的特性分支。

如果你有個叫 `serverfix` 的分支需要和他人一起開發，可以運行 `git push (遠端倉庫名) (分支名)`：

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

這裡其實走了一點捷徑。Git 自動把 `serverfix` 分支名擴展為

`refs/heads/serverfix:refs/heads/serverfix`，意為“取出我在本地的 `serverfix` 分支，推送到遠端倉庫的 `serverfix` 分支中去”。我們將在第九章進一步介紹 `refs/heads/` 部分的細節，不過一般使用的時候都可以省略它。也可以運行 `git push origin serverfix:serverfix` 來實現相同的效果，它的意思是“上傳我本地的 `serverfix` 分支到遠端倉庫中去，仍舊稱它為 `serverfix` 分支”。通過此語法，你可以把本地分支推送到某個命名不同的遠端分支：若想把遠端分支叫作 `awesomebranch`，可以用 `git push origin serverfix:awesomebranch` 來推送數據。

接下來，當你的協作者再次從伺服器上獲取資料時，他們將得到一個新的遠端分支 `origin/serverfix`，並指向伺服器上 `serverfix` 所指向的版本：

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix -> origin/serverfix
```

值得注意的是，在 `fetch` 操作下載好新的遠端分支之後，你仍然無法在本地編輯該遠端倉庫中的分支。換句話說，在本例中，你不會有一個新的 `serverfix` 分支，有的只是一個你無法移動的 `origin/serverfix` 指標。

如果要把該遠端分支的內容合併到當前分支，可以運行 `git merge origin/serverfix`。如果想要一份自己的 `serverfix` 來開發，可以在遠端分支的基礎上分化出一個新的分支來：

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

這會切換到新建的 `serverfix` 本地分支，其內容同遠端分支 `origin/serverfix` 一致，這樣你就可以在裡面繼續開發了。

跟蹤遠端分支

從遠端分支 `checkout` 出來的本地分支，稱為 跟蹤分支 (tracking branch)。跟蹤分支是一種和某個遠端分支有直接聯繫的本地分支。在跟蹤分支裡輸入 `git push`，Git 會自行推斷應該向哪個伺服器的哪個分支推送資料。同樣，在這些分支裡運行 `git pull` 會獲取所有遠端索引，並把它們的資料都合併到本地分支中來。

在克隆倉庫時，Git 通常會自動創建一個名為 `master` 的分支來跟蹤 `origin/master`。這正是 `git push` 和 `git pull` 一開始就能正常工作的原因。當然，你可以隨心所欲地設定為其它跟蹤分支，比如 `origin` 上除了 `master` 之外的其它分支。剛才我們已經看到了這樣的一個例子：`git checkout -b [分支名] [遠端名]/[分支名]`。如果你有 1.6.2 以上版本的 Git，還可以用 `--track` 選項簡化：

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

要為本地分支設定不同于遠端分支的名字，只需在第一個版本的命令裡換個名字：

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

現在你的本地分支 `sf` 會自動將推送和抓取資料的位置定位到 `origin/serverfix` 了。

刪除遠端分支

如果不再需要某個遠端分支了，比如搞定了某個特性並把它合併進了遠端的 `master` 分支（或任何其他存放穩定代碼的分支），可以用這個非常無厘頭的語法來刪除它：`git push [遠端名] :[分支名]`。如果想在伺服器上刪除 `serverfix` 分支，運行下面的命令：

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

咚！伺服器上的分支沒了。你最好特別留心這一頁，因為你一定會用到那個命令，而且你很可能會忘掉它的語法。有種方便記憶這條命令的方法：記住我們不久前見過的 `git push [遠端名] [本地分支]:[遠端分支]` 語法，如果省略 `[本地分支]`，那就等於是說“在這裡提取空白然後把它變成 `[遠端分支]`”。

分支的衍合

把一個分支中的修改整合到另一個分支的辦法有兩種：`merge` 和 `rebase`（譯注：`rebase` 的翻譯暫定為“衍合”，大家知道就可以了。）。在本章我們會學習什麼是衍合，如何使用衍合，為什麼衍合操作如此富有魅力，以及我們應該在什麼情況下使用衍合。

基本的衍合操作

請回顧之前有關合併的一節（見圖 3-27），你會看到開發進程分叉到兩個不同分支，又各自提交了更新。

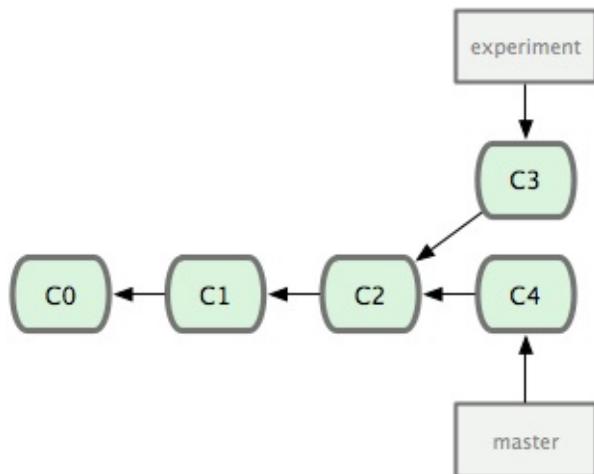


圖 3-27. 最初分叉的提交歷史。

之前介紹過，最容易的整合分支的方法是 `merge` 命令，它會把兩個分支最新的快照（C3 和 C4）以及二者最新的共同祖先（C2）進行三方合併，合併的結果是產生一個新的提交物件（C5）。如圖 3-28 所示：

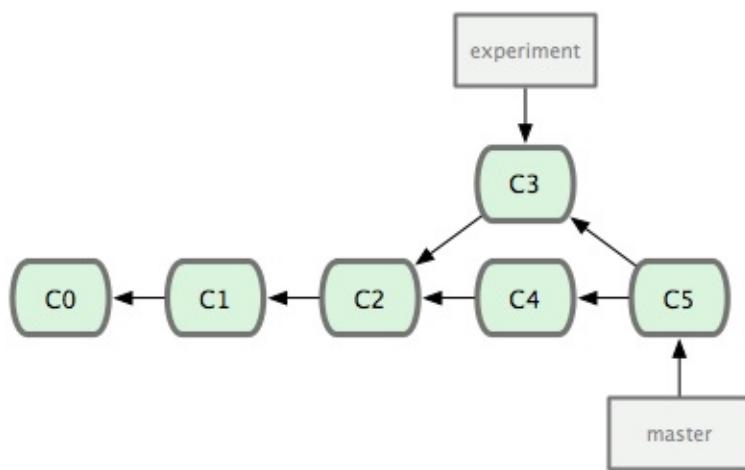


圖 3-28. 通過合併一個分支來整合分叉了的歷史。

其實，還有另外一個選擇：你可以在 C3 裡產生的變化補丁在 C4 的基礎上重新打一遍。在 Git 裡，這種操作叫做衍合（*rebase*）。有了 `rebase` 命令，就可以把在一個分支裡提交的改變移到另一個分支裡重放一遍。

在上面這個例子中，運行：

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

它的原理是回到兩個分支最近的共同祖先，根據當前分支（也就是要進行衍合的分支 `experiment`）後續的歷次提交物件（這裡只有一個 C3），生成一系列檔補丁，然後以基底分支（也就是主幹分支 `master`）最後一個提交物件（C4）為新的出發點，逐個應用之前準備好的補丁檔，最後會生成一個新的合併提交物件（C3'），從而改寫 `experiment` 的提交歷史，使它成為 `master` 分支的直接下游，如圖 3-29 所示：

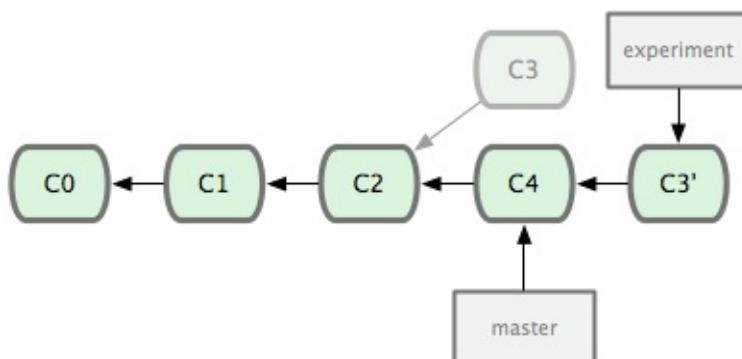


圖 3-29. 把 C3 裡產生的改變到 C4 上重演一遍。

現在回到 `master` 分支，進行一次快進合併（見圖 3-30）：

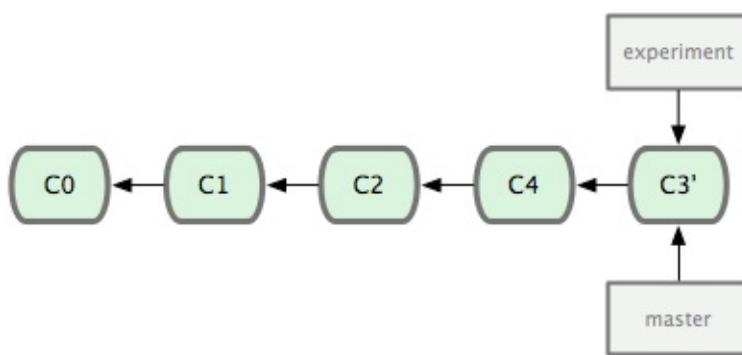


圖 3-30. master 分支的快進。

現在的 C3' 對應的快照，其實和普通的三方合併，即上個例子中的 C5 對應的快照內容一模一樣了。雖然最後整合得到的結果沒有任何區別，但衍合能產生一個更為整潔的提交歷史。如果觀察一個衍合過的分支的歷史記錄，看起來會更清楚：彷彿所有修改都是在一根線上先後進行的，儘管實際上它們原本是同時並行發生的。

一般我們使用衍合的目的，是想要得到一個能在遠端分支上乾淨應用的補丁 — 比如某些項目你不是維護者，但想幫點忙的話，最好用衍合：先在自己的一個分支裡進行開發，當準備向主專案提交補丁的時候，根據最新的 `origin/master` 進行一次衍合操作然後再提交，這樣維護者就不需要做任何整合工作（譯注：實際上是把解決分支補丁同最新主幹代碼之間衝突的責任，化轉為由提交補丁的人來解決。），只需根據你提供的倉庫位址作一次快進合併，或者直接採納你提交的補丁。

請注意，合併結果中最後一次提交所指向的快照，無論是通過衍合，還是三方合併，都會得到相同的快照內容，只不過提交歷史不同罷了。衍合是按照每行的修改次序重演一遍修改，而合併是把最終結果合在一起。

有趣的衍合

衍合也可以放到其他分支進行，並不一定非得根據分化之前的分支。以圖 3-31 的歷史為例，我們為了給伺服器端代碼添加一些功能而創建了特性分支 `server`，然後提交 C3 和 C4。然後又從 C3 的地方再增加一個 `client` 分支來對用戶端代碼進行一些相應修改，所以提交了 C8 和 C9。最後，又回到 `server` 分支提交了 C10。

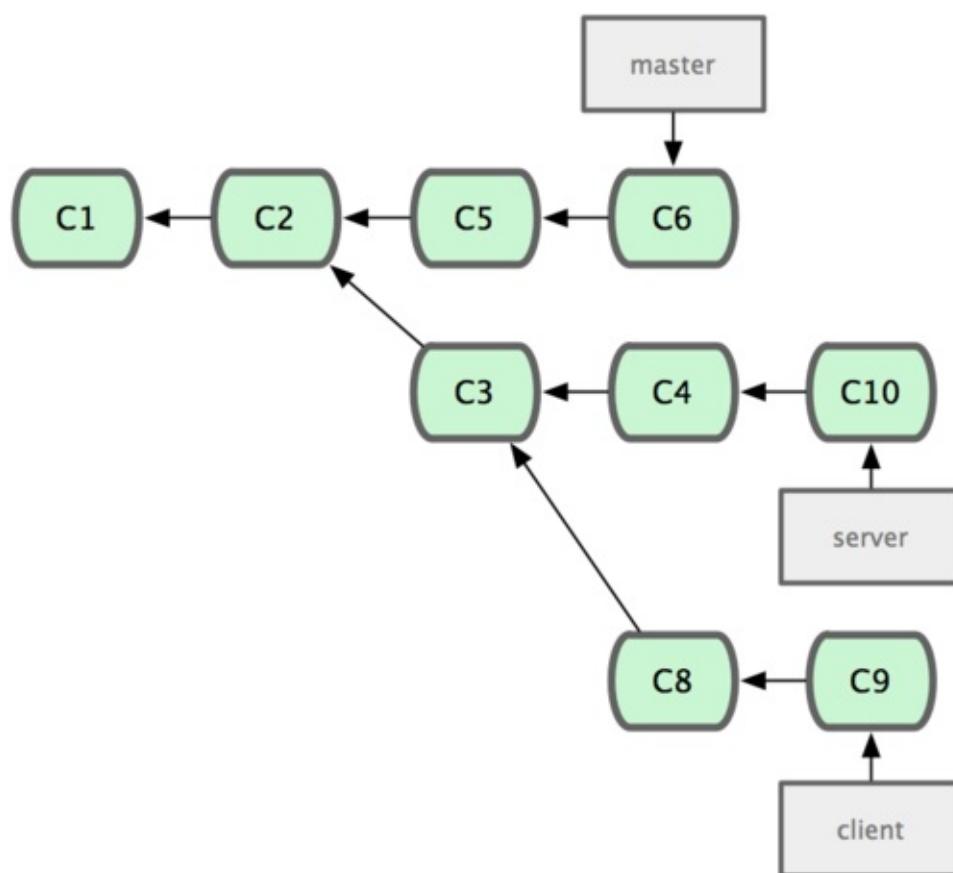


圖 3-31. 從一個特性分支裡再分出一個特性分支的歷史。

假設在接下來的一次軟體發佈中，我們決定先把用戶端的修改並到主線中，而暫緩併入服務端軟體的修改（因為還需要進一步測試）。這個時候，我們就可以把基於 `server` 分支而非 `master` 分支的改變（即 C8 和 C9），跳過 `server` 直接放到 `master` 分支中重演一遍，但這需要用 `git rebase` 的 `--onto` 選項指定新的基底分支 `master`：

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

這好比在說：“取出 `client` 分支，找出 `client` 分支和 `server` 分支的共同祖先之後的變化，然後把它們在 `master` 上重演一遍”。是不是有點複雜？不過它的結果如圖 3-32 所示，非常酷（譯注：雖然 `client` 裡的 C8, C9 在 C3 之後，但這僅表明時間上的先後，而非在 C3 修改的基礎上進一步改動，因為 `server` 和 `client` 這兩個分支對應的代碼應該是兩套檔，雖然這麼說不是很嚴格，但應理解為在 C3 時間點之後，對另外的檔所做的 C8, C9 修 改，放到主幹重演。）：

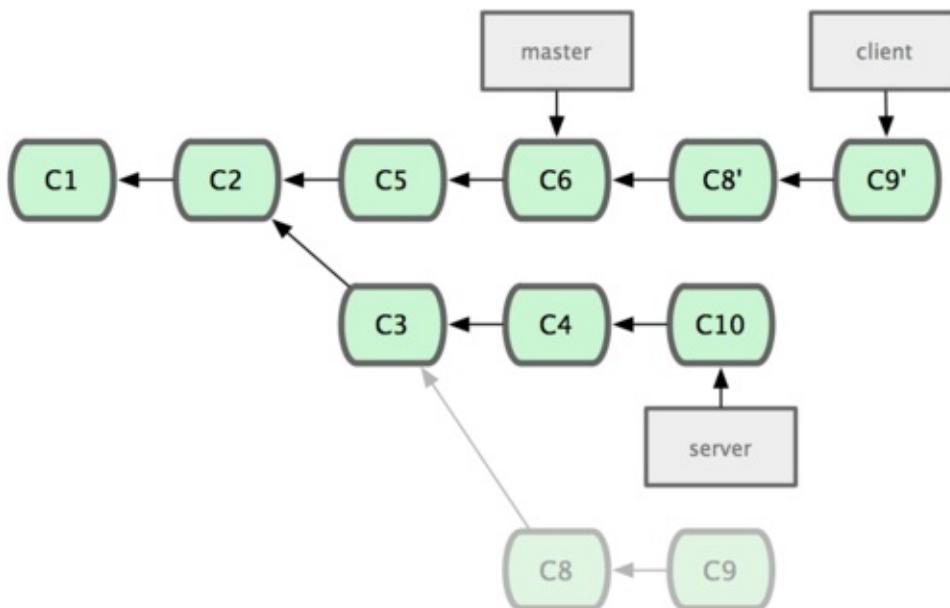


圖 3-32. 將特性分支上的另一個特性分支衍合到其他分支。

現在可以快進 `master` 分支了（見圖 3-33）：

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

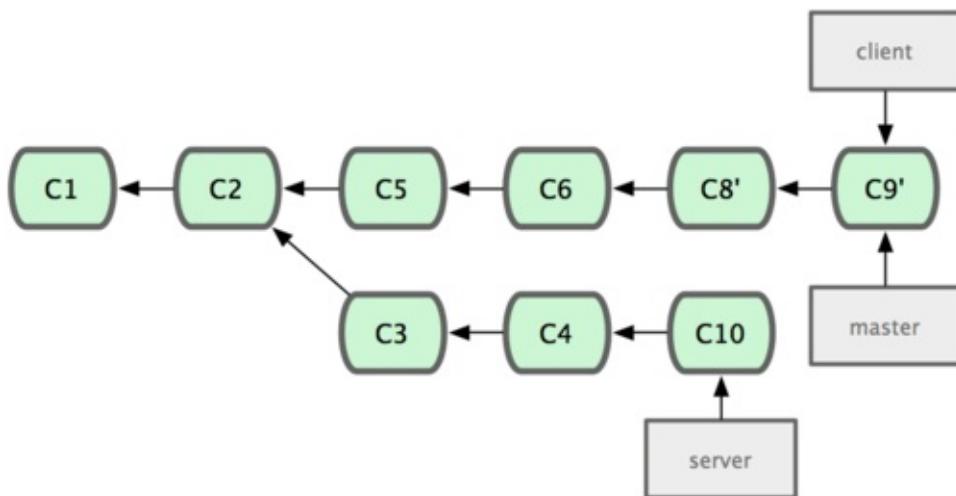


圖 3-33. 快進 `master` 分支，使之包含 `client` 分支的變化。

現在我們決定把 `server` 分支的變化也包含進來。我們可以直接把 `server` 分支衍合到 `master`，而不用手工切換到 `server` 分支後再執行衍合操作 — `git rebase [主分支] [特性分支]` 命令會先取出特性分支 `server`，然後在主分支 `master` 上重演：

```
$ git rebase master server
```

於是，`server` 的進度應用到 `master` 的基礎上，如圖 3-34 所示：

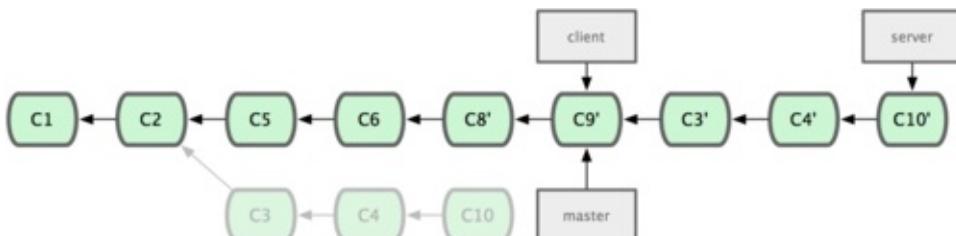


圖 3-34. 在 `master` 分支上衍合 `server` 分支。

然後就可以快進主幹分支 `master` 了：

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

現在 `client` 和 `server` 分支的變化都已經集成到主幹分支來了，可以刪掉它們了。最終我們的提交歷史會變成圖 3-35 的樣子：

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

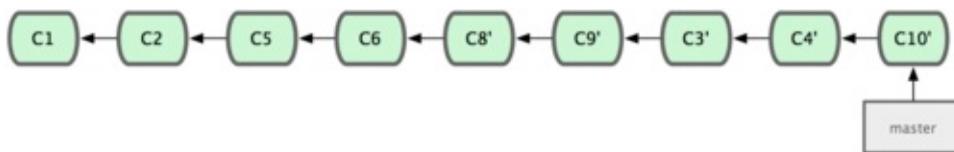


圖 3-35. 最終的提交歷史

衍合的風險

呃，奇妙的衍合也並非完美無缺，要用它得遵守一條準則：

一旦分支中的提交物件發佈到公共倉庫，就千萬不要對該分支進行衍合操作。

如果你遵循這條金科玉律，就不會出差錯。否則，人民群眾會仇恨你，你的朋友和家人也會嘲笑你，唾棄你。

在進行衍合的時候，實際上拋棄了一些現存的提交物件而創造了一些類似但不同的新的提交物件。如果你把原來分支中的提交物件發佈出去，並且其他人更新下載後在其基礎上開展工作，而稍後你又用 `git rebase` 拋棄這些提交物件，把新的重演後的提交物件發佈出去的話，你的合作者就不得不重新合併他們的工作，這樣當你再次從他們那裡獲取內容時，提交歷史就會變得一團糟。

下面我們用一個實際例子來說明為什麼公開的衍合會帶來問題。假設你從一個中央伺服器克隆然後在它的基礎上搞了一些開發，提交歷史類似圖 3-36 所示：

git.team1.ourcompany.com



My Computer

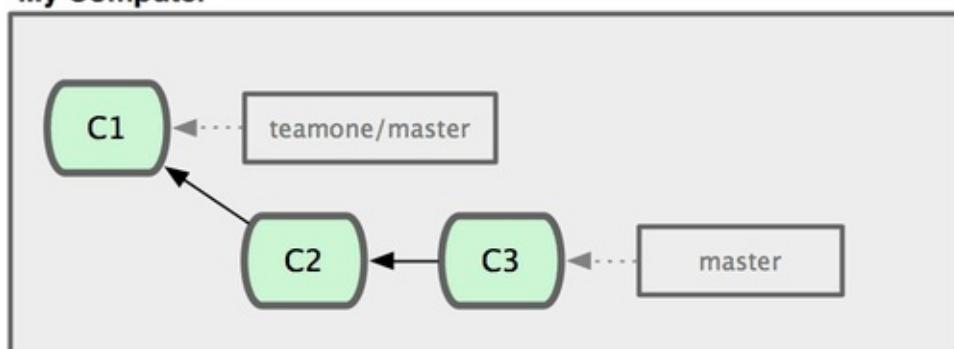
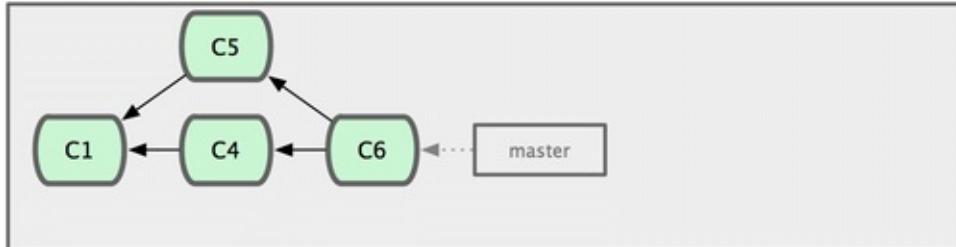


圖 3-36. 克隆一個倉庫，在其基礎上工作一番。

現在，某人在 C1 的基礎上做了些改變，併合並他自己的分支得到結果 C6，推送到中央伺服器。當你抓取併合並這些資料到你本地的開發分支中後，會得到合併結果 C7，歷史提交會變成圖 3-37 這樣：

git.team1.ourcompany.com



My Computer

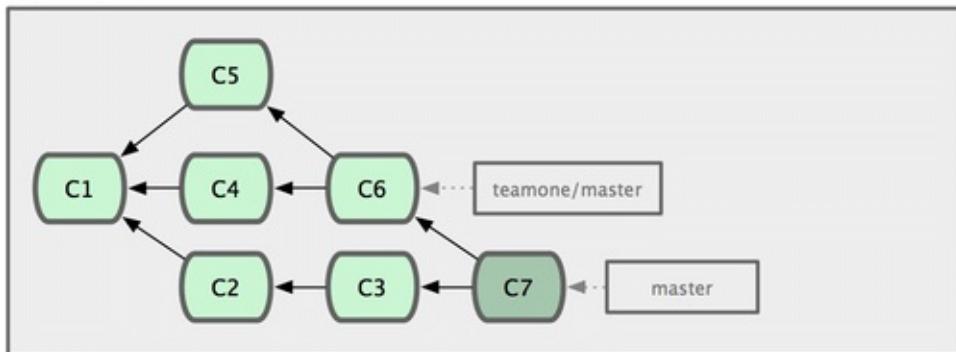
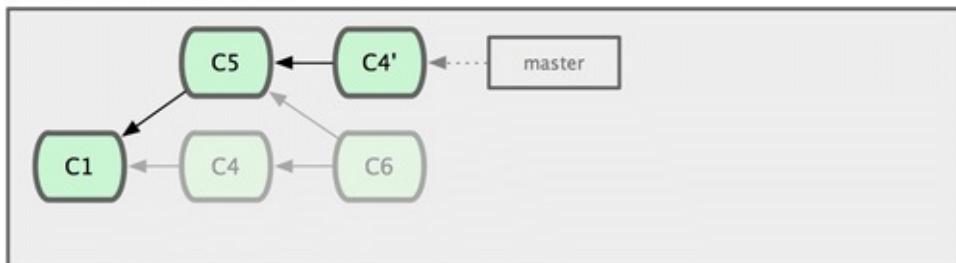


圖 3-37. 抓取他人提交，併入自己主幹。

接下來，那個推送 C6 上來的人決定用衍合取代之前的合併操作；繼而又用 `git push --force` 覆蓋了伺服器上的歷史，得到 C4'。而之後當你再從伺服器上下載最新提交後，會得到：

git.team1.ourcompany.com



My Computer

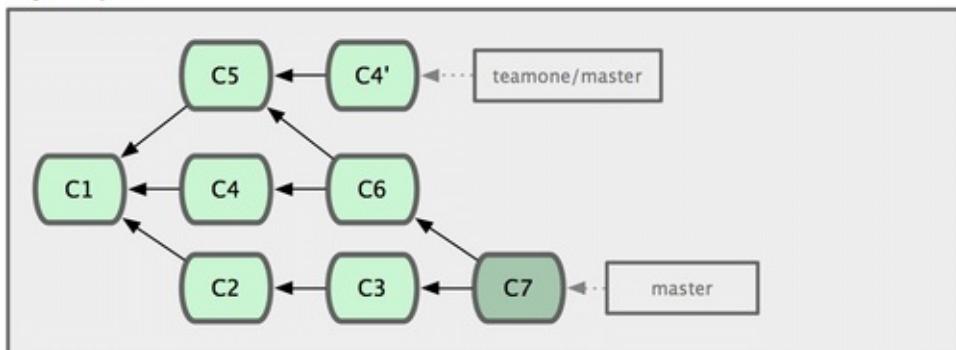


圖 3-38. 有人推送了衍合後得到的 C4'，丟棄了你作為開發基礎的 C4 和 C6。

下載更新後需要合併，但此時衍合產生的提交物件 C4' 的 SHA-1 校驗值和之前 C4 完全不同，所以 Git 會把它們當作新的提交物件處理，而實際上此刻你的提交歷史 C7 中早已經包含了 C4 的修改內容，於是合併操作會把 C7 和 C4' 合併為 C8（見圖 3-39）：

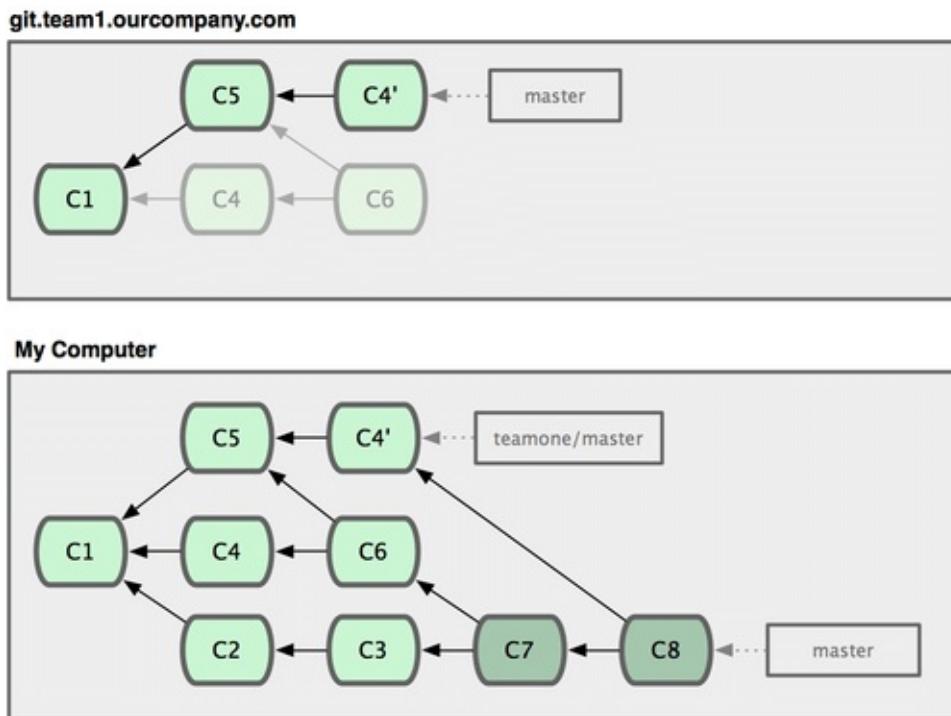


圖 3-39. 你把相同的内容又合併了一遍，生成一個新的提交 C8。

C8 這一步的合併是遲早會發生的，因為只有這樣你才能和其他協作者提交的內容保持同步。而在 C8 之後，你的提交歷史裡就會同時包含 C4 和 C4'，兩者有著不同的 SHA-1 校驗值，如果用 `git log` 查看歷史，會看到兩個提交擁有相同的作者日期與說明，令人費解。而更糟的是，當你把這樣的歷史推送到伺服器後，會再次把這些衍合後的提交引入到中央伺服器，進一步困擾其他人（譯注：這個例子中，出問題的責任方是那個發佈了 C6 後又用衍合發佈 C4' 的人，其他人會因此回饋雙重歷史到共用主幹，從而混淆大家的視聽。）。

如果把衍合當成一種在推送之前清理提交歷史的手段，而且僅僅衍合那些尚未公開的提交物件，就沒問題。如果衍合那些已經公開的提交物件，並且已經有人基於這些提交物件開展了後續開發工作的話，就會出現叫人沮喪的麻煩。

小結

讀到這裡，你應該已經學會了如何創建分支並切換到新分支，在不同分支間轉換，合併本地分支，把分支推送到共用伺服器上，使用共用分支與他人協作，以及在分享之前進行衍合。

伺服器上的 Git

到目前為止，你應該已經學會了使用 Git 來完成日常工作。然而，如果想與他人合作，還需要一個遠端的 Git 倉庫。儘管技術上可以從個人的倉庫裡推送和拉取修改內容，但我們不鼓勵這樣做，因為一不留心就很容易弄混其他人的進度。另外，你也一定希望合作者們即使在自己不開機的時候也能從倉庫獲取資料 — 擁有一個更穩定的公共倉庫十分有用。因此，更好的合作方式是建立一個大家都可以訪問的共用倉庫，從那裡推送和拉取資料。我們將把這個倉庫稱為 "Git 伺服器"；代理一個 Git 倉庫只需要花費很少的資源，幾乎從不需要整個伺服器來支援它的運行。

架設一台 Git 伺服器並不難。第一步是選擇與伺服器通訊的協定。本章第一節將介紹可用的協議以及各自優缺點。下面一節將介紹一些針對各個協議典型的設置以及如何在伺服器上實施。最後，如果你不介意在他人伺服器上保存你的代碼，又想免去自己架設和維護伺服器的麻煩，倒可以試試我們介紹的幾個倉庫託管服務。

如果你對架設自己的伺服器沒興趣，可以跳到本章最後一節去看看如何申請一個代碼託管服務的帳戶然後繼續下一章，我們會在那裡討論分散式源碼控制環境的林林總總。

遠端倉庫通常只是一個裸倉庫 (*bare repository*) — 即一個沒有當前工作目錄的倉庫。因為該倉庫只是一個合作媒介，所以不需要從硬碟上取出最新版本的快照；倉庫裡存放的僅僅是 Git 的資料。簡單地說，裸倉庫就是你工作目錄中 `.git` 子目錄內的內容。

協議

Git 可以使用四種主要的協定來傳輸資料：本地傳輸，SSH 協定，Git 協定和 HTTP 協定。下面分別介紹一下哪些情形應該使用（或避免使用）這些協定。

值得注意的是，除了 HTTP 協定外，其他所有協定都要求在伺服器端安裝並運行 Git。

本地協定

最基本的就是本地協議（*Local protocol*），所謂的遠端倉庫在該協定中的表示，就是硬碟上的另一個目錄。這常見於團隊每一個成員都對一個共用的檔案系統（例如 NFS）擁有訪問權，或者比較少見的多人共用同一台電腦的情況。後面一種情況並不安全，因為所有代碼倉庫實例都儲存在同一台電腦裡，增加了災難性資料損失的可能性。

如果你使用一個共用的檔案系統，就可以在一個本地檔案系統中克隆倉庫，推送和獲取。克隆的時候只需要將遠端倉庫的路徑作為 URL 使用，比如下面這樣：

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

或者這樣：

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

如果在 URL 開頭明確使用 `file://`，那麼 Git 會以一種略微不同的方式運行。如果你只給出路徑，Git 會嘗試使用硬連結或直接複製它所需要的檔。如果使用了 `file://`，Git 會調用它平時通過網路來傳輸資料的工序，而這種方式的效率相對較低。使用 `file://` 首碼的主要原因是當你需要一個不包含無關引用或物件的乾淨倉庫副本的時候 — 一般指從其他版本控制系統導入的，或類似情形（參見第 9 章的維護任務）。我們這裡僅僅使用普通路徑，這樣更快。

要添加一個本地倉庫作為現有 Git 項目的遠端倉庫，可以這樣做：

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

然後就可以像在網路上一樣向這個遠端倉庫推送和獲取資料了。

優點

基於檔倉庫的優點在於它的簡單，同時保留了現存檔的許可權和網路存取權限。如果你的團隊已經有一個全體共用的檔案系統，建立倉庫就十分容易了。你只需把一份裸倉庫的副本放在大家都能訪問的地方，然後像對其他共用目錄一樣設置讀寫許可權就可以了。我們將在下一節“在伺服器上部署 Git”中討論如何匯出一個裸倉庫的副本。

這也是從別人工業目錄中獲取工作成果的快捷方法。假如你和你的同事在一個項目中合作，他們想讓你檢出一些東西的時候，運行類似 `git pull /home/john/project` 通常會比他們推送到伺服器，而你再從伺服器獲取簡單得多。

缺點

這種方法的缺點是，與基本的網路連接訪問相比，難以控制從不同位置來的存取權限。如果你想從家裡的筆記型電腦上推送，就要先掛載遠端硬碟，這和基於網路連接的訪問相比更加困難和緩慢。

另一個很重要的問題是該方法不一定就是最快的，尤其是對於共用掛載的檔案系統。本地倉庫只有在你對資料存取速度快的時候才快。在同一個伺服器上，如果二者同時允許 Git 訪問本地硬碟，通過 NFS 訪問倉庫通常會比 SSH 慢。

SSH 協議

Git 使用的傳輸協議中最常見的可能就是 SSH 了。這是因為大多數環境已經支持通過 SSH 對伺服器的訪問 — 即便還沒有，架設起來也很容易。SSH 也是唯一一個同時支持讀寫操作的網路通訊協定。另外兩個網路通訊協定（HTTP 和 Git）通常都是唯讀的，所以雖然二者對大多數人都可用，但執行寫操作時還是需要 SSH。SSH 同時也是一個驗證授權的網路通訊協定；而因為其普遍性，一般架設和使用都很容易。

通過 SSH 克隆一個 Git 倉庫，你可以像下面這樣給出 `ssh://` 的 URL：

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

或者不指明某個協定 — 這時 Git 會預設使用 SSH：

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

如果不指明用戶，Git 會默認使用當前登錄的用戶名連接伺服器。

優點

使用 SSH 的好處有很多。首先，如果你想擁有對網路倉庫的寫許可權，基本上不可能不使用 SSH。其次，SSH 架設相對比較簡單 — SSH 守護進程很常見，很多網路系統管理員都有一些使用經驗，而且很多作業系統都自帶了它或者相關的管理工具。再次，通過 SSH 進行訪問是安全的 — 所有資料傳輸都是加密和授權的。最後，和 Git 及本地協議一樣，SSH 也很高效，會在傳輸之前盡可能壓縮資料。

缺點

SSH 的限制在於你不能通過它實現倉庫的匿名訪問。即使僅為讀取資料，人們也必須在能通過 SSH 訪問主機的前提下才能訪問倉庫，這使得 SSH 不利於開源的項目。如果你僅僅在公司網路裡使用，SSH 可能是你唯一需要使用的協定。如果想允許對項目的匿名唯讀訪問，那麼除了為自己推送而架設 SSH 協定之外，還需要支援其他協定以便他人訪問讀取。

Git 協議

接下來是 Git 協議。這是一個包含在 Git 套裝軟體中的特殊守護進程；它會監聽一個提供類似於 SSH 服務的特定埠（9418），而無需任何授權。打算支援 Git 協定的倉庫，需要先創建 `git-daemon-export-ok` 檔 — 它是協定進程提供倉庫服務的必要條件 — 但除此之外該服務沒有什麼安全措施。要麼所有人都能克隆 Git 倉庫，要麼誰也不能。這也意味著該協議通常不能用來進行推送。你可以允許推送操作；然而由於沒有授權機制，一旦允許該操作，網路上任何一個知道專案 URL 的人將都有推送許可權。不用說，這是十分罕見的情況。

優點

Git 協定是現存最快的傳輸協議。如果你在提供一個有很大訪問量的公共專案，或者一個不需要對讀操作進行授權的龐大項目，架設一個 Git 守護進程來供應倉庫是個不錯的選擇。它使用與 SSH 協定相同的資料傳輸機制，但省去了加密和授權的開銷。

缺點

Git 協議消極的一面是缺少授權機制。用 Git 協議作為訪問專案的唯一方法通常是不可取的。一般的做法是，同時提供 SSH 介面，讓幾個開發者擁有推送（寫）許可權，其他人通過 `git://` 擁有唯讀許可權。Git 協定可能也是最難架設的協議。它要求有單獨的守護進程，需要定制 — 我們將在本章的 “*Gitosis*” 一節詳細介紹它的架設 — 需要設定 `xinetd` 或類似的程式，而這些工作就沒那麼輕鬆了。該協議還要求防火牆開放 9418 埠，而企業級防火牆一般不允許對這個非標準埠的訪問。大型企業級防火牆通常會封鎖這個少見的埠。

HTTP/S 協議

最後還有 HTTP 協議。HTTP 或 HTTPS 協議的優美之處在於架設的簡便性。基本上，只需要把 Git 的裸倉庫檔放在 HTTP 的根目錄下，配置一個特定的 `post-update` 掛鉤（hook）就可以搞定（Git 掛鉤的細節見第 7 章）。此後，每個能訪問 Git 倉庫所在伺服器上 web 服務的人都可以進行克隆操作。下面的操作可以允許通過 HTTP 對倉庫進行讀取：

```
$ 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
```

這樣就可以了。Git 附帶的 `post-update` 掛鉤會默認運行合適的命令（`git update-server-info`）來確保通過 HTTP 的獲取和克隆正常工作。這條命令在你用 SSH 向倉庫推送內容時運行；之後，其他人就可以用下面的命令來克隆倉庫：

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

在本例中，我們使用了 Apache 設定中常用的 `/var/www/htdocs` 路徑，不過你可以使用任何靜態 web 服務 — 把裸倉庫放在它的目錄裡就行。Git 的資料是以最基本的靜態檔的形式提供的（關於如何提供檔的詳情見第 9 章）。

通過 HTTP 進行推送操作也是可能的，不過這種做法不太常見，並且牽扯到複雜的 WebDAV 設定。由於很少用到，本書將略過對該內容的討論。如果對 HTTP 推送協議感興趣，不妨打開這個位址看一下操作方法：<http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt>。通過 HTTP 推送的好處之一是你可以使用任何 WebDAV 伺服器，不需要為 Git 設定特殊環境；所以如果主機提供商支援通過 WebDAV 更新網站內容，你也可以使用這項功能。

優點

使用 HTTP 協定的好處是易於架設。幾條必要的命令就可以讓全世界讀取到倉庫的內容。花費不過幾分鐘。HTTP 協議不會佔用過多伺服器資源。因為它一般只用到靜態的 HTTP 服務提供所有資料，普通的 Apache 伺服器平均每秒能支撐數千個檔的併發訪問 — 哪怕讓一個小型伺服器超載都很難。

你也可以通過 HTTPS 提供唯讀的倉庫，這意味著你可以加密傳輸內容；你甚至可以要求用戶端使用特定簽名的 SSL 證書。一般情況下，如果到了這一步，使用 SSH 公共金鑰可能是更簡單的方案；不過也存在一些特殊情況，這時通過 HTTPS 使用帶簽名的 SSL 證書或者其他基於 HTTP 的唯讀連接授權方式是更好的解決方案。

HTTP 還有個額外的好處：HTTP 是一個如此常見的協議，以至於企業級防火牆通常都允許其埠的通信。

缺點

HTTP 協議的消極面在於，相對來說用戶端效率更低。克隆或者下載倉庫內容可能會花費更多時間，而且 HTTP 傳輸的體積和網路開銷比其他任何一個協議都大。因為它沒有按需供應的能力 — 傳輸過程中沒有服務端的動態計算 — 因而 HTTP 協定經常會被稱為傻瓜 (*dumb*) 協議。更多 HTTP 協定和其他協定效率上的差異見第 9 章。

在伺服器上部署 Git

開始架設 Git 伺服器前，需要先把現有倉庫匯出為裸倉庫 — 即一個不包含當前工作目錄的倉庫。做法直截了當，克隆時用 `--bare` 選項即可。裸倉庫的目錄名一般以 `.git` 結尾，像這樣：

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

該命令的輸出或許會讓人有些不解。其實 `clone` 操作基本上相當於 `git init` 加 `git fetch`，所以這裡出現的其實是 `git init` 的輸出，先由它建立一個空目錄，而之後傳輸資料物件的操作並無任何輸出，只是悄悄在幕後執行。現在 `my_project.git` 目錄中已經有了一份 Git 目錄資料的副本。

整體上的效果大致相當於：

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

但在設定檔中有若干小改動，不過對使用者來講，使用方式都一樣，不會有什麼影響。它僅取出 Git 倉庫的必要原始資料，存放在該目錄中，而不會另外創建工作目錄。

把裸倉庫移到伺服器上

有了裸倉庫的副本後，剩下的就是把它放到伺服器上並設定相關協定。假設一個功能變數名稱為 `git.example.com` 的伺服器已經架設好，並可以通過 SSH 訪問，我們打算把所有 Git 倉庫儲存在 `/opt/git` 目錄下。只要把裸倉庫複製過去：

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

現在，所有對該伺服器有 SSH 存取權限，並可讀取 `/opt/git` 目錄的使用者都可以用下面的命令克隆該項目：

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

如果某個 SSH 用戶對 `/opt/git/my_project.git` 目錄有寫許可權，那他就有推送許可權。如果到該專案目錄中運行 `git init` 命令，並加上 `--shared` 選項，那麼 Git 會自動修改該倉庫目錄的組許可權為可寫（譯注：實際上 `--shared` 可以指定其他行為，只是默認為將組許可

權改為可寫並執行 `g+sx`，所以最後會得到 `rws`。）。

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

由此可見，根據現有的 Git 倉庫創建一個裸倉庫，然後把它放上你和同事都有 SSH 訪問權的伺服器是多麼容易。現在已經可以開始在同一專案上密切合作了。

值得注意的是，這的確是架設一個少數人具有連接權的 Git 服務的全部 — 只要在伺服器上加入可以用 SSH 登錄的帳號，然後把裸倉庫放在大家都有讀寫許可權的地方。一切都準備停當，無需更多。

下面的幾節中，你會瞭解如何擴展到更複雜的設定。這些內容包含如何避免為每一個用戶建立一個帳戶，給倉庫添加公共讀取許可權，架設網頁介面，使用 Gitosis 工具等等。然而，只是和幾個人在一個不公開的專案上合作的話，僅僅是一個 SSH 伺服器和裸倉庫就足夠了，記住這點就可以了。

小型安裝

如果設備較少或者你只想在小型開發團隊裡嘗試 Git，那麼一切都很簡單。架設 Git 服務最複雜的地方在於帳戶管理。如果需要倉庫對特定的使用者可讀，而給另一部分用戶讀寫許可權，那麼訪問和許可的安排就比較困難。

SSH 連接

如果已經有了一個所有開發成員都可以用 SSH 訪問的伺服器，架設第一個伺服器將變得異常簡單，幾乎什麼都不用做（正如上節中介紹的那樣）。如果需要對倉庫進行更複雜的存取控制，只要使用伺服器作業系統的本地檔訪問許可機制就行了。

如果需要團隊裡的每個人都對倉庫有寫許可權，又不能給每個人在伺服器上建立帳戶，那麼提供 SSH 連接就是唯一的選擇了。我們假設用來共用倉庫的伺服器已經安裝了 SSH 服務，而且你通過它訪問伺服器。

有好幾個辦法可以讓團隊的每個人都有訪問權。第一個辦法是給每個人建立一個帳戶，直截了當但略過繁瑣。反復運行 `adduser` 並給所有人設定臨時密碼可不是好玩的。

第二個辦法是在主機上建立一個 `git` 帳戶，讓每個需要寫許可權的人發送一個 SSH 公鑰，然後將其加入 `git` 帳戶的 `~/.ssh/authorized_keys` 文件。這樣一來，所有人都將通過 `git` 帳戶訪問主機。這絲毫不會影響提交的資料 — 訪問主機用的身份不會影響提交物件的提交者資訊。

另一個辦法是讓 SSH 伺服器通過某個 LDAP 服務，或者其他已經設定好的集中授權機制，來進行授權。只要每個人都能獲得主機的 shell 訪問權，任何可用的 SSH 授權機制都能達到相同效果。

生成 SSH 公開金鑰

大多數 Git 伺服器都會選擇使用 SSH 公開金鑰來進行授權。系統中的每個使用者都必須提供一個公開金鑰用於授權，沒有的話就要生成一個。生成公開金鑰的過程在所有作業系統上都差不多。首先先確認一下是否已經有一個公開金鑰了。SSH 公開金鑰預設儲存在帳戶的主目錄下的 `~/.ssh` 目錄。進去看看：

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

關鍵是看有沒有用 `something` 和 `something.pub` 來命名的一對檔，這個 `something` 通常就是 `id_dsa` 或 `id_rsa`。有 `.pub` 尾碼的檔就是公開金鑰，另一個檔則是金鑰。假如沒有這些檔，或者乾脆連 `.ssh` 目錄都沒有，可以用 `ssh-keygen` 來創建。該程式在 Linux/Mac 系統上由 SSH 包提供，而在 Windows 上則包含在 MSysGit 包裡：

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

它先要求你確認保存公開金鑰的位置（`.ssh/id_rsa`），然後它會讓你重複一個密碼兩次，如果不想在使用公開金鑰的時候輸入密碼，可以留空。

現在，所有做過這一步的用戶都得把它們的公開金鑰給你或者 Git 伺服器的管理員（假設 SSH 服務被設定為使用公開金鑰機制）。他們只需要複製 `.pub` 檔的內容然後發郵件給管理員。公開金鑰的樣子大致如下：

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BwDSU
GPL+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrvQzM7x1ELEVf4h91FX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnPnTP189ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraT1MqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

關於在多個作業系統上設立相同 SSH 公開金鑰的教程，可以查閱 GitHub 上有關 SSH 公開金鑰的嚮導：<http://github.com/guides/providing-your-ssh-key>。

架設伺服器

現在我們過一邊伺服器端架設 SSH 訪問的流程。本例將使用 `authorized_keys` 方法來給用戶授權。我們還將假定使用類似 Ubuntu 這樣的標準 Linux 發行版本。首先，創建一個名為 'git' 的用戶，並為其創建一個 `.ssh` 目錄。

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

接下來，把開發者的 SSH 公開金鑰添加到這個用戶的 `authorized_keys` 檔中。假設你通過電子郵件收到了幾個公開金鑰並存到了暫存檔案裡。重複一下，公開金鑰大致看起來是這個樣子：

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLM9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpgW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtpofwFB1gc+myiv
07TCUSBdLQlgMV0Fq1I2uPWQ0k0WQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgtzg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

只要把它們逐個追加到 `authorized_keys` 檔案結尾部即可：

```
$ 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
```

現在可以用 `--bare` 選項運行 `git init` 來建立一個裸倉庫，這會初始化一個不包含工作目錄的倉庫。

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

這時，Join，Josie 或者 Jessica 就可以把它加為遠端倉庫，推送一個分支，從而把第一個版本的專案檔案上傳到倉庫裡了。值得注意的是，每次添加一個新專案都需要通過 shell 登入主機並創建一個裸倉庫目錄。我們不妨以 `gitserver` 作為 `git` 用戶及項目倉庫所在的主機名稱。如果在網路內部運行該主機，並在 DNS 中設定 `gitserver` 指向該主機，那麼以下這些命令都是可用的：

```
# 在 John 的電腦上
$ 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
```

這樣，其他人的克隆和推送也一樣變得很簡單：

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

用這個方法可以很快捷地為少數幾個開發者架設一個可讀寫的 Git 服務。

作為一個額外的防範措施，你可以用 Git 自帶的 `git-shell` 工具限制 `git` 用戶的活動範圍。只要把它設為 `git` 用戶登入的 `shell`，那麼該用戶就無法使用普通的 `bash` 或者 `csh` 什麼的 `shell` 程式。編輯 `/etc/passwd` 檔：

```
$ sudo vim /etc/passwd
```

在檔末尾，你應該能找到類似這樣的行：

```
git:x:1000:1000::/home/git:/bin/sh
```

把 `bin/sh` 改為 `/usr/bin/git-shell`（或者用 `which git-shell` 查看它的實際安裝路徑）。該行修改後的樣子如下：

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

現在 `git` 用戶只能用 SSH 連接來推送和獲取 Git 倉庫，而不能直接使用主機 `shell`。嘗試普通 SSH 登錄的話，會看到下面這樣的拒絕資訊：

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

公共訪問

匿名的讀取許可權該怎麼實現呢？也許除了內部私有的專案之外，你還需要託管一些開源專案。或者因為要用一些自動化的伺服器來進行編譯，或者有一些經常變化的伺服器群組，而又不想整天生成新的 SSH 金鑰 — 總之，你需要簡單的匿名讀取許可權。

或許對小型的配置來說最簡單的辦法就是運行一個靜態 web 服務，把它的根目錄設定為 Git 倉庫所在的位置，然後開啓本章第一節提到的 `post-update` 掛鉤。這裡繼續使用之前的例子。假設倉庫處於 `/opt/git` 目錄，主機上運行著 Apache 服務。重申一下，任何 web 服務程式都可以達到相同效果；作為範例，我們將用一些基本的 Apache 設定來展示大體需要的步驟。

首先，開啓掛鉤：

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

`post-update` 掛鉤是做什麼的呢？其內容大致如下：

```
$ cat .git/hooks/post-update
#!/bin/sh
#
# An example hook script to prepare a packed repository for use over
# dumb transports.
#
# To enable this hook, rename this file to "post-update".
#
exec git-update-server-info
```

意思是當通過 SSH 向伺服器推送時，Git 將運行這個 `git-update-server-info` 命令來更新匿名 HTTP 訪問獲取資料時所需要的檔。

接下來，在 Apache 設定檔中添加一個 `VirtualHost` 條目，把文檔根目錄設為 Git 專案所在的根目錄。這裡我們假定 DNS 服務已經配置好，會把對 `.gitserver` 的請求發送到這台主機：

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

另外，需要把 `/opt/git` 目錄的 Unix 使用者組設定為 `www-data`，這樣 web 服務才可以讀取倉庫內容，因為運行 CGI 腳本的 Apache 實例進程預設就是以該使用者的身份起來的：

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

重啓 Apache 之後，就可以通過專案的 URL 來克隆該目錄下的倉庫了。

```
$ git clone http://git.gitserver/project.git
```

這一招可以讓你在幾分鐘內為相當數量的用戶架設好基於 HTTP 的讀取許可權。另一個提供非授權訪問的簡單方法是開啓一個 Git 守護進程，不過這將要求該進程作為後臺進程常駐——接下來的這一節就要討論這方面的細節。

GitWeb

現在我們的項目已經有了可讀可寫和唯讀的連接方式，不過如果能有一個簡單的 web 介面訪問就更好了。Git 自帶一個叫做 GitWeb 的 CGI 腳本，運行效果可以到 <http://git.kernel.org> 這樣的網站體驗下（見圖 4-1）。

The screenshot shows the GitWeb interface for the 'git' repository at <http://git.kernel.org/pub/scm/git/git.git>. The top navigation bar includes links for 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. A search bar is also present. The main content area displays the repository's metadata (description, owner, last change, URL) and a detailed 'shortlog' of recent commits. The commits are listed with their author, date, subject, and a link to the full commit details.

Date	Author	Subject	Links
33 hours ago	Junio C Hamano	Merge branch 'maint' master	commit commitdiff tree snapshot
34 hours ago	Matthieu Moy	More friendly message when locking the index fails. maint	commit commitdiff tree snapshot
34 hours ago	Matthieu Moy	Document git blame --reverse.	commit commitdiff tree snapshot
34 hours ago	Marcel M. Cary	gitweb: Hyperlink multiple git hashes on the same commi ...	commit commitdiff tree snapshot
34 hours ago	Johannes Schindelin	system_path(): simplify using strip_path_suffix(), ...	commit commitdiff tree snapshot
34 hours ago	Johannes Schindelin	Introduce the function strip_path_suffix()	commit commitdiff tree snapshot
2 days ago	Todd Zullinger	Documentation: Note file formats send-email accepts	commit commitdiff tree snapshot
2 days ago	Junio C Hamano	Merge branch 'maint'	commit commitdiff tree snapshot
2 days ago	Junio C Hamano	tests: fix "export var=val"	commit commitdiff tree snapshot
2 days ago	Lars Noschinski	filter-branch -d: Export GIT_DIR earlier	commit commitdiff tree snapshot
2 days ago	Jay Soffian	disallow providing multiple upstream branches to rebase ...	commit commitdiff tree snapshot
2 days ago	Michael Spang	Skip timestamp differences for diff --no-index	commit commitdiff tree snapshot
2 days ago	Junio C Hamano	git-svn: fix parsing of timestamp obtained from svn	commit commitdiff tree snapshot
2 days ago	Marcel M. Cary	gitweb: Fix warnings with override permitted but no ...	commit commitdiff tree snapshot
2 days ago	Gerrit Pape	Documentation/git-push: --all, --mirror, --tags can ...	commit commitdiff tree snapshot
2 days ago	Thomas Rast	bash completion: only show 'log --merge' if merging	commit commitdiff tree snapshot

Figure 4-1. 基於網頁的 GitWeb 使用者介面

如果想看看自己項目的效果，不妨用 Git 自帶的一個命令，可以使用類似 `lighttpd` 或 `webrick` 這樣羽量級的伺服器啓動一個臨時進程。如果是在 Linux 主機上，通常都預裝了 `lighttpd`，可以到專案目錄中鍵入 `git instaweb` 來啓動。如果用的是 Mac，Leopard 預裝了 Ruby，所以 `webrick` 應該是最好的選擇。如果要用 `lighttpd` 以外的程式來啓動 `git instaweb`，可以通過 `--httpd` 選項指定：

```
$ 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]
```

這會在 1234 埠開啓一個 HTTPD 服務，隨之在流覽器中顯示該頁，十分簡單。關閉服務時，只需在原來的命令後面加上 `--stop` 選項就可以了：

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

如果需要為團隊或者某個開源專案長期運行 GitWeb，那麼 CGI 腳本就要由正常的網頁服務來運行。一些 Linux 發行版本可以通過 `apt` 或 `yum` 安裝一個叫做 `gitweb` 的套裝軟體，不妨首先嘗試一下。我們將快速介紹一下手動安裝 GitWeb 的流程。首先，你需要 Git 的源碼，其中帶有 GitWeb，並能生成定制的 CGI 腳本：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
    prefix=/usr gitweb
$ sudo cp -Rf gitweb /var/www/
```

注意，通過指定 `GITWEB_PROJECTROOT` 變數告訴編譯命令 Git 倉庫的位置。然後，設置 Apache 以 CGI 方式運行該腳本，添加一個 VirtualHost 配置：

```
<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>
```

不難想像，GitWeb 可以使用任何相容 CGI 的網頁服務來運行；如果偏向使用其他 web 伺服器，配置也不會很麻煩。現在，通過 `http://gitserver` 就可以線上訪問倉庫了，在 `http://git.server` 上還可以通過 HTTP 克隆和獲取倉庫的內容。

Gitosis

把所有用戶的公開金鑰保存在 `authorized_keys` 檔的做法，只能湊和一陣子，當用戶數量達到幾百人的規模時，管理起來就會十分痛苦。每次改刪用戶都必須登錄伺服器不去說，這種做法還缺少必要的許可權管理 — 每個人都對所有專案擁有完整的讀寫許可權。

幸好我們還可以選擇應用廣泛的 **Gitosis** 項目。簡單地說，**Gitosis** 就是一套用來管理 `authorized_keys` 檔和實現簡單連接限制的腳本。有趣的是，用來添加用戶和設定許可權的並非通過網頁程式，而只是管理一個特殊的 Git 倉庫。你只需要在這個特殊倉庫內做好相應的設定，然後推送到伺服器上，**Gitosis** 就會隨之改變運行策略，聽起來就很酷，對吧？

Gitosis 的安裝算不上傻瓜化，但也不算太難。用 Linux 伺服器架設起來最簡單 — 以下例子中，我們使用裝有 Ubuntu 8.10 系統的伺服器。

Gitosis 的工作依賴於某些 Python 工具，所以首先要安裝 Python 的 `setuptools` 包，在 Ubuntu 上稱為 `python-setuptools`：

```
$ apt-get install python-setuptools
```

接下來，從 **Gitosis** 項目主頁克隆並安裝：

```
$ git clone https://github.com/tv42/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

這會安裝幾個供 **Gitosis** 使用的工具。預設 **Gitosis** 會把 `/home/git` 作為存儲所有 Git 倉庫的根目錄，這沒什麼不好，不過我們之前已經把項目倉庫都放在 `/opt/git` 裡面了，所以為方便起見，我們可以做一個符號連接，直接劃轉過去，而不必重新配置：

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis 將會幫我們管理用戶公開金鑰，所以先把當前控制檔改名備份，以便稍後重新添加，準備好讓 **Gitosis** 自動管理 `authorized_keys` 文件：

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

接下來，如果之前把 `git` 用戶的登錄 shell 改為 `git-shell` 命令的話，先恢復 '`git`' 用戶的登錄 shell。改過之後，大家仍然無法通過該帳號登錄（譯注：因為 `authorized_keys` 檔已經沒有了。），不過不用擔心，這會交給 **Gitosis** 來實現。所以現在先打開 `/etc/passwd` 檔，把

這行：

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

改回：

```
git:x:1000:1000::/home/git:/bin/sh
```

好了，現在可以初始化 Gitosis 了。你可以用自己的公開金鑰執行 `gitosis-init` 命令，要是公開金鑰不在伺服器上，先臨時複製一份：

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

這樣該公開金鑰的擁有者就能修改用於配置 Gitosis 的那個特殊 Git 倉庫了。接下來，需要手工對該倉庫中的 `post-update` 腳本加上可執行許可權：

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

基本上就算是好了。如果設定過程沒出什麼差錯，現在可以試一下用初始化 Gitosis 的公開金鑰的擁有者身份 SSH 登錄伺服器，應該會看到類似下面這樣：

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
ERROR:gitosis.serve.main:Need SSH_ORIGINAL_COMMAND in environment.
Connection to gitserver closed.
```

說明 Gitosis 認出了該用戶的身份，但由於沒有運行任何 Git 命令，所以它切斷了連接。那麼，現在運行一個實際的 Git 命令 — 克隆 Gitosis 的控制倉庫：

```
# 在你本地電腦上
$ git clone git@gitserver:gitosis-admin.git
```

這會得到一個名為 `gitosis-admin` 的工作目錄，主要由兩部分組成：

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

`gitosis.conf` 檔是用來設置用戶、倉庫和許可權的控制檔。`keydir` 目錄則是保存所有具有存取權限用戶公開金鑰的地方— 每人一個。在 `keydir` 裡的檔案名（比如上面的 `scott.pub`）應該跟你的不一樣 — Gitosis 會自動從使用 `gitosis-init` 腳本導入的公開金鑰尾部的描述中獲取該名字。

看一下 `gitosis.conf` 檔的內容，它應該只包含與剛剛克隆的 `gitosis-admin` 相關的資訊：

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
members = scott
writable = gitosis-admin
```

它顯示使用者 `scott` — 初始化 Gitosis 公開金鑰的擁有者 — 是唯一能管理 `gitosis-admin` 專案的人。

現在我們來添加一個新項目。為此我們要建立一個名為 `mobile` 的新段落，在其中羅列手機開發團隊的開發者，以及他們擁有寫許可權的專案。由於 'scott' 是系統中的唯一使用者，我們把他設為唯一用戶，並允許他讀寫名為 `iphone_project` 的新項目：

```
[group mobile]
members = scott
writable = iphone_project
```

修改完之後，提交 `gitosis-admin` 裡的改動，並推送到伺服器使其生效：

```
$ git commit -am 'add iphone_project and mobile group'
[master 8962da8] add iphone_project and mobile group
 1 file changed, 4 insertions(+)
$ git push origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 272 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:gitosis-admin.git
 fb27aec..8962da8  master -> master
```

在新工程 `iphone_project` 裡首次推送資料到伺服器前，得先設定該伺服器地址為遠端倉庫。但你不用事先到伺服器上手工創建該項目的裸倉庫— Gitosis 會在第一次遇到推送時自動創建：

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
 * [new branch]      master -> master
```

請注意，這裡不用指明完整路徑（實際上，如果加上反而沒用），只需要一個冒號加項目名字即可 — Gitosis 會自動幫你映射到實際位置。

要和朋友們在一個專案上協同工作，就得重新添加他們的公開金鑰。不過這次不用在伺服器上一個一個手工添加到 `~/.ssh/authorized_keys` 檔末端，而只需管理 `keydir` 目錄中的公開金鑰檔。文件的命名將決定在 `gitosis.conf` 中對使用者的標識。現在我們為 John，Josie 和 Jessica 添加公開金鑰：

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

然後把他們都加進 'mobile' 團隊，讓他們對 `iphone_project` 具有讀寫許可權：

```
[group mobile]
members = scott john josie jessica
writable = iphone_project
```

如果你提交並推送這個修改，四個用戶將同時具有該項目的讀寫許可權。

Gitosis 也具有簡單的存取控制功能。如果想讓 John 只有讀許可權，可以這樣做：

```
[group mobile]
members = scott josie jessica
writable = iphone_project

[group mobile_ro]
members = john
readonly = iphone_project
```

現在 John 可以克隆和獲取更新，但 Gitosis 不會允許他向專案推送任何內容。像這樣的組可以隨意創建，多少不限，每個都可以包含若干不同的用戶和項目。甚至還可以指定某個組為成員之一（在組名前加上 `@` 首碼），自動繼承該組的成員：

```
[group mobile_committers]
members = scott josie jessica

[group mobile]
members = @mobile_committers
writable = iphone_project

[group mobile_2]
members = @mobile_committers john
writable = another_iphone_project
```

如果遇到意外問題，試試看把 `loglevel=DEBUG` 加到 `[gitosis]` 的段落（譯注：把日誌設置為調試級別，記錄更詳細的運行資訊。）。如果一不小心搞錯了配置，失去了推送許可權，也可以手工修改伺服器上的 `/home/git/.gitosis.conf` 檔 — **Gitosis** 實際是從該檔讀取資訊的。它在得到推送資料時，會把新的 `gitosis.conf` 存到該路徑上。所以如果你手工編輯該檔的話，它會一直保持到下次向 `gitosis-admin` 推送新版本的配置內容為止。

Gitolite

This section serves as a quick introduction to Gitolite, and provides basic installation and setup instructions. 不能完全替代隨 gitolite 自帶的大量文檔。There may also be occasional changes to this section itself, so you may also want to look at the latest version [here](#).

Gitolite is an authorization layer on top of Git, relying on `sshd` or `httpd` for authentication. (Recap: authentication is identifying who the user is, authorization is deciding if he is allowed to do what he is attempting to).

Gitolite 允許你定義訪問許可而不只作用於倉庫，而同樣於倉庫中的每個branch和tag name。你可以定義確切的人(或一組人)只能push特定的 "refs" (或者branches或者tags)而不是其他人。

安裝

安裝 Gitolite 非常簡單，你甚至不用讀自帶的那一大堆文檔。你需要一個unix伺服器上的帳戶；許多linux變種和solaris 10都已經試過了。你不需要root訪問，假設git，perl，和一個openssh相容的ssh伺服器已經裝好了。在下面的例子裡，我們會用 `git` 帳戶在 `gitserver` 上。

Gitolite 是不同於 "server" 的軟體 -- 通過ssh訪問，而且每個在伺服器上的userid都是一個潛在的 "gitolite host". We will describe the simplest install method in this article; for the other methods please see the documentation.

To begin, create a user called `git` on your server and login to this user. Copy your SSH public key (a file called `~/.ssh/id_rsa.pub` if you did a plain `ssh-keygen` with all the defaults) from your workstation, renaming it to `<yourname>.pub` (we'll use `scott.pub` in our examples). Then run these commands:

```
$ git clone git://github.com/sitaramc/gitolite
$ gitolite/install -ln
    # assumes $HOME/bin exists and is in your $PATH
$ gitolite setup -pk $HOME/scott.pub
```

That last command creates new Git repository called `gitolite-admin` on the server.

Finally, back on your workstation, run `git clone git@gitserver:gitolite-admin`. And you're done! Gitolite has now been installed on the server, and you now have a brand new repository called `gitolite-admin` in your workstation. You administer your Gitolite setup by making changes to this repository and pushing.

定制安裝

默認快速安裝對大多數人都管用，還有一些定制安裝方法如果你用的上的話。Some changes can be made simply by editing the rc file, but if that is not sufficient, there's documentation on customising Gitolite.

設定檔和訪問規則

安裝結束後，你切換到 `gitolite-admin` 倉庫 (放在你的 HOME 目錄) 然後看看都有啥：

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/scott.pub
$ cat conf/gitolite.conf

repo gitolite-admin
    RW+          = scott

repo testing
    RW+          = @all
```

注意 "scott" (之前用 `gl-setup` 命令時候的 `pubkey` 名稱) 有讀寫許可權而且在 `gitolite-admin` 倉庫裡有一個同名的公開金鑰檔。

Adding users is easy. To add a user called "alice", obtain her public key, name it `alice.pub`, and put it in the `keydir` directory of the clone of the `gitolite-admin` repo you just made on your workstation. Add, commit, and push the change, and the user has been added.

gitolite設定檔的語法在 `conf/example.conf` 裡，我們只會提到一些主要的。

你可以給用戶或者倉庫分組。分組名就像一些宏；定義的時候，無所謂他們是工程還是使用者；區別在於你'使用“宏”的時候

```
@oss_repos      = linux perl rakudo git gitolite
@secret_repos   = fenestra pear

@admin          = scott
@interns        = ashok
@engineers      = sitaram dilbert wally alice
@staff          = @admins @engineers @interns
```

你可以控制許可在 "ref" 級別。在下面的例子裡，實習生可以 push "int" branch. 工程師可以 push任何有 "eng-" 開頭的branch，還有refs/tags下面用 "rc" 開頭的後面跟數字的。而且管理員可以隨便改 (包括rewind) 對任何參考名。

```
repo @oss_repos
  RW  int$          = @interns
  RW  eng-          = @engineers
  RW  refs/tags/rc[0-9] = @engineers
  RW+             = @admins
```

在 `RW` or `RW+` 之後的運算式是規則運算式 (regex) 對應著後面的push用的參考名字 (ref)。所以我們叫它 "參考正則" (refex)！當然，一個 refex 可以比這裡表現的更強大，所以如果你對perl的規則運算式不熟的話就不要改過頭。

同樣，你可能猜到了，Gitolite 字頭 `refs/heads/` 是一個便捷句法如果參考正則沒有用 `refs/` 開頭。

一個這個設定檔語法的重要功能是，所有的倉庫的規則不需要在同一個位置。你能報所有普通的東西放在一起，就像上面的對所有 `oss_repos` 的規則那樣，然後建一個特殊的規則對後面的特殊案例，就像：

```
repo gitolite
  RW+          = sitaram
```

那條規則剛剛加入規則集的 `gitolite` 倉庫。

這次你可能會想要知道存取控制規則是如何應用的，我們簡要介紹一下。

在gitolite裡有兩級存取控制。第一是在倉庫級別；如果你已經讀或者寫訪問過了任何在倉庫裡的參考，那麼你已經讀或者寫訪問倉庫了。

第二級，應用只能寫訪問，通過在倉庫裡的 `branch`或者 `tag`。用戶名如果嘗試過訪問 (`w` 或 `+`)，參考名被更新為已知。訪問規則檢查是否出現在設定檔裡，為這個聯合尋找匹配(但是記得參考名是正則匹配的，不是字串匹配的)。如果匹配被找到了，push就成功了。不匹配的訪問會被拒絕。

帶'拒絕'的高級存取控制

目前，我們只看過了許可是 `R`, `RW`, 或者 `RW+` 這樣子的。但是gitolite還允許另外一種許可：`-`，代表 "拒絕"。這個給了你更多的能力，當然也有一點複雜，因為不匹配並不是唯一的拒絕訪問的方法，因此規則的順序變得無關了！

這麼說好了，在前面的情況中，我們想要工程師可以 `rewind` 任意 `branch` 除了 `master` 和 `integ`。這裡是如何做到的

```
RW  master integ      = @engineers
-   master integ      = @engineers
RW+                         = @engineers
```

你再一次簡單跟隨規則從上至下知道你找到一個匹配你的訪問模式的，或者拒絕。非 `rewind` `push` 到 `master` 或者 `integ` 被第一條規則允許。一個 `rewind push` 到那些 `refs` 不匹配第一條規則，掉到第二條，因此被拒絕。任何 `push` (`rewind` 或非 `rewind`) 到參考或者其他 `master` 或者 `integ` 不會被前兩條規則匹配，即被第三條規則允許。

通過改變檔限制 `push`

此外限制用戶 `push` 改變到哪條 `branch` 的，你也可以限制哪個檔他們可以碰的到。比如，可能 `Makefile` (或者其他哪些程式) 真的不能被任何人做任何改動，因為好多東西都靠著它呢，或者如果某些改變剛好不對就會崩潰。你可以告訴 `gitolite`:

```
repo foo
RW                               = @junior_devs @senior_devs
-
-   VREF/NAME/Makefile  = @junior_devs
```

這是一個強力的公能寫在 `conf/example.conf` 裡。

個人分支

`Gitolite` 也支援一個叫 "個人分支" 的功能 (或者叫, "個人分支命名空間") 在合作環境裡非常有用。

在 `git` 世界裡許多代碼交換通過 "`pull`" 請求發生。然而在合作環境裡，委任制的訪問是'絕不'，一個開發者工作站不能認證，你必須 `push` 到中心伺服器並且叫其他人從那裡 `pull`。

這個通常會引起一些 `branch` 名稱簇變成像 `VCS` 裡一樣集中化，加上設置許可變成管理員的苦差事。

`Gitolite` 讓你定義一個 "個人的" 或者 "亂七八糟的" 命名空間字首給每個開發人員 (比如，`refs/personal/<devname>/``)；看在 `doc/3-faq-tips-etc.mkd` 裡的 "personal branches" 一段獲取細節。

"萬用字元" 倉庫

Gitolite 允許你定義帶萬用字元的倉庫 (其實還是 perl正則式), 比如隨便整個例子的話 `assignments/s[0-9][0-9]/a[0-9][0-9]`。這是一個非常有用的功能，需要通過設置 `$GL_WILDREPOS = 1;` 在 `rc`文件中啓用。允許你安排一個新許可模式 ("C") 允許用戶創建倉庫基於萬用字元，自動分配擁有權對特定用戶 - 創建者，允許他交出 R和 RW許可給其他合作用戶等等。這個功能在 `doc/4-wildcard-repositories.mkd` 文檔裡

其他功能

我們用一些其他功能的例子結束這段討論，這些以及其他功能都在 "faqs, tips, etc" 和其他文檔裡。

記錄: Gitolite 記錄所有成功的訪問。如果你太放鬆給了別人 `rewind`許可 (`RW+`) 和其他孩子弄沒了 "master"， 記錄檔會救你的命，如果其他簡單快速的找到SHA都不管用。

訪問權報告: 另一個方便的功能是你嘗試用 `ssh`連接到伺服器的時候發生了什麼。Gitolite告訴你哪個 `repos`你訪問過，那個訪問可能是什麼。這裡是例子：

```
hello scott, this is git@git running gitolite3 v3.01-18-g9609868 on git 1.7.4.4

R      anu-wsd
R      entrans
R  W   git-notes
R  W   gitolite
R  W   gitolite-admin
R      indic_web_input
R      shreelipi_converter
```

委託: 真正的大安裝，你可以把責任委託給一組倉庫給不同的人然後讓他們獨立管理那些部分。這個減少了主管理者的負擔，讓他瓶頸更小。這個功能在他自己的文檔目錄裡的 `doc/` 下面。

鏡像: Gitolite可以幫助你維護多個鏡像，如果主要伺服器掛掉的話在他們之間很容易切換。

Git 守護進程

對於提供公共的，非授權的唯讀訪問，我們可以拋棄 HTTP 協議，改用 Git 自己的協議，這主要是出於性能和速度的考慮。Git 協定遠比 HTTP 協定高效，因而存取速度也快，所以它能節省很多用戶的時間。

重申一下，這一點隻適用於非授權的唯讀訪問。如果建在防火牆之外的伺服器上，那麼它所提供的服務應該只是那些公開的唯讀項目。如果是在防火牆之內的伺服器上，可用于支撐大量參與人員或自動系統（用於持續集成或編譯的主機）唯讀訪問的專案，這樣可以省去逐一配置 SSH 公開金鑰的麻煩。

但不管哪種情形，Git 協定的配置設定都很簡單。基本上，只要以守護進程的形式運行該命令即可：

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

這裡的 `--reuseaddr` 選項表示在重啓服務前，不等之前的連接逾時就立即重啓。而 `--base-path` 選項則允許克隆專案時不必給出完整路徑。最後面的路徑告訴 Git 守護進程允許開放給使用者訪問的倉庫目錄。假如有防火牆，則需要為該主機的 9418 埠設置為允許通信。

以守護進程的形式運行該進程的方法有很多，但主要還得看用的是什麼作業系統。在 Ubuntu 主機上，可以用 Upstart 腳本達成。編輯該檔：

```
/etc/event.d/local-git-daemon
```

加入以下內容：

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

出於安全考慮，強烈建議用一個對倉庫只有讀取許可權的使用者身份來運行該進程 — 只需要簡單地新建一個名為 `git-ro` 的用戶（譯注：新建用戶默認對倉庫檔不具備寫許可權，但這取決於倉庫目錄的許可權設定。務必確認 `git-ro` 對倉庫只能讀不能寫。），並用它的身份來啓動進程。這裡為了簡化，後面我們還是用之前運行 Gitosis 的用戶 '`git`'。

這樣一來，當你重啓電腦時，Git 進程也會自動啓動。要是進程意外退出或者被殺掉，也會自行重啓。在設置完成後，不重啓電腦就啓動該守護進程，可以運行：

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

而在其他作業系統上，可以用 `xinetd`，或者 `sysvinit` 系統的腳本，或者其他類似的腳本 — 只要能讓那個命令變為守護進程並可監控。

接下來，我們必須告訴 Gitosis 哪些倉庫允許通過 Git 協議進行匿名唯讀訪問。如果每個倉庫都設有各自的段落，可以分別指定是否允許 Git 進程開放給使用者匿名讀取。比如允許通過 Git 協議訪問 `iphone_project`，可以把下面兩行加到 `gitosis.conf` 文件的末尾：

```
[repo iphone_project]
daemon = yes
```

在提交和推送完成後，運行中的 Git 守護進程就會回應來自 9418 埠對該專案的訪問請求。

如果不考慮 Gitosis，單單起了 Git 守護進程的話，就必須到每一個允許匿名唯讀訪問的倉庫目錄內，創建一個特殊名稱的空檔作為標誌：

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

該檔的存在，表明允許 Git 守護進程開放對該專案的匿名唯讀訪問。

Gitosis 還能設定哪些項目允許放在 GitWeb 上顯示。先打開 GitWeb 的設定檔 `/etc/gitweb.conf`，添加以下四行：

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

接下來，只要配置各個專案在 Gitosis 中的 `gitweb` 參數，便能達成是否允許 GitWeb 用戶流覽該專案。比如，要讓 `iphone_project` 專案在 GitWeb 裡出現，把 `repo` 的設定改成下面的樣子：

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

在提交並推送過之後，GitWeb 就會自動開始顯示 `iphone_project` 專案的細節和歷史。

Git 記管服務

如果不想經歷自己架設 Git 伺服器的麻煩，網路上有幾個專業的倉庫託管服務可供選擇。這樣做有幾大優點：託管帳戶的建立通常比較省時，方便專案的啓動，而且不涉及伺服器的維護和監控。即使內部創建並運行著自己的伺服器，同時為開源項目提供一個公共託管網站還是有好處的——讓開源社區更方便地找到該專案，並給予幫助。

目前，可供選擇的託管服務數量繁多，各有利弊。在 Git 官方 wiki 上的 Githosting 頁面有一個最新的託管服務清單：

<https://git.wiki.kernel.org/index.php/GitHosting>

由於本書無法全部一一介紹，而本人（譯注：指本書作者 Scott Chacon。）剛好在其中一家公司工作，所以接下來我們將會介紹如何在 GitHub 上建立新帳戶並啓動專案。至於其他託管服務大體也是這麼一個過程，基本的想法都是差不多的。

GitHub 是目前為止最大的開源 Git 記管服務，並且還是少數同時提供公共代碼和私有代碼託管服務的網站之一，所以你可以在上面同時保存開源和商業代碼。事實上，本書就是放在 GitHub 上合作編著的。（譯注：本書的翻譯也是放在 GitHub 上廣泛協作的。）

GitHub

GitHub 和大多數的代碼託管網站在處理專案命名空間的方式上略有不同。GitHub 的設計更側重於用戶，而不是完全基於專案。也就是說，如果我在 GitHub 上託管一個名為 grit 的項目的話，它的位址不會是 `github.com/grit`，而是按在用戶底下 `github.com/shacon/grit`（譯注：本書作者 Scott Chacon 在 GitHub 上的用户名是 shacon。）。不存在所謂某個項目的官方版本，所以假如第一作者放棄了某個項目，它可以無縫轉移到其它用戶的名下。

GitHub 同時也是一個向使用私有倉庫的用戶收取費用的商業公司，但任何人都可以方便快捷地申請到一個免費帳戶，並在上面託管數量不限的開源項目。接下來我們快速介紹一下 GitHub 的基本使用。

建立新帳戶

首先註冊一個免費帳戶。訪問 Pricing and Signup 頁面 <http://github.com/plans> 並點擊 Free account 裡的 Sign Up 按鈕（見圖 4-2），進入註冊頁面。



圖 4-2. GitHub 服務簡介頁面

選擇一個系統中尚未使用的用戶名，提供一個與之相關聯的電子郵件地址，並輸入密碼（見圖 4-3）：

The screenshot shows the GitHub sign-up form. It includes fields for "Username", "Email Address" (which is highlighted in yellow), "Password", and "Confirm Password". Below these is a field for "SSH Public Key" with a note: "Please enter one key only. You may add more later. This field is not required to sign up." At the bottom, there is a message: "You're signing up for the free plan. If you have any questions please [email support](#). By signing up, you agree to the [Terms of Service](#), [Privacy](#), and [Refund](#) policies." A "I agree, sign me up!" button is at the bottom.

圖 4-3. GitHub 用戶註冊表單

如果方便，現在就可以提供你的 SSH 公開金鑰。我們在前文的"小型安裝"一節介紹過生成新公開金鑰的方法。把新生成的公開金鑰複製粘貼到 SSH Public Key 文字方塊中即可。要是對生成公開金鑰的步驟不太清楚，也可以點擊 "explain ssh keys" 連結，會顯示各個主流作業系統上完成該步驟的介紹。點擊 "I agree, sign me up" 按鈕完成使用者註冊，並轉到該使用者的 dashboard 頁面（見圖 4-4）：

The screenshot shows the GitHub user profile page for 'testinguser'. At the top, there's a navigation bar with links for 'Browse', 'Guides', 'Advanced', 'Search', and a user icon. To the right of the user icon, there are links for 'account', 'profile', 'log out', 'dashboard', and 'gists'. Below the navigation, there's a 'News Feed' section with a link to 'create a new one'. On the right, there's a 'Your Repositories' section with links for 'all', 'public', 'private', 'sources', and 'forks'.

圖 4-4. GitHub 的用戶面板

接下來就可以建立新倉庫了。

建立新倉庫

點擊用戶面板上倉庫旁邊的 "create a new one" 連結，顯示 Create a New Repository 的表單（見圖 4-5）：

The screenshot shows the 'Create a New Repository' form. It includes fields for 'Project Name' (set to 'iphone_project'), 'Description' (set to 'iphone project for our mobile group'), and 'Homepage URL' (empty). There's also a section for 'Who has access to this repository?' with options for 'Anyone' (selected) and 'Upgrade your plan to create more private repositories!'. At the bottom is a 'Create Repository' button.

圖 4-5. 在 GitHub 上建立新倉庫

當然，項目名稱是必不可少的，此外也可以適當描述一下專案的情況或者給出官方網站的位址。然後點擊 "Create Repository" 按鈕，新倉庫就建立起來了（見圖 4-6）：

The screenshot shows the GitHub repository overview for 'testinguser / iphone_project'. It displays basic information: 'Description: iphone project for our mobile group', 'Homepage: Click to edit', 'Public Clone URL: git@github.com:testinguser/iphone_project.git', and 'Your Clone URL: git@github.com:testinguser/iphone_project.git'. There are also icons for issues (1), pull requests (1), and a yellow warning icon.

圖 4-6. GitHub 上各個專案的概要資訊

由於尚未提交代碼，點擊專案位址後 GitHub 會顯示一個簡要的指南，告訴你如何新建一個專案並推送上來，如何從現有項目推送，以及如何從一個公共的 Subversion 倉庫導入項目（見圖 4-7）：



圖 4-7. 新倉庫指南

該指南和本書前文介紹的類似，對於新的專案，需要先在本地初始化為 Git 專案，添加要管理的檔並作首次提交：

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

然後在這個本地倉庫內把 GitHub 添加為遠端倉庫，並推送 master 分支上來：

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

現在該項目就託管在 GitHub 上了。你可以把它的 URL 分享給每位對此項目感興趣的人。本例的 URL 是 http://github.com/testinguser/iphone_project。而在專案頁面的摘要部分，你會發現有兩個 Git URL 位址（見圖 4-8）：

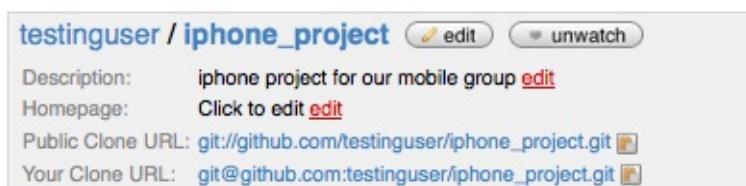


圖 4-8. 項目摘要中的公共 URL 和私有 URL

Public Clone URL 是一個公開的，唯讀的 Git URL，任何人都可以通過它克隆該專案。可以隨意散播這個 URL，比如發佈到個人網站之類的地方等等。

Your Clone URL 是一個基於 SSH 協議的可讀可寫 URL，只有使用與上傳的 SSH 公開金鑰對應的金鑰來連接時，才能通過它進行讀寫操作。其他使用者訪問該專案頁面時只能看到之前那個公共的 URL，看不到這個私有的 URL。

從 Subversion 導入項目

如果想把某個公共 Subversion 項目導入 Git，GitHub 可以幫忙。在指南的最後有一個指嚮導入 Subversion 頁面的連結。點擊它會看到一個表單，包含有關導入流程的資訊以及一個用來粘貼公共 Subversion 項目連接的文字方塊（見圖 4-9）：

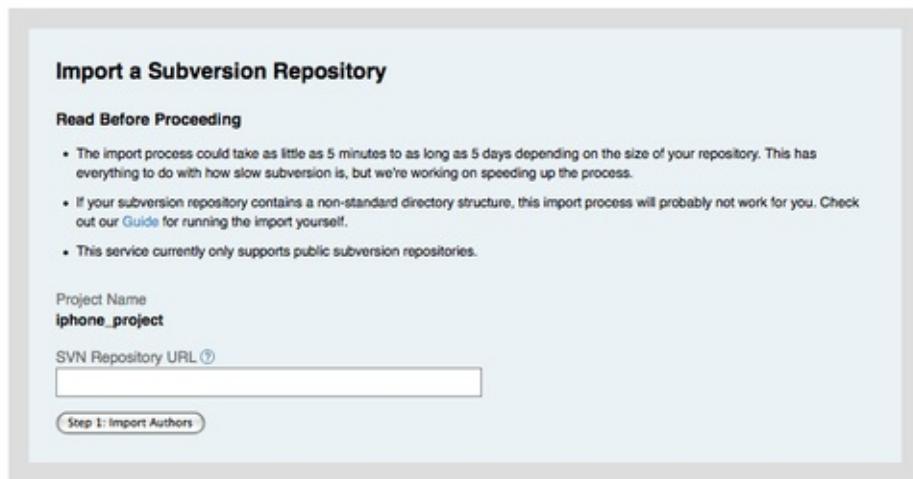


圖 4-9. Subversion 導入介面

如果專案很大，採用非標準結構，或者是私有的，那就無法借助該工具實現導入。到第 7 章，我們會介紹如何手工導入複雜工程的具體方法。

添加協作開發者

現在把團隊裡的其他人也加進來。如果 John，Josie 和 Jessica 都在 GitHub 註冊了帳戶，要賦予他們對該倉庫的推送許可權，可以把他們加為專案協作者。這樣他們就可以通過各自的公開金鑰訪問我的這個倉庫了。

點擊專案頁面上方的 "edit" 按鈕或者頂部的 Admin 標籤，進入該專案的管理頁面（見圖 4-10）：

The screenshot shows the GitHub repository settings for 'iphone_project'. It includes fields for Description, Homepage, Public Clone URL, Your Clone URL, GitHub Page, Pull Requests, RubyGem, and Donations. Below this, there are sections for Privacy (set to public) and Repository Collaborators, which currently has one entry: 'Add another collaborator'.

圖 4-10. GitHub 的專案管理頁面

為了給另一個用戶添加項目的寫許可權，點擊 "Add another collaborator" 連結，出現一個用於輸入用戶名的表單。在輸入的同時，它會自動跳出一個符合條件的候選名單。找到正確用戶名之後，點 Add 按鈕，把該使用者設為專案協作者（見圖 4-11）：

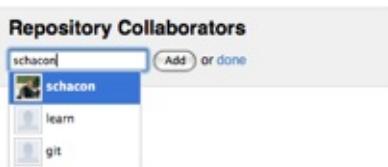


圖 4-11. 為專案添加協作者

添加完協作者之後，就可以在 Repository Collaborators 區域看到他們的名單（見圖 4-12）：

The screenshot shows the 'Repository Collaborators' list. It displays three users: 'schacon', 'duncanparkes', and 'spearce', each with a 'revoke' link next to their names. At the bottom, there is an empty input field and a 'Done' button.

圖 4-12. 專案協作者名單

如果要取消某人的訪問權，點擊 "revoke" 即可取消他的推送許可權。對於將來的專案，你可以從現有專案複製協作者名單，或者直接借用協作者群組。

專案頁面

在推送或從 Subversion 導入項目之後，你會看到一個類似圖 4-13 的項目主頁：

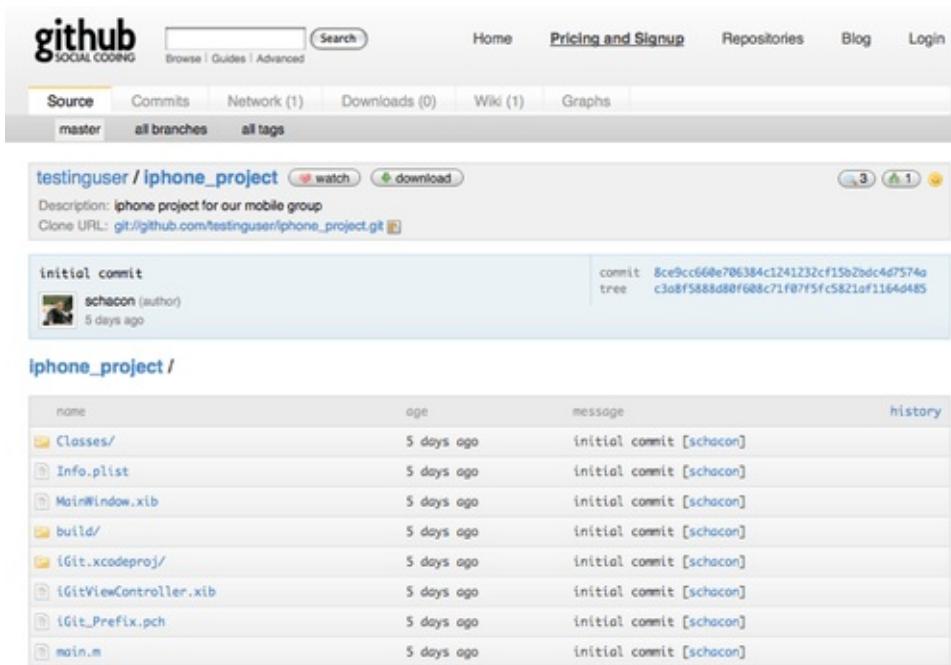


圖 4-13. GitHub 上的項目主頁

別人訪問你的專案時看到的就是這個頁面。它有若干導航標籤，Commits 標籤用於顯示提交歷史，最新的提交位於最上方，這和 `git log` 命令的輸出類似。Network 標籤展示所有派生了該專案並做出貢獻的用戶的關係圖譜。Downloads 標籤允許你上傳項目的二進位檔案，提供下載該項目各個版本的 tar/zip 包。Wiki 標籤提供了一個用於撰寫文檔或其他專案相關資訊的 wiki 網站。Graphs 標籤包含了一些視覺化的專案資訊與資料。預設打開的 Source 標籤頁面，則列出了該專案的目錄結構和概要資訊，並在下方自動展示 README 檔的內容（如果該檔存在的話），此外還會顯示最近一次提交的相關資訊。

派生項目

如果要為一個自己沒有推送許可權的項目貢獻代碼，GitHub 鼓勵使用派生（fork）。到那個感興趣的項目主頁上，點擊頁面上方的 "fork" 按鈕，GitHub 就會為你複製一份該項目的副本到你的倉庫中，這樣你就可以向自己的這個副本推送資料了。

採取這種辦法的好處是，專案擁有者不必忙於應付賦予他人推送許可權的工作。隨便誰都可以通過派生得到一個專案副本並在其中展開工作，事後只需要專案維護者將這些副本倉庫加為遠端倉庫，然後提取更新合併即可。

要派生一個專案，到原始專案的頁面（本例中是 `mojombo/chronic`）點擊 "fork" 按鈕（見圖 4-14）：



圖 4-14. 點擊 "fork" 按鈕獲得任意專案的可寫副本

幾秒鐘之後，你將進入新建的專案頁面，會顯示該專案派生自哪一個項目（見圖 4-15）：

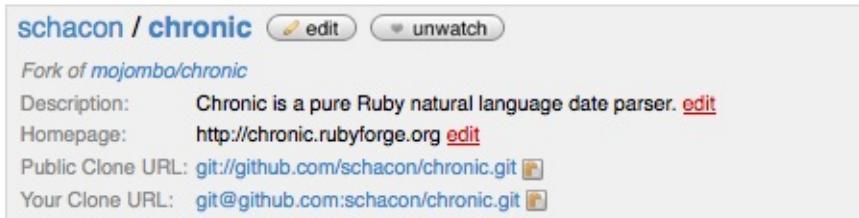


圖 4-15. 派生後得到的項目副本

GitHub 小結

關於 GitHub 就先介紹這麼多，能夠快速達成這些事情非常重要（譯注：門檻的降低和完成基本任務的簡單高效，對於推動開源專案的協作發展有著舉足輕重的意義。）。短短幾分鐘內，你就能創建一個新帳戶，添加一個專案並開始推送。如果專案是開源的，整個龐大的開發者社區都可以立即訪問它，提供各式各樣的幫助和貢獻。最起碼，這也是一種 Git 新手立即體驗嘗試 Git 的捷徑。

小結

我們討論並介紹了一些建立遠端 Git 倉庫的方法，接下來你可以通過這些倉庫同他人分享或合作。

運行自己的伺服器意味著更多的控制權以及在防火牆內部操作的可能性，當然這樣的伺服器通常需要投入一定的時間精力來架設維護。如果直接託管，雖然能免去這部分工作，但有時出於安全或版權的考慮，有些公司禁止將商業代碼託管到協力廠商服務商。

所以究竟採取哪種方案，並不是個難以取捨的問題，或者其一，或者相互配合，哪種合適就用哪種。

分散式 Git

為了便於專案中的所有開發者分享代碼，我們準備好了一台伺服器存放遠端 Git 倉庫。經過前面幾章的學習，我們已經學會了一些基本的本地工作流程中所需用到的命令。接下來，我們要學習下如何利用 Git 來組織和完成分散式工作流程。

特別是，當作為項目貢獻者時，我們該怎麼做才能方便維護者採納更新；或者作為專案維護者時，又該怎樣有效管理大量貢獻者的提交。

分散式工作流程

同傳統的集中式版本控制系統（CVCS）不同，開發者之間的協作方式因著 Git 的分散式特性而變得更為靈活多樣。在集中式系統上，每個開發者就像是連接在集線器上的節點，彼此的工作方式大體相像。而在 Git 網路中，每個開發者同時扮演著節點和集線器的角色，這就是說，每一個開發者都可以將自己的代碼貢獻到另外一個開發者的倉庫中，或者建立自己的公共倉庫，讓其他開發者基於自己的工作開始，為自己的倉庫貢獻代碼。於是，Git 的分散式協作便可以衍生出種種不同的工作流程，我會在接下來的章節介紹幾種常見的應用方式，並分別討論各自的優缺點。你可以選擇其中的一種，或者結合起來，應用到你自己的專案中。

集中式工作流

通常，集中式工作流程使用的都是單點協作模型。一個存放代碼倉庫的中心伺服器，可以接受所有開發者提交的代碼。所有的開發者都是普通的節點，作為中心集線器的消費者，平時的工作就是和中心倉庫同步資料（見圖 5-1）。

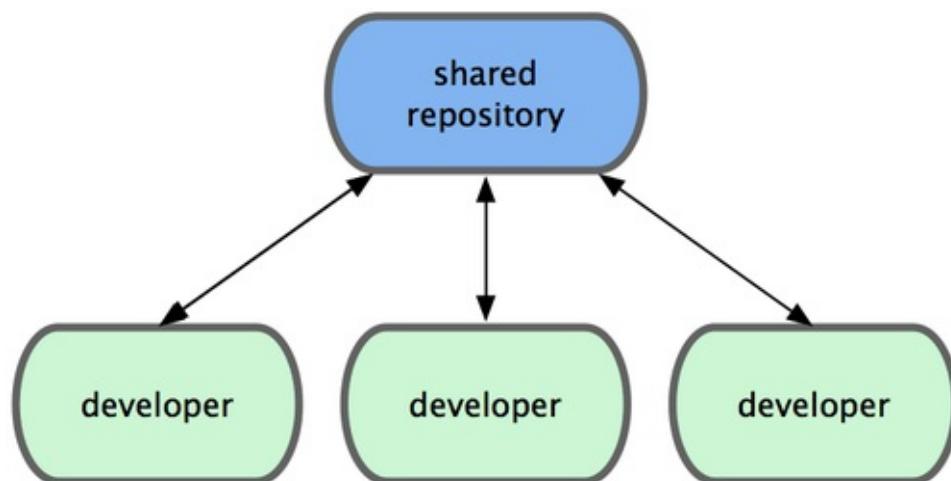


圖 5-1. 集中式工作流

如果兩個開發者從中心倉庫克隆代碼下來，同時作了一些修訂，那麼只有第一個開發者可以順利地把資料推送到共用伺服器。第二個開發者在提交他的修訂之前，必須先下載合併伺服器上的資料，解決衝突之後才能推送資料到共用伺服器上。在 Git 中這麼用也決無問題，這就好比是在用 Subversion（或其他 CVCS）一樣，可以很好地工作。

如果你的團隊不是很大，或者大家都已經習慣了使用集中式工作流程，完全可以採用這種簡單的模式。只需要配置好一台中心伺服器，並給每個人推送資料的許可權，就可以開展工作了。但如果提交代碼時有衝突，Git 根本就不會讓用戶覆蓋他人代碼，它直接駁回第二個人的

提交操作。這就等於告訴提交者，你所作的修訂無法通過快近（fast-forward）來合併，你必須先拉取最新資料下來，手工解決衝突合併後，才能繼續推送新的提交。絕大多數人都熟悉和瞭解這種模式的工作方式，所以使用也非常廣泛。

集成管理員工作流

由於 Git 允許使用多個遠端倉庫，開發者便可以建立自己的公共倉庫，往裡面寫資料並共用給他人，而同時又可以從別人的倉庫中提取他們的更新過來。這種情形通常都會有個代表著官方發佈的專案倉庫（blessed repository），開發者們由此倉庫克隆出一個自己的公共倉庫（developer public），然後將自己的提交推送上來，請求官方倉庫的維護者拉取更新合併到主項目。維護者在自己的本地也有個克隆倉庫（integration manager），他可以將你的公共倉庫作為遠端倉庫添加進來，經過測試無誤後合併到主幹分支，然後再推送到官方倉庫。工作流程看起來就像圖 5-2 所示：

1. 專案維護者可以推送資料到公共倉庫 blessed repository。
2. 貢獻者克隆此倉庫，修訂或編寫新代碼。
3. 貢獻者推送資料到自己的公共倉庫 developer public。
4. 貢獻者給維護者發送郵件，請求拉取自己的最新修訂。
5. 維護者在自己本地的 integration manger 倉庫中，將貢獻者的倉庫加為遠程倉庫，合併更新並做測試。
6. 維護者將合併後的更新推送到主倉庫 blessed repository。

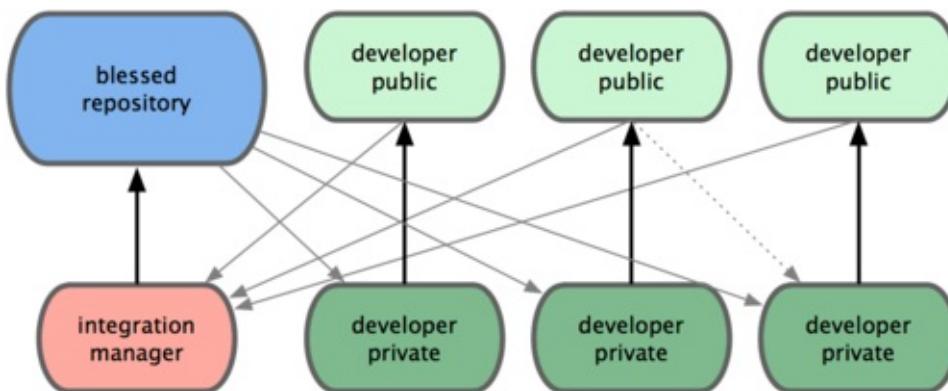


圖 5-2. 集成管理員工作流

在 GitHub 網站上使用得最多的就是這種工作流。人們可以複製（fork 亦即克隆）某個項目到自己的列表中，成為自己的公共倉庫。隨後將自己的更新提交到這個倉庫，所有人都可以看到你的每次更新。這麼做最主要的優點在於，你可以按照自己的節奏繼續工作，而不必等待維護者處理你提交的更新；而維護者也可以按照自己的節奏，任何時候都可以過來處理接納你的貢獻。

司令官與副官工作流

這其實是上一種工作流的變體。一般超大型的專案才會用到這樣的工作方式，像是擁有數百協作開發者的 Linux 內核專案就是如此。各個集成管理員分別負責集成專案中的特定部分，所以稱為副官（lieutenant）。而所有這些集成管理員頭上還有一位負責統籌的總集成管理員，稱為司令官（dictator）。司令官維護的倉庫用於提供所有協作者拉取最新集成的項目代碼。整個流程看起來如圖 5-3 所示：

1. 一般的開發者在自己的特性分支上工作，並不定期地根據主幹分支（dictator 上的 master）衍合。
2. 副官（lieutenant）將普通開發者的特性分支合併到自己的 master 分支中。
3. 司令官（dictator）將所有副官的 master 分支併入自己的 master 分支。
4. 司令官（dictator）將集成後的 master 分支推送到共用倉庫 blessed repository 中，以便所有其他開發者以此為基礎進行衍合。

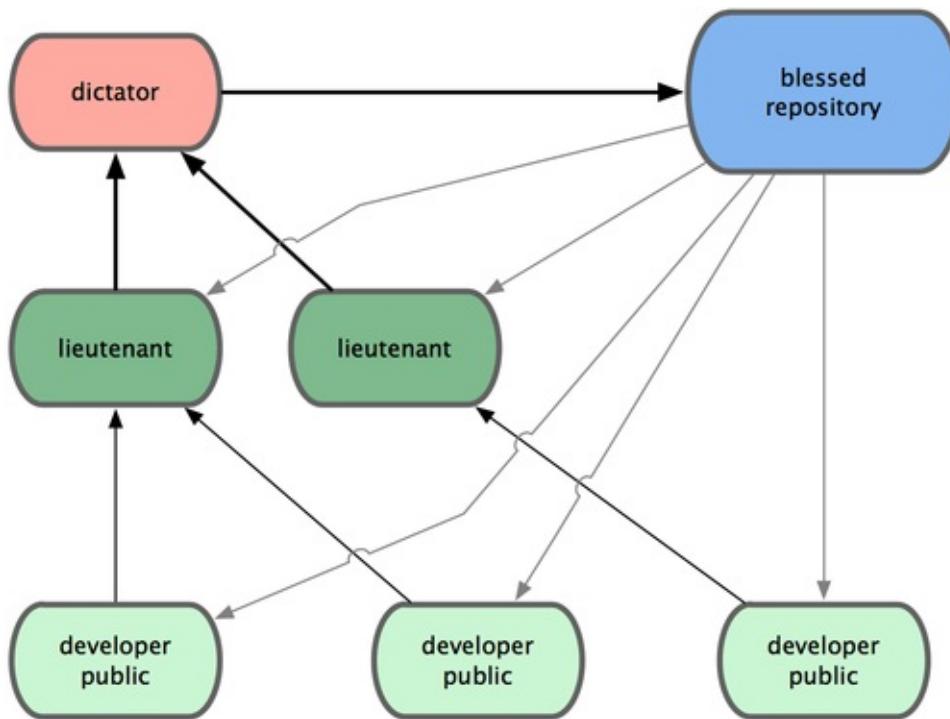


圖 5-3. 司令官與副官工作流

這種工作流程並不常用，只有當專案極為龐雜，或者需要多級別管理時，才會體現出優勢。利用這種方式，專案總負責人（即司令官）可以把大量分散的集成工作委託給不同的小組負責人分別處理，最後再統籌起來，如此各人的職責清晰明確，也不易出錯（譯注：此乃分而治之）。

以上介紹的是常見的分散式系統可以應用的工作流程，當然不止於 Git。在實際的開發工作中，你可能會遇到各種為了滿足特定需求而有所變化的工作方式。我想現在你應該已經清楚，接下來自己需要用哪種方式開展工作了。下節我還會再舉些例子，看看各式工作流中的每個角色具體應該如何操作。

為專案作貢獻

接下來，我們來學習一下作為專案貢獻者，會有哪些常見的工作模式。

不過要說清楚整個協作過程真的很難，Git 如此靈活，人們的協作方式便可以各式各樣，沒有固定不變的範式可循，而每個專案的具體情況又多少會有些不同，比如說參與者的規模，所選擇的工作流程，每個人的提交許可權，以及 Git 以外貢獻等等，都會影響到具體操作的細節。

首當其衝的是參與者規模。專案中有多少開發者是經常提交代碼的？經常又是多久呢？大多數兩至三人的小團隊，一天大約只有幾次提交，如果不是什麼熱門項目的話就更少了。可要是在大公司裡，或者大項目中，參與者可以多到上千，每天都會有十幾個上百個補丁提交上來。這種差異帶來的影響是顯著的，越是多的人參與進來，就越難保證每次合併正確無誤。你正在工作的代碼，可能會因為合併進來其他人的更新而變得過時，甚至受創無法運行。而已經提交上去的更新，也可能在等著審核合併的過程中變得過時。那麼，我們該怎樣做才能確保代碼是最新的，提交的補丁也是可用的呢？

接下來便是專案所採用的工作流。是集中式的，每個開發者都具有等同的寫許可權？專案是否有專人負責檢查所有補丁？是不是所有補丁都做過同行複閱（peer-review）再通過審核的？你是否參與審核過程？如果使用副官系統，那你是不是限定于只能向此副官提交？

還有你的提交許可權。有或沒有向主專案提交更新的許可權，結果完全不同，直接決定最終採用怎樣的工作流。如果不能直接提交更新，那該如何貢獻自己的代碼呢？是不是該有個什麼策略？你每次貢獻代碼會有多少量？提交頻率呢？

所有以上這些問題都會或多或少影響到最終採用的工作流。接下來，我會在一系列由簡入繁的具體用例中，逐一闡述。此後在實踐時，應該可以借鑒這裡的例子，略作調整，以滿足實際需要構建自己的工作流。

提交指南

開始分析特定用例之前，先來瞭解下如何撰寫提交說明。一份好的提交指南可以說明協作者更輕鬆更有效地配合。Git 項目本身就提供了一份文檔（Git 專案原始程式碼目錄中 [Documentation/SubmittingPatches](#)），列數了大量提示，從如何編撰提交說明到提交補丁，不一而足。

首先，請不要在更新中提交多餘的白字元（whitespace）。Git 有種檢查此類問題的方法，在提交之前，先運行 `git diff --check`，會把可能的多餘白字元修正列出來。下面的示例，我已經把終端中顯示為紅色的白字元用 `x` 替換掉：

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+    def command(git_cmd)XXXX
```

這樣在提交之前你就可以看到這類問題，及時解決以免困擾其他開發者。

接下來，請將每次提交限定於完成一次邏輯功能。並且可能的話，適當地分解為多次小更新，以便每次小型提交都更易於理解。請不要在週末窮追猛打一次性解決五個問題，而最後拖到週一再提交。就算是這樣也請盡可能利用暫存區域，將之前的改動分解為每次修復一個問題，再分別提交和加註說明。如果針對兩個問題改動的是同一個檔，可以試試看 `git add -patch` 的方式將部分內容置入暫存區域（我們會在第六章再詳細介紹）。無論是五次小提交還是混雜在一起的大提交，最終分支末端的專案快照應該還是一樣的，但分解開來之後，更便於其他開發者複閱。這麼做也方便自己將來取消某個特定問題的修復。我們將在第六章介紹一些重寫提交歷史，同暫存區域交互的技巧和工具，以便最終得到一個乾淨有意義，且易於理解的提交歷史。

最後需要謹記的是提交說明的撰寫。寫得好可以讓大家協作起來更輕鬆。一般來說，提交說明最好限制在一行以內，50 個字元以下，簡明扼要地描述更新內容，空開一行後，再展開詳細注解。Git 專案本身需要開發者撰寫詳盡注解，包括本次修訂的因由，以及前後不同實現之間的比較，我們也該借鑒這種做法。另外，提交說明應該用祈使現在式語態，比如，不要說成“*I added tests for*”或“*Adding tests for*”而應該用“*Add tests for*”。下麵是來自 tpope.net 的 Tim Pope 原創的提交說明格式模版，供參考：

本次更新的簡要描述（50 個字元以內）

如果必要，此處展開詳盡闡述。段落寬度限定在 72 個字元以內。

某些情況下，第一行的簡要描述將用作郵件標題，其餘部分作為郵件正文。

其間的空行是必要的，以區分兩者（當然沒有正文另當別論）。

如果並在一起，`rebase` 這樣的工具就可能會迷惑。

另起空行後，再進一步補充其他說明。

- 可以使用這樣的條目列舉式。
- 一般以單個空格緊跟短劃線或者星號作為每項條目的起始符。每個條目間用一空行隔開。
不過這裡按自己項目的約定，可以略作變化。

如果你的提交說明都用這樣的格式來書寫，好多事情就可以變得十分簡單。Git 專案本身就是這樣要求的，我強烈建議你到 Git 專案倉庫下運行 `git log --no-merges` 看看，所有提交歷史的說明是怎樣撰寫的。（譯注：如果現在還沒有克隆 git 項目原始程式碼，是時候 `git clone git://git.kernel.org/pub/scm/git/git.git` 了。）

爲簡單起見，在接下來的例子（及本書隨後的所有演示）中，我都不會用這種格式，而使用 `-m` 選項提交 `git commit`。不過請還是按照我之前講的做，別學我這裡偷懶的方式。

私有的小型團隊

我們從最簡單的情況開始，一個私有專案，與你一起協作的還有另外一到兩位開發者。這裡說私有，是指原始程式碼不公開，其他人無法訪問項目倉庫。而你和其他開發者則都具有推送資料到倉庫的許可權。

這種情況下，你們可以用 Subversion 或其他集中式版本控制系統類似的工作流來協作。你仍然可以得到 Git 帶來的其他好處：離線提交，快速分支與合併等等，但工作流程還是差不多的。主要區別在於，合併操作發生在用戶端而非伺服器上。讓我們來看看，兩個開發者一起使用同一個共用倉庫，會發生些什麼。第一個人，John，克隆了倉庫，作了些更新，在本地提交。（下面的例子中省略了常規提示，用 `...` 代替以節約版面。）

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ 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(-)
```

第二個開發者，Jessica，一樣這麼做：克隆倉庫，提交更新：

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

現在，Jessica 將她的工作推送到伺服器上：

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

John 也嘗試推送自己的工作上去：

```
# 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 的推送操作被駁回，因為 Jessica 已經推送了新的資料上去。請注意，特別是你用慣了 Subversion 的話，這裡其實修改的是兩個檔，而不是同一個檔的同一個地方。Subversion 會在伺服器端自動合併提交上來的更新，而 Git 則必須先在本地合併後才能推送。於是，John 不得不先把 Jessica 的更新拉下來：

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

此刻，John 的本地倉庫如圖 5-4 所示：

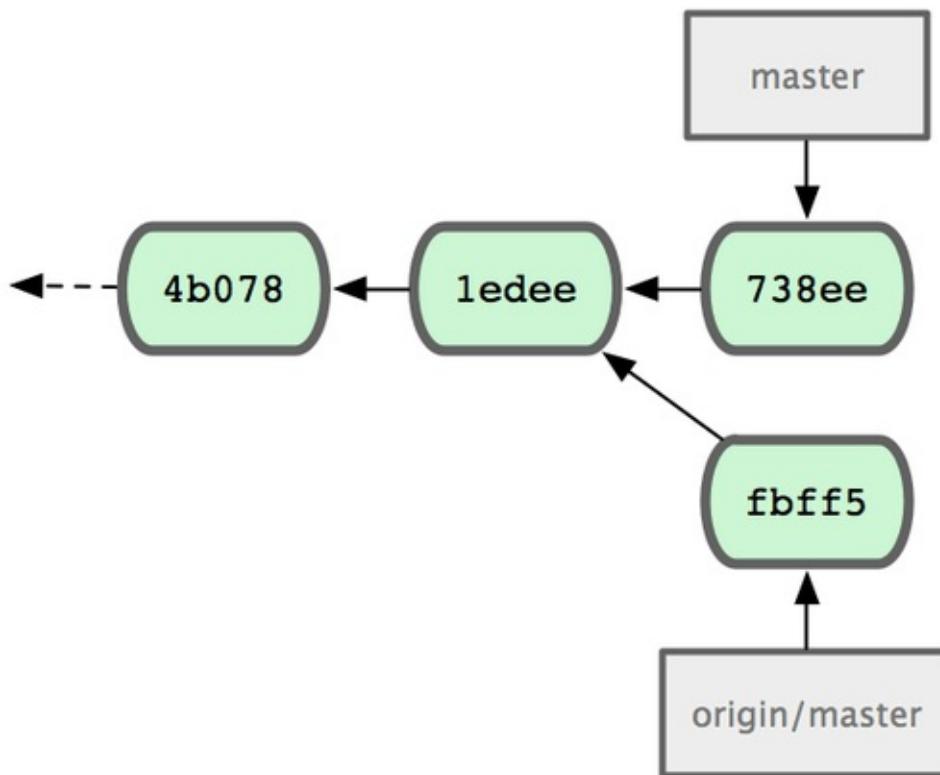


圖 5-4. John 的倉庫歷史

雖然 John 下載了 Jessica 推送到伺服器的最近更新 (fbff5)，但目前只是 `origin/master` 指標指向它，而當前的本地分支 `master` 仍然指向自己的更新 (738ee)，所以需要先把她的提交合併過來，才能繼續推送資料：

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

還好，合併過程非常順利，沒有衝突，現在 John 的提交歷史如圖 5-5 所示：

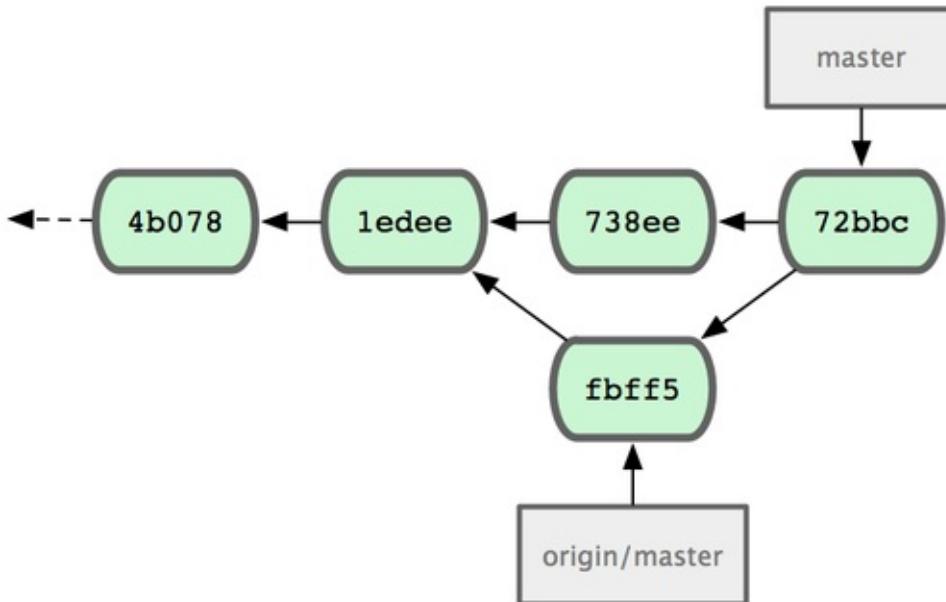


圖 5-5. 合併 origin/master 後 John 的倉庫歷史

現在，John 應該再測試一下代碼是否仍然正常工作，然後將合併結果（72bbc）推送到伺服器上：

```
$ git push origin master
...
To john@githost:simplegit.git
  fbf5bc..72bbc59  master -> master
```

最終，John 的提交歷史變為圖 5-6 所示：

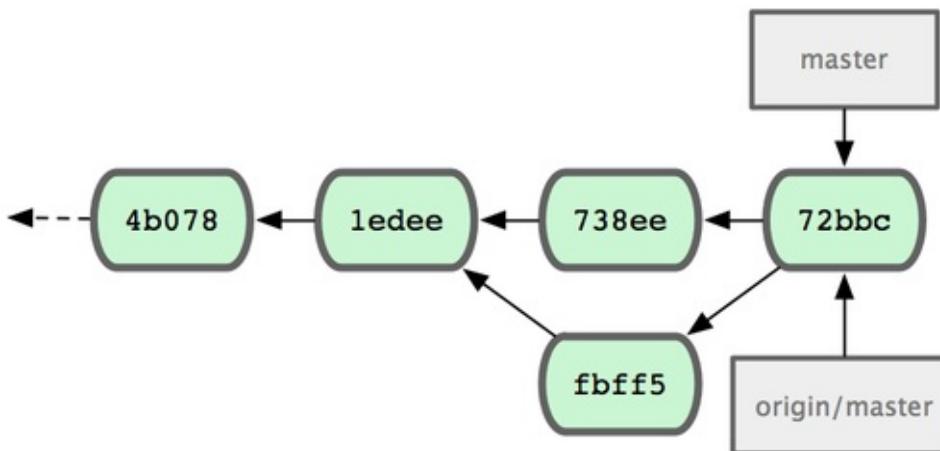


圖 5-6. 推送後 John 的倉庫歷史

而在這段時間，Jessica 已經開始在另一個特性分支工作了。她創建了 issue54 並提交了三次更新。她還沒有下載 John 提交的合併結果，所以提交歷史如圖 5-7 所示：

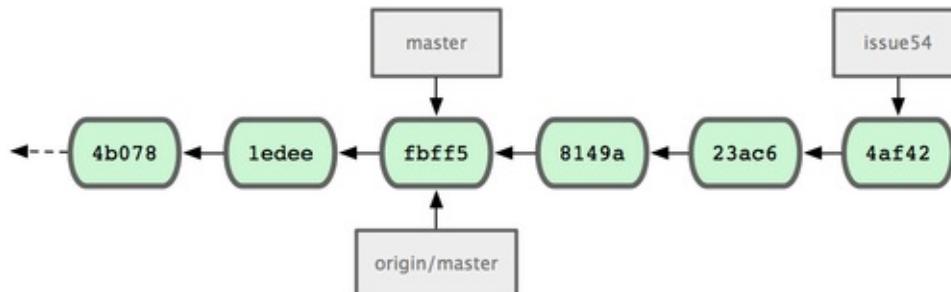


圖 5-7. Jessica 的提交歷史

Jessica 想要先和伺服器上的資料同步，所以先下載資料：

```

# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
  fbf5bc..72bbc59  master      -> origin/master
  
```

於是 Jessica 的本地倉庫歷史多出了 John 的兩次提交（738ee 和 72bbc），如圖 5-8 所示：

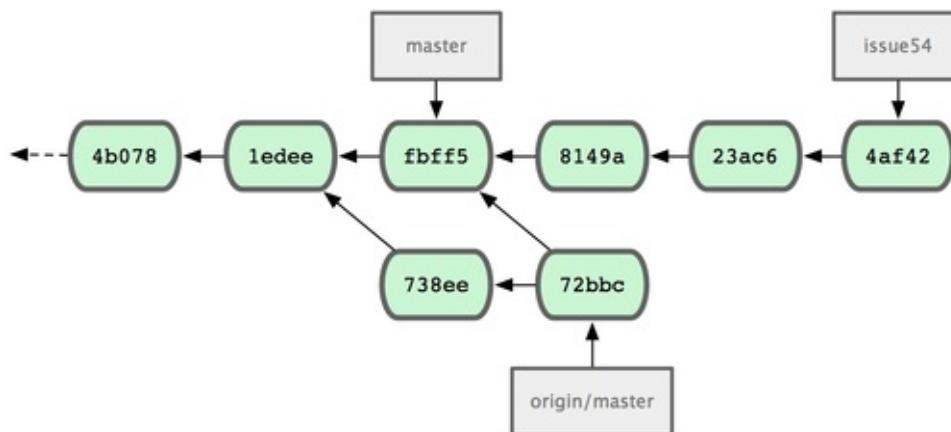


圖 5-8. 獲取 John 的更新之後 Jessica 的提交歷史

此時，Jessica 在特性分支上的工作已經完成，但她想在推送資料之前，先確認下要並進來的資料究竟是什麼，於是運行 `git log` 查看：

```

$ git log --no-merges origin/master ^issue54
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

        removed invalid default value
  
```

現在，Jessica 可以將特性分支上的工作並到 `master` 分支，然後再併入 John 的工作（`origin/master`）到自己的 `master` 分支，最後再推送回伺服器。當然，得先切回主分支才能集成所有資料：

```
$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

要合併 `origin/master` 或 `issue54` 分支，誰先誰後都沒有關係，因為它們都在上游（*upstream*）（譯注：想像分叉的更新像是匯流成河的源頭，所以上游 *upstream* 是指最新的提交），所以無所謂先後順序，最終合併後的內容快照都是一樣的，而僅是提交歷史看起來會有些先後差別。Jessica 選擇先合併 `issue54`：

```
$ git merge issue54
Updating fffffbc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

正如所見，沒有衝突發生，僅是一次簡單快進。現在 Jessica 開始合併 John 的工作（`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(-)
```

所有的合併都非常乾淨。現在 Jessica 的提交歷史如圖 5-9 所示：

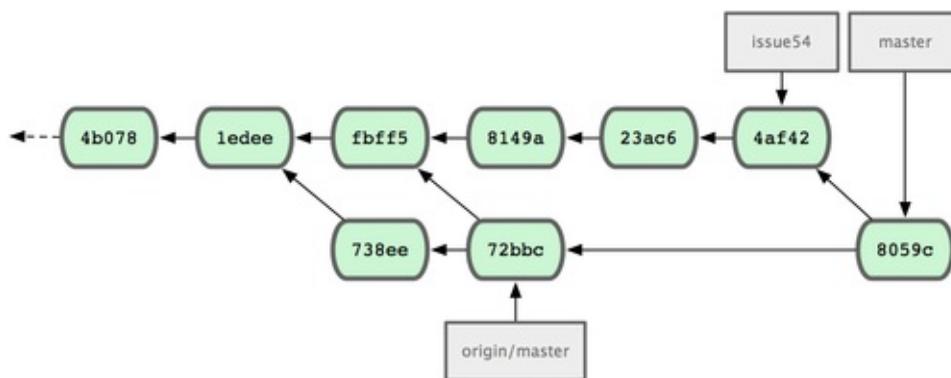


圖 5-9. 合併 John 的更新後 Jessica 的提交歷史

現在 Jessica 已經可以在自己的 `master` 分支中訪問 `origin/master` 的最新改動了，所以她應該可以成功推送到伺服器上（假設 John 此時沒再推送新資料上來）：

```
$ git push origin master
...
To jessica@githost:simplegit.git
 72bbc59..8059c15  master -> master
```

至此，每個開發者都提交了若干次，且成功合併了對方的工作成果，最新的提交歷史如圖 5-10 所示：

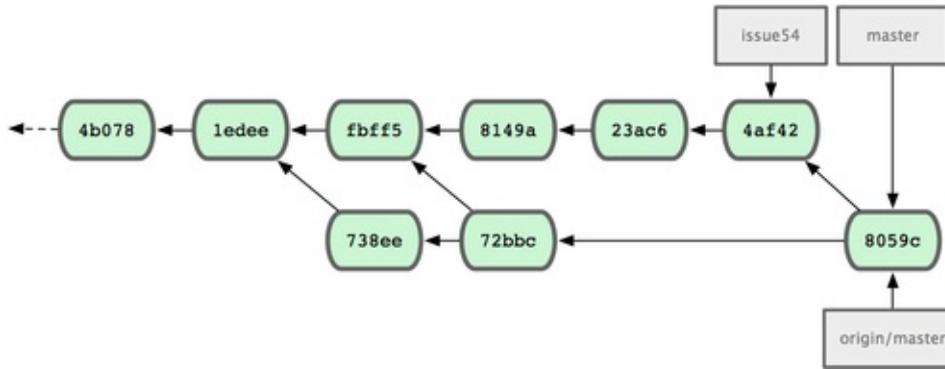


圖 5-10. Jessica 推送資料後的提交歷史

以上就是最簡單的協作方式之一：先在自己的特性分支中工作一段時間，完成後合併到自己的 `master` 分支；然後下載合併 `origin/master` 上的更新（如果有的話），再推回遠端伺服器。一般的協作流程如圖 5-11 所示：



圖 5-11. 多使用者共用倉庫協作方式的一般工作流程時序

私有團隊間協作

現在我們來看更大一點規模的私有團隊協作。如果有幾個小組分頭負責若干特性的開發和集成，那他們之間的協作過程是怎樣的。

假設 John 和 Jessica 一起負責開發某項特性 A，而同時 Jessica 和 Josie 一起負責開發另一項功能 B。公司使用典型的集成管理員式工作流，每個組都有一名管理員負責集成本組代碼，及更新專案主倉庫的 `master` 分支。所有開發都在代表小組的分支上進行。

讓我們跟隨 Jessica 的視角看看她的工作流程。她參與開發兩項特性，同時和不同小組的開發者一起協作。克隆生成本地倉庫後，她打算先著手開發特性 A。於是創建了新的 `featureA` 分支，繼而編寫代碼：

```
# 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(-)
```

此刻，她需要分享目前的進展給 John，於是她將自己的 `featureA` 分支提交到伺服器。由於 Jessica 沒有許可權推送資料到主倉庫的 `master` 分支（只有集成管理員有此許可權），所以只能將此分支推上去同 John 共用協作：

```
$ git push origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica 發郵件給 John 讓他上來看看 `featureA` 分支上的進展。在等待他的回饋之前，Jessica 決定繼續工作，和 Josie 一起開發 `featureB` 上的特性 B。當然，先創建此分支，分叉點以伺服器上的 `master` 為起點：

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch "featureB"
```

隨後，Jessica 在 `featureB` 上提交了若干更新：

```
$ 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(-)
```

現在 Jessica 的更新歷史如圖 5-12 所示：

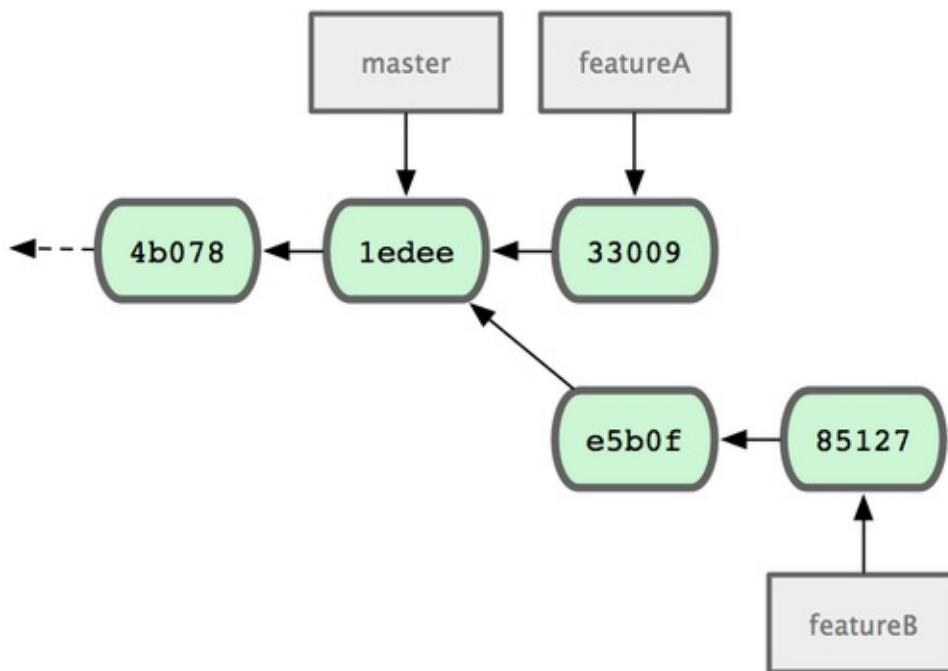


圖 5-12. Jessica 的更新歷史

Jessica 正準備推送自己的進展上去，卻收到 Josie 的來信，說是她已經將自己的工作推到伺服器上的 `featureBee` 分支了。這樣，Jessica 就必須先將 Josie 的代碼合併到自己本地分支中，才能再一起推送回伺服器。她用 `git fetch` 下載 Josie 的最新代碼：

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

然後 Jessica 使用 `git merge` 將此分支合併到自己分支中：

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
lib/simplegit.rb |    4 +++
1 files changed, 4 insertions(+), 0 deletions(-)
```

合併很順利，但另外有個小問題：她要推送自己的 `featureB` 分支到伺服器上的 `featureBee` 分支上去。當然，她可以使用冒號 (:) 格式指定目標分支：

```
$ git push origin featureB:featureBee
...
To jessica@githost:simplegit.git
 fba9af8..cd685d1  featureB -> featureBee
```

我們稱此為 `refsing`。更多有關於 Git `refsing` 的討論和使用方式會在第九章作詳細闡述。

接下來，John 發郵件給 Jessica 告訴她，他看了之後作了些修改，已經推回伺服器 `featureA` 分支，請她過目下。於是 Jessica 運行 `git fetch` 下載最新資料：

```
$ git fetch origin
...
From jessica@githost:simplegit
 3300904..aad881d  featureA  -> origin/featureA
```

接下來便可以用 `git log` 查看更新了些什麼：

```
$ git log origin/featureA ^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
```

最後，她將 John 的工作合併到自己的 `featureA` 分支中：

```
$ 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 稍做一番修整後同步到伺服器：

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@githost:simplegit.git
 3300904..774b3ed  featureA -> featureA
```

現在的 Jessica 提交歷史如圖 5-13 所示：

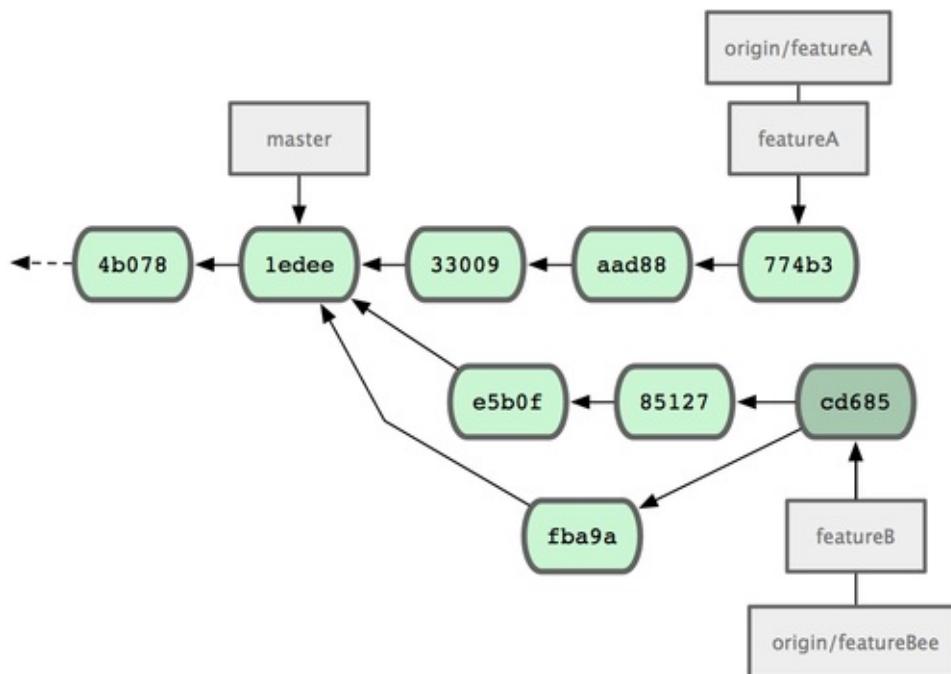


圖 5-13. 在特性分支中提交更新後的提交歷史

現在，Jessica，Josie 和 John 通知集成管理員伺服器上的 `featureA` 及 `featureBee` 分支已經準備好，可以併入主線了。在管理員完成集成工作後，主分支上便多出一個新的合併提交（`5399e`），用 `fetch` 命令更新到本地後，提交歷史如圖 5-14 所示：

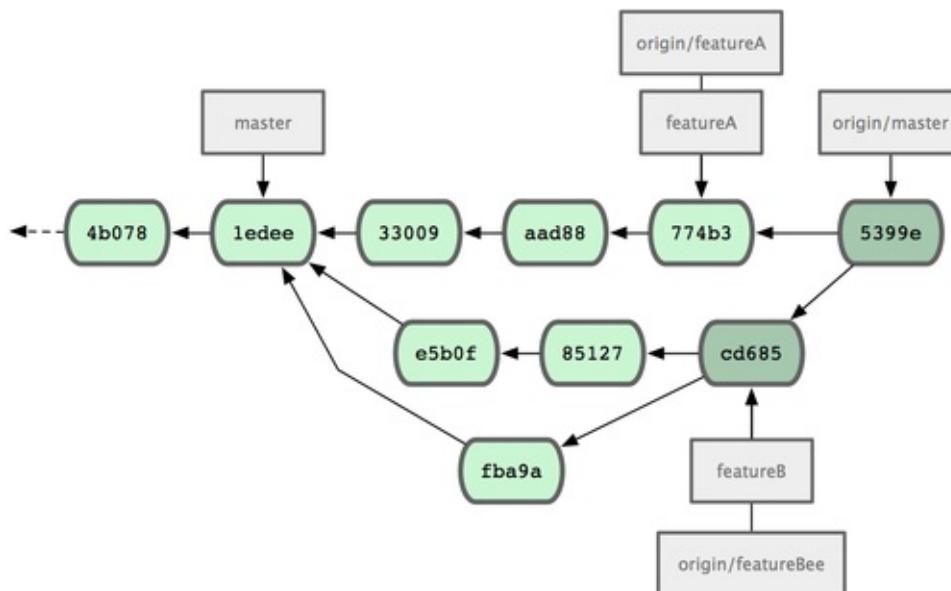


圖 5-14. 合併特性分支後的 Jessica 提交歷史

許多開發小組改用 Git 就是因為它允許多個小組間並行工作，而在稍後恰當時機再行合併。通過共用遠端分支的方式，無需干擾整體專案代碼便可以開展工作，因此使用 Git 的小型團隊間協作可以變得非常靈活自由。以上工作流程的時序如圖 5-15 所示：

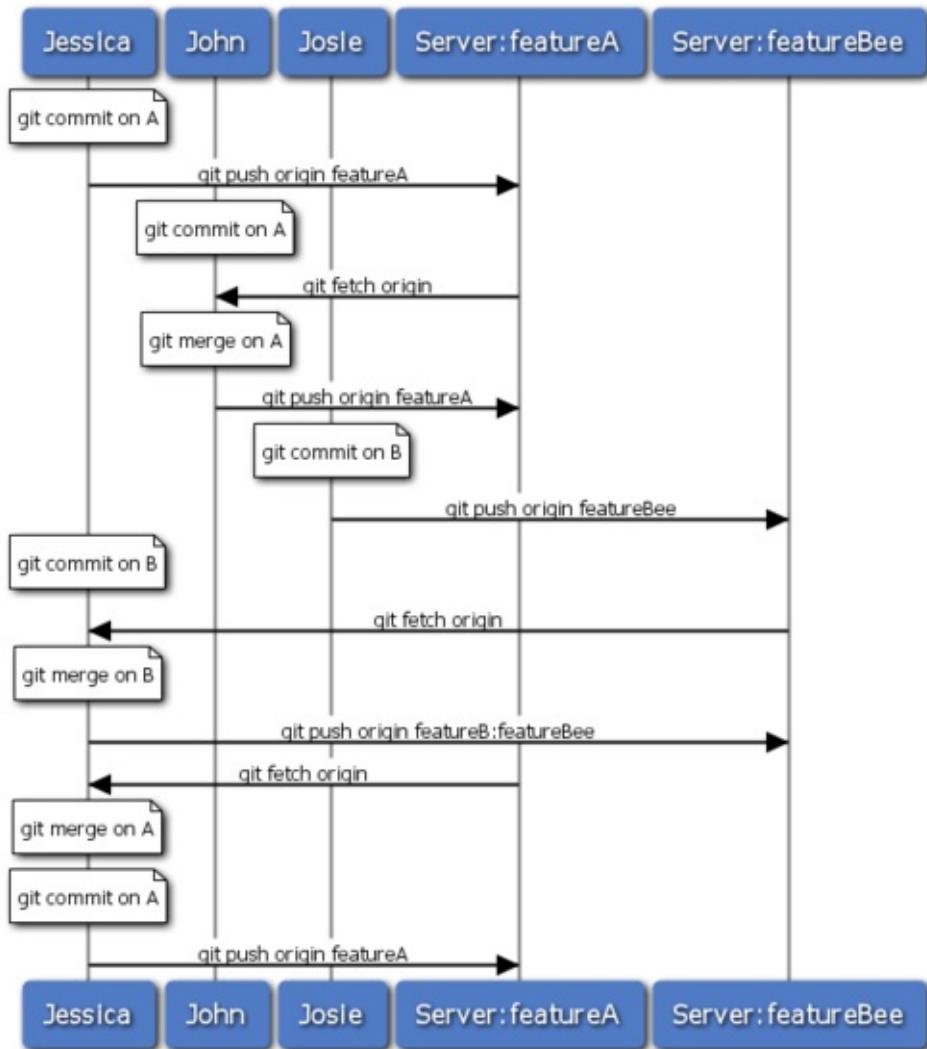


圖 5-15. 團隊間協作工作流程基本時序

公開的小型專案

上面說的是私有專案協作，但要給公開專案作貢獻，情況就有些不同了。因為你沒有直接更新主倉庫分支的許可權，得尋求其它方式把工作成果交給專案維護人。下面會介紹兩種方法，第一種使用 git 託管服務商提供的倉庫複製功能，一般稱作 fork，比如 repo.or.cz 和 GitHub 都支援這樣的動作，而且許多項目管理員都希望大家使用這樣的方式。另一種方法是通過電子郵件寄送檔補丁。

但不管哪種方式，起先我們總需要克隆原始倉庫，而後創建特性分支開展工作。基本工作流程如下：

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
$ (work)
$ git commit
$ (work)
$ git commit
```

你可能想到用 `rebase -i` 將所有更新先變作單個提交，又或者想重新安排提交之間的差異補丁，以方便專案維護者審閱 -- 有關互動式衍合操作的細節見第六章。

在完成了特性分支開發，提交給專案維護者之前，先到原始專案的頁面上點擊“Fork”按鈕，創建一個自己可寫的公共倉庫（譯注：即下面的 `url` 部分，參照後續的例子，應該是 `git://githost/simplegit.git`）。然後將此倉庫添加為本地的第二個遠端倉庫，姑且稱為 `myfork`：

```
$ git remote add myfork (url)
```

你需要將本地更新推送到這個倉庫。要是將遠端 `master` 合併到本地再推回去，還不如把整個特性分支推上去來得乾脆直接。而且，假若項目維護者未採納你的貢獻的話（不管是直接合併還是 `cherry pick`），都不用回退（`rewind`）自己的 `master` 分支。但若維護者合併或 `cherry-pick` 了你的工作，最後總還可以從他們的更新中同步這些代碼。好吧，現在先把 `featureA` 分支整個推上去：

```
$ git push myfork featureA
```

然後通知專案管理員，讓他來抓取你的代碼。通常我們把這件事叫做 `pull request`。可以直接用 GitHub 等網站提供的“`pull request`”按鈕自動發送請求通知；或手工把 `git request-pull` 命令輸出結果電郵給專案管理員。

`request-pull` 命令接受兩個參數，第一個是本地特性分支開始前的原始分支，第二個是請求對方來抓取的 Git 倉庫 URL（譯注：即下面 `myfork` 所指的，自己可寫的公共倉庫）。比如現在 Jessica 準備要給 John 發一個 `pull request`，她之前在自己的特性分支上提交了兩次更新，並把分支整個推到了伺服器上，所以運行該命令會看到：

```
$ 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(-)
```

輸出的內容可以直接發郵件給管理者，他們就會明白這是從哪次提交開始旁支出去的，該到哪裡去抓取新的代碼，以及新的代碼增加了哪些功能等等。

像這樣隨時保持自己的 `master` 分支和官方 `origin/master` 同步，並將自己的工作限制在特性分支上的做法，既方便又靈活，採納和丟棄都輕而易舉。就算原始主幹發生變化，我們也能重新衍合提供新的補丁。比如現在要開始第二項特性的開發，不要在原來已推送的特性分支上繼續，還是按原始 `master` 開始：

```
$ git checkout -b featureB origin/master
$ (work)
$ git commit
$ git push myfork featureB
$ (email maintainer)
$ git fetch origin
```

現在，A、B 兩個特性分支各不相擾，如同竹筒裡的兩顆豆子，佇列中的兩個補丁，你隨時都可以分別從頭寫過，或者衍合，或者修改，而不用擔心特性代碼的交叉混雜。如圖 5-16 所示：

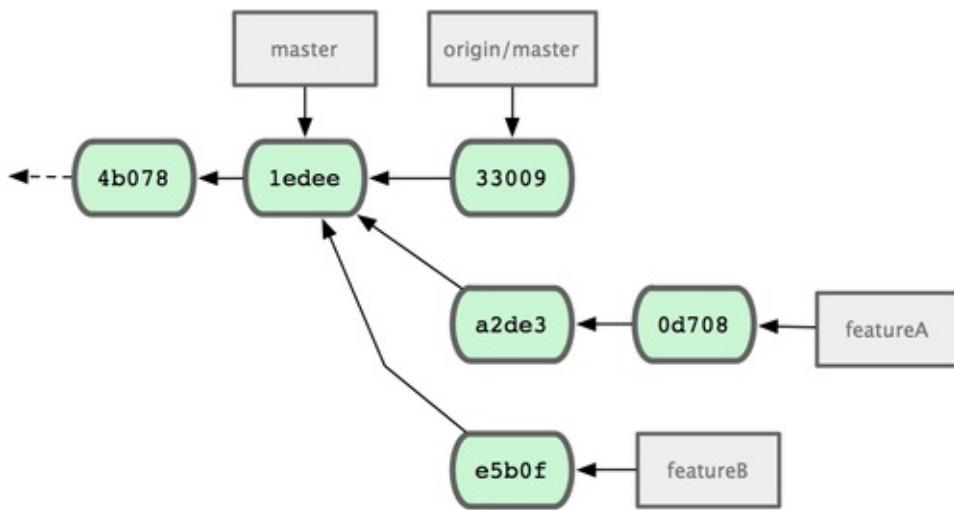


圖 5-16. featureB 以後的提交歷史

假設項目管理員接納了許多別人提交的補丁後，準備要採納你提交的第一個分支，卻發現因為代碼基準不一致，合併工作無法正確乾淨地完成。這就需要你再次衍合到最新的 `origin/master`，解決相關衝突，然後重新提交你的修改：

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

自然，這會重寫提交歷史，如圖 5-17 所示：

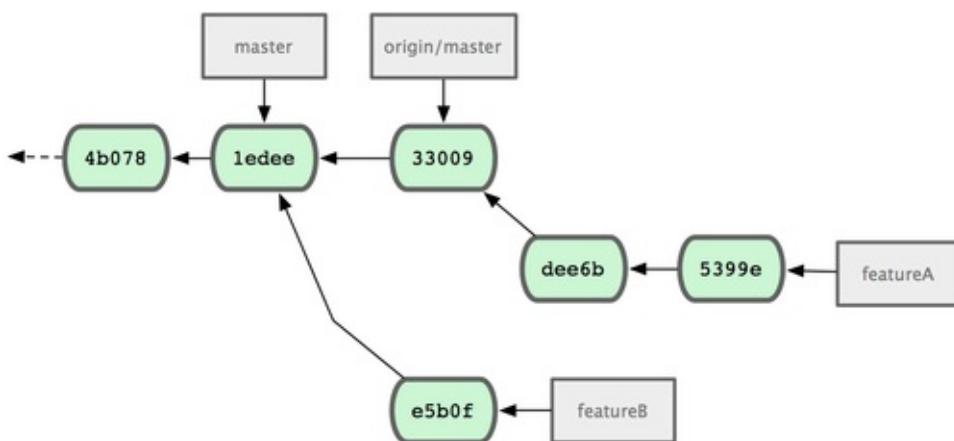


圖 5-17. featureA 重新衍合後的提交歷史

注意，此時推送分支必須使用 `-f` 選項（譯注：表示 force，不作檢查強制重寫）替換遠端已有的 `featureA` 分支，因為新的 `commit` 並非原來的後續更新。當然你也可以直接推送到另一個新的分支上去，比如稱作 `featureAv2`。

再考慮另一種情形：管理員看過第二個分支後覺得思路新穎，但想請你改下具體實現。我們只需以當前 `origin/master` 分支為基準，開始一個新的特性分支 `featureBv2`，然後把原來的 `featureB` 的更新拿過來，解決衝突，按要求重新實現部分代碼，然後將此特性分支推送上去：

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
$ (change implementation)
$ git commit
$ git push myfork featureBv2
```

這裡的 `--squash` 選項將目標分支上的所有更改全拿來應用到當前分支上，而 `--no-commit` 選項告訴 Git 此時無需自動生成和記錄（合併）提交。這樣，你就可以在原來代碼基礎上，繼續工作，直到最後一起提交。

好了，現在可以請管理員抓取 `featureBv2` 上的最新代碼了，如圖 5-18 所示：

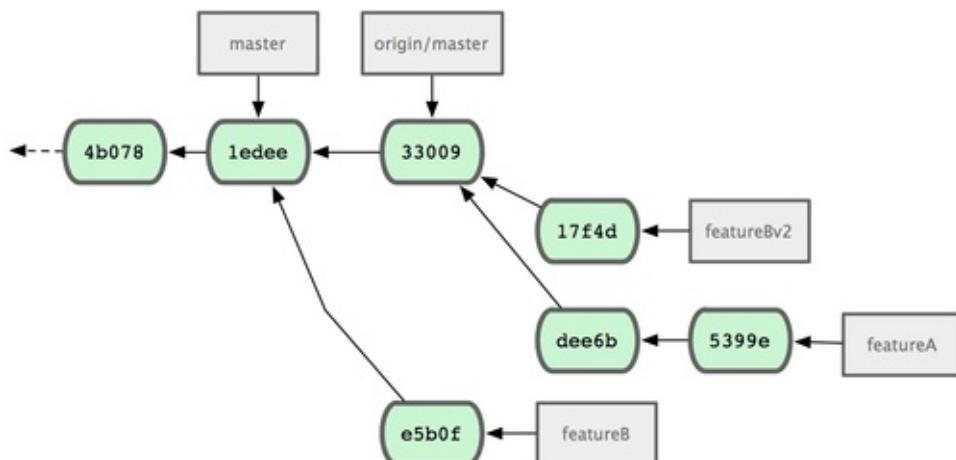


圖 5-18. `featureBv2` 之後的提交歷史

公開的大型專案

許多大型專案都會立有一套自己的接受補丁流程，你應該注意下其中細節。但多數專案都允許通過開發者郵寄清單接受補丁，現在我們來看具體例子。

整個工作流程類似上面的情形：為每個補丁創建獨立的特性分支，而不同之處在於如何提交這些補丁。不需要創建自己可寫的公共倉庫，也不用將自己的更新推送到自己的伺服器，你只需將每次提交的差異內容以電子郵件的方式依次發送到郵寄清單中即可。

```
$ git checkout -b topicA
$ (work)
$ git commit
$ (work)
$ git commit
```

如此一番後，有了兩個提交要發到郵寄清單。我們可以用 `git format-patch` 命令來生成 `mbox` 格式的檔然後作為附件發送。每個提交都會封裝為一個 `.patch` 尾碼的 `mbox` 檔，但其中只包含一封郵件，郵件標題就是提交消息（譯注：額外有首碼，看例子），郵件內容包含

補丁正文和 Git 版本號。這種方式的妙處在於接受補丁時仍可保留原來的提交消息，請看接下來的例子：

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

`format-patch` 命令依次創建補丁文件，並輸出檔案名。上面的 `-M` 選項允許 Git 檢查是否有對檔重命名的提交。我們來看看補丁檔的內容：

```
$ 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')
--
```

如果有額外資訊需要補充，但又不想放在提交消息中說明，可以編輯這些補丁檔，在第一個 `---` 行之前添加說明，但不要修改下面的補丁正文，比如例子中的 `Limit log functionality to the first 20` 部分。這樣，其它開發者能閱讀，但在採納補丁時不會將此合併進來。

你可以用郵件用戶端軟體發送這些補丁檔，也可以直接在命令列發送。有些所謂智慧的郵件用戶端軟體會自作主張幫你調整格式，所以粘貼補丁到郵件正文時，有可能會丟失分行符號和若干空格。Git 提供了一個通過 IMAP 發送補丁檔的工具。接下來我會演示如何通過 Gmail 的 IMAP 伺服器發送。另外，在 Git 原始程式碼中有個 `Documentation/SubmittingPatches` 檔，可以仔細讀讀，看看其它郵件程式的相關導引。

首先在 `~/.gitconfig` 檔中配置 `imap` 項。每個選項都可用 `git config` 命令分別設置，當然直接編輯檔添加以下內容更便捷：

```
[imap]
folder = "[Gmail]/Drafts"
host = imaps://imap.gmail.com
user = user@gmail.com
pass = p4ssw0rd
port = 993
sslverify = false
```

如果你的 IMAP 伺服器沒有啓用 SSL，就無需配置最後那兩行，並且 `host` 應該以 `imap://` 開頭而不再是 `s` 的 `imaps://`。保存設定檔後，就能用 `git send-email` 命令把補丁作為郵件依次發送到指定的 IMAP 伺服器上的資料夾中（譯注：這裡就是 Gmail 的 `[Gmail]/Drafts` 資料夾。但如果你的語言設置不是英文，此處的資料夾 `Drafts` 字樣會變為對應的語言。）：

```
$ 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
```

接下來，Git 會根據每個補丁依次輸出類似下面的日誌：

```
(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
```

小結

本節主要介紹了常見 Git 專案協作的工作流程，還有一些說明處理這些工作的命令和工具。接下來我們要看看如何維護 Git 項目，並成為一個合格的項目管理員，或是集成經理。

專案的管理

既然是相互協作，在貢獻代碼的同時，也免不了要維護管理自己的專案。像是怎麼處理別人用 `format-patch` 生成的補丁，或是集成遠端倉庫上某個分支上的變化等等。但無論是管理代碼倉庫，還是幫忙審核收到的補丁，都需要同貢獻者約定某種長期可持續的工作方式。

使用特性分支進行工作

如果想要集成新的代碼進來，最好局限在特性分支上做。臨時的特性分支可以讓你隨意嘗試，進退自如。比如碰上無法正常工作的補丁，可以先擱在那邊，直到有時間仔細核查修復為止。創建的分支可以用相關的主題關鍵字命名，比如 `ruby_client` 或者其它類似的描述性詞語，以幫助將來回憶。Git 專案本身還時常把分支名稱分置於不同命名空間下，比如 `sc/ruby_client` 就說明這是 `sc` 這個人貢獻的。現在從當前主幹分支為基礎，新建臨時分支：

```
$ git branch sc/ruby_client master
```

另外，如果你希望立即轉到分支上去工作，可以用 `checkout -b`：

```
$ git checkout -b sc/ruby_client master
```

好了，現在已經準備妥當，可以試著將別人貢獻的代碼合併進來了。之後評估一下有沒有問題，最後再決定是不是真的要併入主幹。

採納來自郵件的補丁

如果收到一個通過電郵發來的補丁，你應該先把它應用到特性分支上進行評估。有兩種應用補丁的方法：`git apply` 或者 `git am`。

使用 `apply` 命令應用補丁

如果收到的補丁文件是用 `git diff` 或由其它 Unix 的 `diff` 命令生成，就該用 `git apply` 命令來應用補丁。假設補丁文件存在 `/tmp/patch-ruby-client.patch`，可以這樣運行：

```
$ git apply /tmp/patch-ruby-client.patch
```

這會修改當前工作目錄下的檔，效果基本與運行 `patch -p1` 打補丁一樣，但它更為嚴格，且不會出現混亂。如果是 `git diff` 格式描述的補丁，此命令還會相應地添加，刪除，重命名檔。當然，普通的 `patch` 命令是不會這麼做的。另外請注意，`git apply` 是一個事務性操作的命令，也就是說，要麼所有補丁都打上去，要麼全部放棄。所以不會出現 `patch` 命令那樣，一部分檔打上了補丁而另一部分卻沒有，這樣一種不上不下的修訂狀態。所以總的來說，`git apply` 要比 `patch` 嚴謹許多。因為僅僅是更新當前的檔，所以此命令不會自動生成提交物件，你得手工緩存相應檔的更新狀態並執行提交命令。

在實際打補丁之前，可以先用 `git apply --check` 查看補丁是否能夠乾淨順利地應用到當前分支中：

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

如果沒有任何輸出，表示我們可以順利採納該補丁。如果有問題，除了報告錯誤資訊之外，該命令還會返回一個非零的狀態，所以在 shell 腳本裡可用於檢測狀態。

使用 `am` 命令應用補丁

如果貢獻者也用 Git，且擅於製作 `format-patch` 補丁，那你的合併工作將會非常輕鬆。因為這些補丁中除了檔內容差異外，還包含了作者資訊和提交消息。所以請鼓勵貢獻者用 `format-patch` 生成補丁。對於傳統的 `diff` 命令生成的補丁，則只能用 `git apply` 處理。

對於 `format-patch` 製作的新式補丁，應當使用 `git am` 命令。從技術上來說，`git am` 能夠讀取 `mbox` 格式的檔。這是種簡單的純文字檔，可以包含多封電郵，格式上用 `From` 加空格以及隨便什麼輔助資訊所組成的行作為分隔行，以區分每封郵件，就像這樣：

```
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
```

這是 `format-patch` 命令輸出的開頭幾行，也是一個有效的 `mbox` 檔案格式。如果有人用 `git send-email` 紿你發了一個補丁，你可以將此郵件下載到本地，然後運行 `git am` 命令來應用這個補丁。如果你的郵件用戶端能將多封電郵匯出為 `mbox` 格式的檔，就可以用 `git am` 一次性應用所有匯出的補丁。

如果貢獻者將 `format-patch` 生成的補丁檔上傳到類似 Request Ticket 一樣的任務處理系統，那麼可以先下載到本地，繼而使用 `git am` 應用該補丁：

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

你會看到它被乾淨地應用到本地分支，並自動創建了新的提交物件。作者資訊取自郵件頭 `From` 和 `Date`，提交消息則取自 `Subject` 以及正文中補丁之前的內容。來看具體實例，採納之前展示的那個 `mbox` 電郵補丁後，最新的提交對象為：

```
$ 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
```

`Commit` 部分顯示的是採納補丁的人，以及採納的時間。而 `Author` 部分則顯示的是原作者，以及創建補丁的時間。

有時，我們也會遇到打不上補丁的情況。這多半是因為主幹分支和補丁的基礎分支相差太遠，但也可能是因為某些依賴補丁還未應用。這種情況下，`git am` 會報錯並詢問該怎麼做：

```
$ 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".
```

Git 會在有衝突的檔裡加入衝突解決標記，這同合併或衍合操作一樣。解決的辦法也一樣，先編輯檔消除衝突，然後暫存檔，最後運行 `git am --resolved` 提交修正結果：

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

如果想讓 Git 更智慧地處理衝突，可以用 `-3` 選項進行三方合併。如果當前分支未包含該補丁的基礎代碼或其祖先，那麼三方合併就會失敗，所以該選項預設為關閉狀態。一般來說，如果該補丁是基於某個公開的提交製作而成的話，你總是可以通過同步來獲取這個共同祖

先，所以用三方合併選項可以解決很多麻煩：

```
$ 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.
```

像上面的例子，對於打過的補丁我又再打一遍，自然會產生衝突，但因為加上了 `-3` 選項，所以它很聰明地告訴我，無需更新，原有的補丁已經應用。

對於一次應用多個補丁時所用的 `mbox` 格式檔，可以用 `am` 命令的交互模式選項 `-i`，這樣就會在打每個補丁前停住，詢問該如何操作：

```
$ 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
```

在多個補丁要打的情況下，這是個非常好的辦法，一方面可以預覽下補丁內容，同時也可以有選擇性的接納或跳過某些補丁。

打完所有補丁後，如果測試下來新特性可以正常工作，那就可以安心地將當前特性分支合併到長期分支中去了。

檢出遠端分支

如果貢獻者有自己的 Git 倉庫，並將修改推送到此倉庫中，那麼當你拿到倉庫的訪問位址和對應分支的名稱後，就可以加為遠端分支，然後在本地進行合併。

比如，Jessica 發來一封郵件，說在她代碼庫中的 `ruby-client` 分支上已經實現了某個非常棒的新功能，希望我們能幫忙測試一下。我們可以先把她的倉庫加為遠端倉庫，然後抓取資料，完了再將她所說的分支檢出到本地來測試：

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

若是不久她又發來郵件，說還有個很棒的功能實現在另一分支上，那我們只需重新抓取下最新資料，然後檢出那個分支到本地就可以了，無需重複設置遠端倉庫。

這種做法便於同別人保持長期的合作關係。但前提是要求貢獻者有自己的伺服器，而我們也需要為每個人建一個遠端分支。有些貢獻者提交代碼補丁並不是很頻繁，所以通過郵件接收補丁效率會更高。同時我們自己也不會希望建上百來個分支，卻只從每個分支取一兩個補丁。但若是用腳本程式來管理，或直接使用代碼倉庫託管服務，就可以簡化此過程。當然，選擇何種方式取決於你和貢獻者的喜好。

使用遠端分支的另外一個好處是能夠得到提交歷史。不管代碼合併是不是會有問題，至少我們知道該分支的歷史分叉點，所以默認會從共同祖先開始自動進行三方合併，無需 `-3` 選項，也不用像打補丁那樣祈禱存在共同的基準點。

如果只是臨時合作，只需用 `git pull` 命令抓取遠端倉庫上的資料，合併到本地臨時分支就可以了。一次性的抓取動作自然不會把該倉庫位址加為遠程倉庫。

```
$ git pull git://github.com/onetimeguy/project.git
From git://github.com/onetimeguy/project
 * branch            HEAD      -> FETCH_HEAD
Merge made by recursive.
```

決斷代碼取捨

現在特性分支上已合併好了貢獻者的代碼，是時候決斷取捨了。本節將回顧一些之前學過的命令，以看清將要合併到主幹的是哪些代碼，從而理解它們到底做了些什麼，是否真的要併入。

一般我們會先看下，特性分支上都有哪些新增的提交。比如在 `contrib` 特性分支上打了兩個補丁，僅查看這兩個補丁的提交資訊，可以用 `--not` 選項指定要遮罩的分支 `master`，這樣就會剔除重複的提交歷史：

```
$ 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
```

還可以查看每次提交的具體修改。請牢記，在 `git log` 後加 `-p` 選項將展示每次提交的內容差異。

如果想看當前分支同其他分支合併時的完整內容差異，有個小竅門：

```
$ git diff master
```

雖然能得到差異內容，但請記住，結果有可能和我們的預期不同。一旦主幹 `master` 在特性分支創建之後有所修改，那麼通過 `diff` 命令來比較的，是最新主幹上的提交快照。顯然，這不是我們所要的。比方在 `master` 分支中某個檔裡添了一行，然後運行上面的命令，簡單的比較最新快照所得到的結論只能是，特性分支中刪除了這一行。

這個很好理解：如果 `master` 是特性分支的直接祖先，不會產生任何問題；如果它們的提交歷史在不同的分叉上，那麼產生的內容差異，看起來就像是增加了特性分支上的新代碼，同時刪除了 `master` 分支上的新代碼。

實際上我們真正想要看的，是新加入到特性分支的代碼，也就是合併時會併入主幹的代碼。所以，準確地講，我們應該比較特性分支和它同 `master` 分支的共同祖先之間的差異。

我們可以手工定位它們的共同祖先，然後與之比較：

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

但這麼做很麻煩，所以 Git 提供了便捷的 `...` 語法。對於 `diff` 命令，可以把 `...` 加在原始分支（擁有共同祖先）和當前分支之間：

```
$ git diff master...contrib
```

現在看到的，就是實際將要引入的新代碼。這是一個非常有用的命令，應該牢記。

代碼集成

一旦特性分支準備停當，接下來的問題就是如何集成到更靠近主線的分支中。此外還要考慮維護專案的總體步驟是什麼。雖然有很多選擇，不過我們這裡只介紹其中一部分。

合併流程

一般最簡單的情形，是在 `master` 分支中維護穩定代碼，然後在特性分支上開發新功能，或是審核測試別人貢獻的代碼，接著將它併入主幹，最後刪除這個特性分支，如此反復。來看示例，假設當前代碼庫中有兩個分支，分別為 `ruby_client` 和 `php_client`，如圖 5-19 所示。然後先把 `ruby_client` 合併進主幹，再合併 `php_client`，最後的提交歷史如圖 5-20 所示。

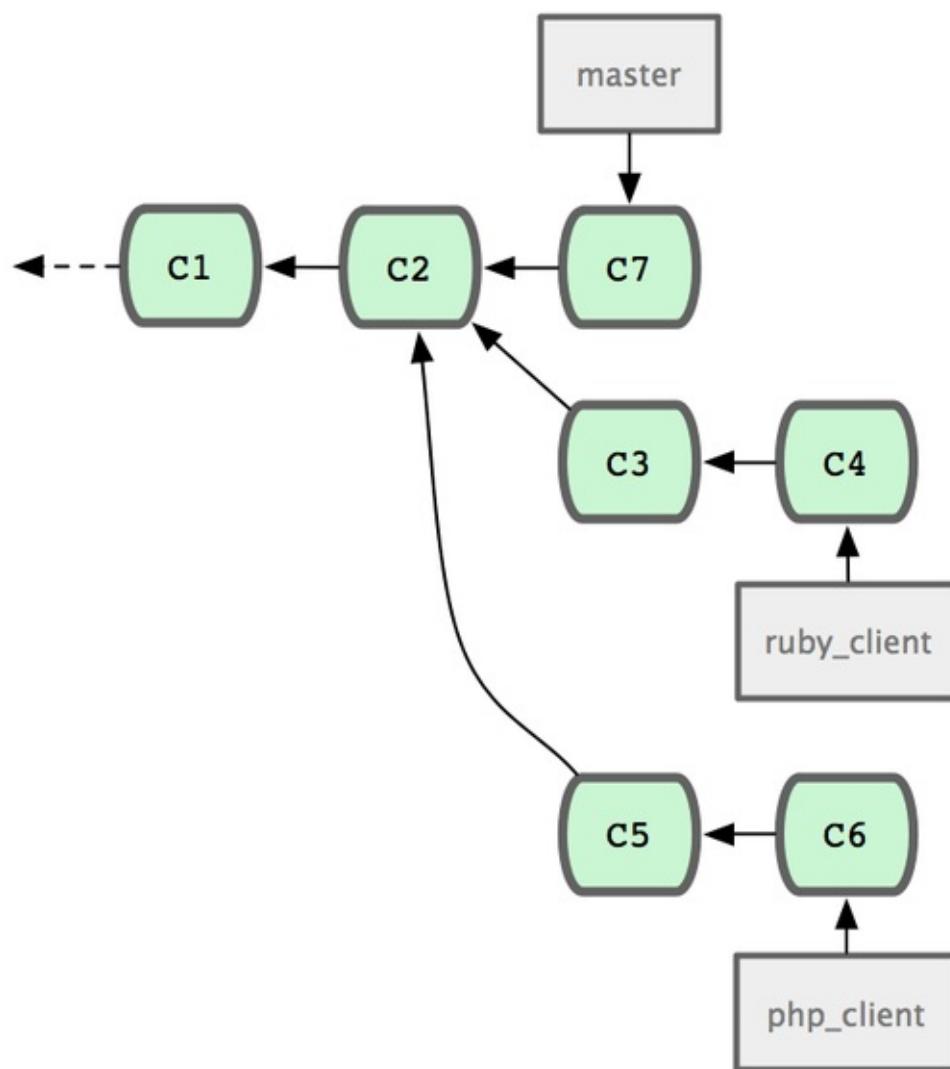


圖 5-19. 多個特性分支

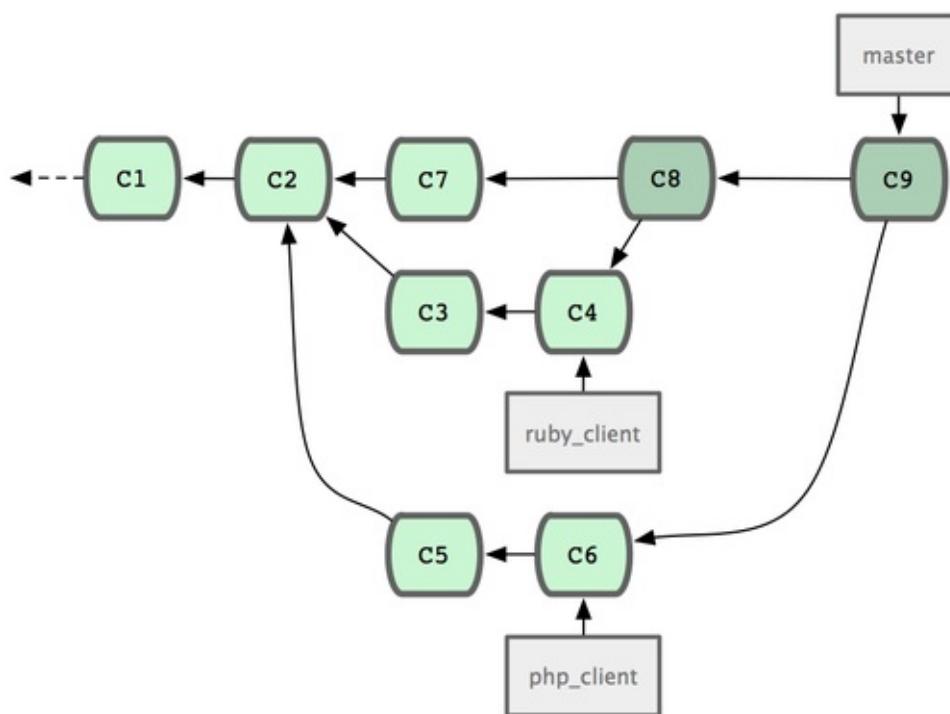


圖 5-20. 合併特性分支之後

這是最簡單的流程，所以在處理大一些的項目時可能會有問題。

對於大型專案，至少需要維護兩個長期分支 `master` 和 `develop`。新代碼（圖 5-21 中的 `ruby_client`）將首先併入 `develop` 分支（圖 5-22 中的 `c8`），經過一個階段，確認 `develop` 中的代碼已穩定到可發行時，再將 `master` 分支快進到穩定點（圖 5-23 中的 `c8`）。而平時這兩個分支都會被推送到公開的代碼庫。

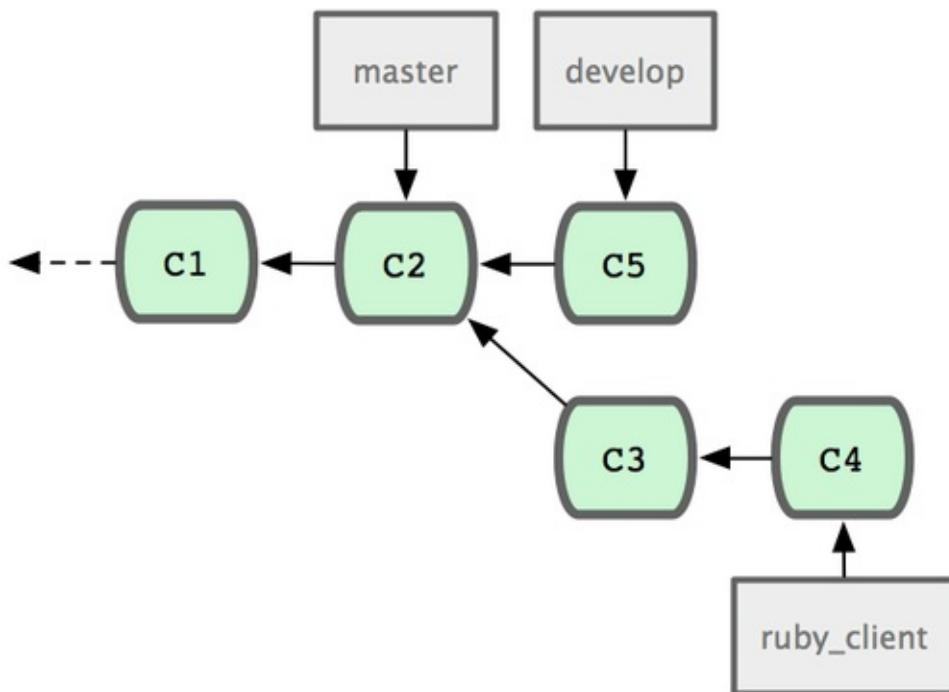


圖 5-21. 特性分支合併前

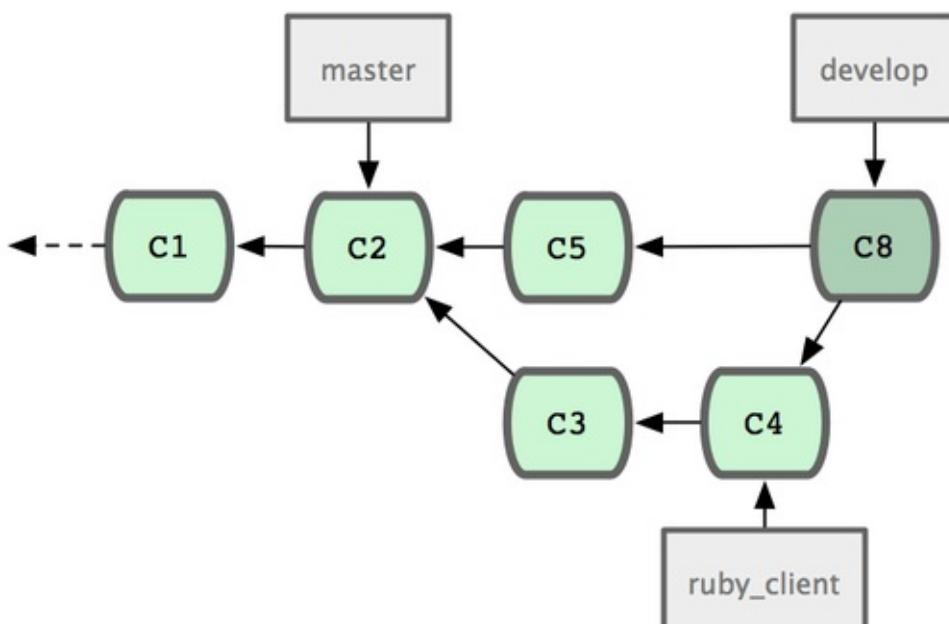


圖 5-22. 特性分支合併後

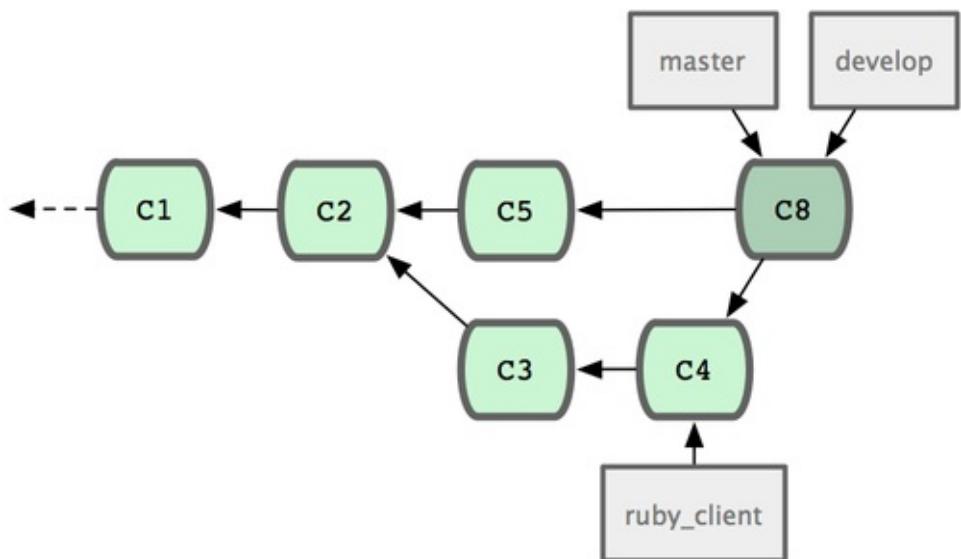


圖 5-23. 特性分支發佈後

這樣，在人們克隆倉庫時就有兩種選擇：既可檢出最新穩定版本，確保正常使用；也能檢出開發版本，試用最前沿的新特性。你也可以擴展這個概念，先將所有新代碼合併到臨時特性分支，等到該分支穩定下來並通過測試後，再併入 `develop` 分支。然後，讓時間檢驗一切，如果這些代碼確實可以正常工作相當長一段時間，那就有理由相信它已經足夠穩定，可以放心併入主幹分支發佈。

大專案的合併流程

Git 專案本身有四個長期分支：用於發佈的 `master` 分支、用於合併基本穩定特性的 `next` 分支、用於合併仍需改進特性的 `pu` 分支（`pu` 是 `proposed updates` 的縮寫），以及用於除錯維護的 `maint` 分支（`maint` 取自 `maintenance`）。維護者可以按照之前介紹的方法，將貢獻者的代碼引入為不同的特性分支（如圖 5-24 所示），然後測試評估，看哪些特性能穩定工作，哪些還需改進。穩定的特性可以併入 `next` 分支，然後再推送到公共倉庫，以供其他人試用。

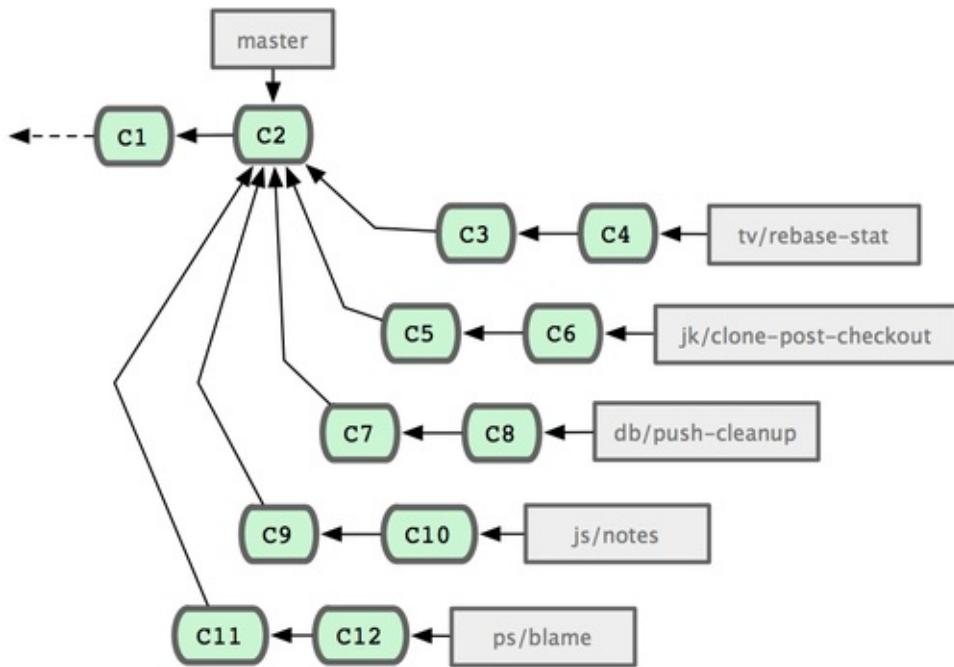


圖 5-24. 管理複雜的並行貢獻

仍需改進的特性可以先併入 `pu` 分支。直到它們完全穩定後再併入 `master`。同時一併檢查下 `next` 分支，將足夠穩定的特性也併入 `master`。所以一般來說，`master` 始終是在快進，`next` 偶爾做下衍合，而 `pu` 則是頻繁衍合，如圖 5-25 所示：

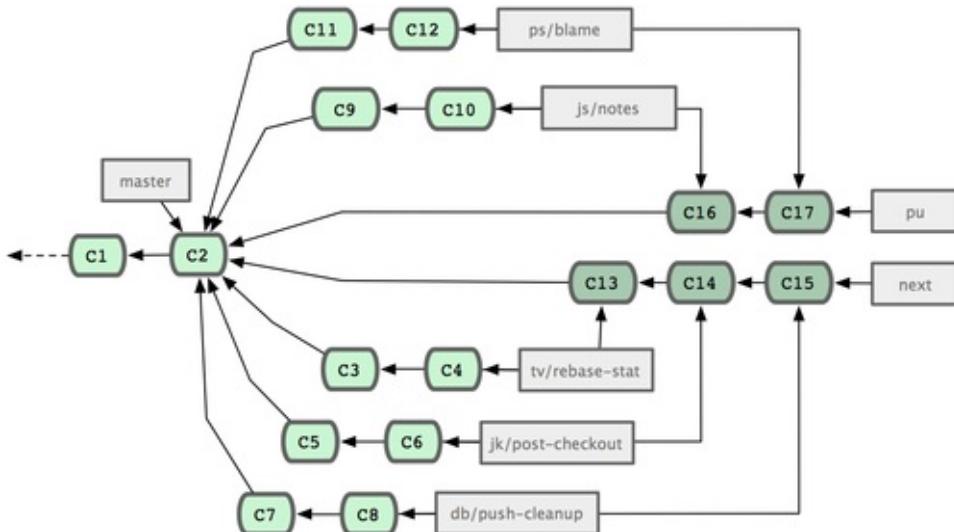


圖 5-25. 將特性併入長期分支

併入 `master` 後的特性分支，已經無需保留分支索引，放心刪除好了。Git 專案還有一個 `maint` 分支，它是以最近一次發行版本為基礎分化而來的，用於維護除錯補丁。所以克隆 Git 項目倉庫後會得到這四個分支，通過檢出不同分支可以瞭解各自進展，或是試用前沿特性，或是貢獻代碼。而維護者則通過管理這些分支，逐步有序地併入協力廠商貢獻。

衍合與挑揀（cherry-pick）的流程

一些維護者更喜歡衍合或者挑揀貢獻者的代碼，而不是簡單的合併，因為這樣能夠保持線性的提交歷史。如果你完成了一個特性的開發，並決定將它引入到主幹代碼中，你可以轉到那個特性分支然後執行衍合命令，好在你的主幹分支上（也可能是 `develop` 分支之類的）重新提交這些修改。如果這些代碼工作得很好，你就可以快進 `master` 分支，得到一個線性的提交歷史。

另一個引入代碼的方法是挑揀。挑揀類似於針對某次特定提交的衍合。它首先提取某次提交的補丁，然後試著應用在當前分支上。如果某個特性分支上有許多個commits，但你只想引入其中之一就可以使用這種方法。也可能僅僅是因為你喜歡用挑揀，討厭衍合。假設你有一個類似圖 5-26 的工程。

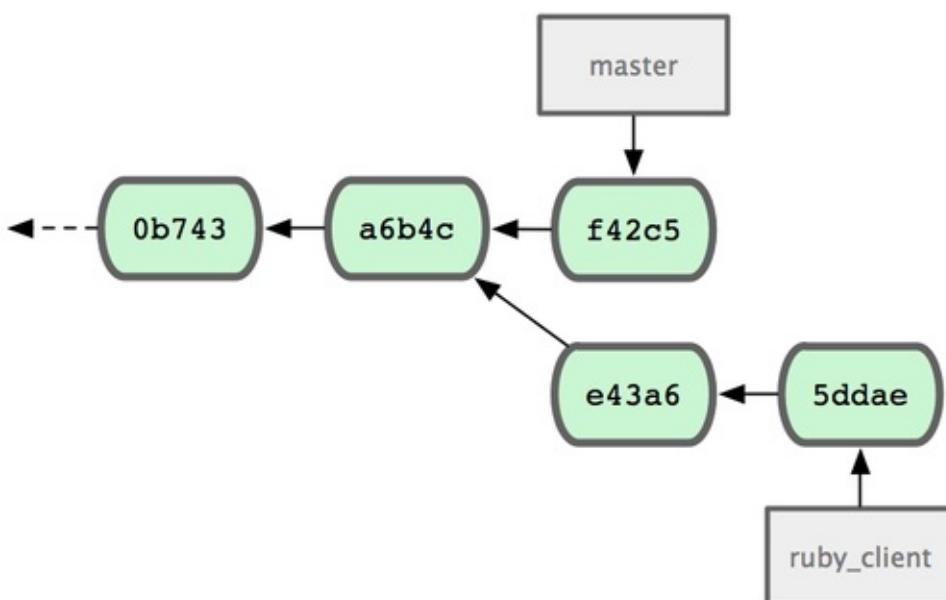


圖 5-26. 挑揀 (cherry-pick) 之前的歷史

如果你希望拉取 `e43a6` 到你的主幹分支，可以這樣：

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

這將會引入 `e43a6` 的代碼，但是會得到不同的SHA-1值，因為應用日期不同。現在你的歷史看起來像圖 5-27.

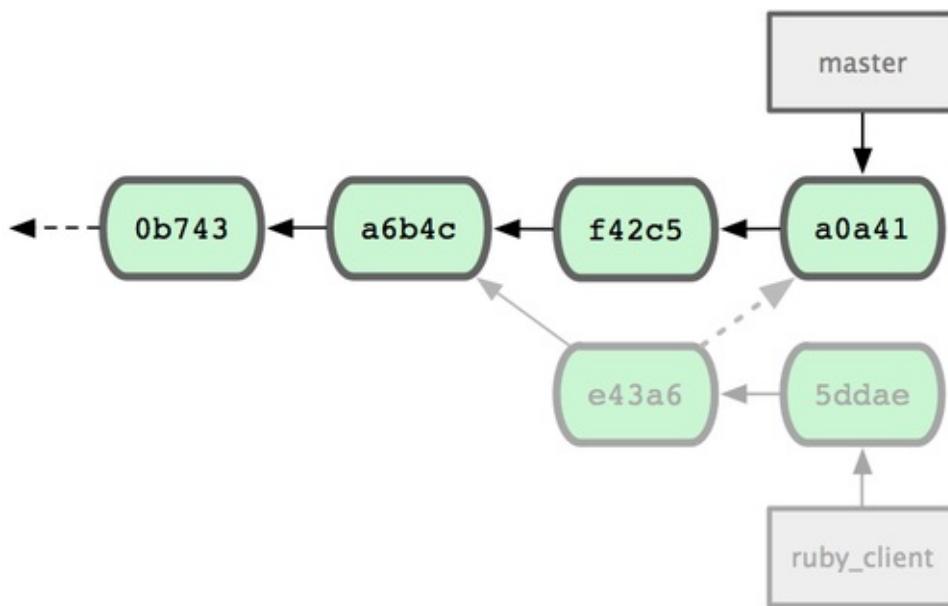


圖 5-27. 挑揀（cherry-pick）之後的歷史

現在，你可以刪除這個特性分支並丟棄你不想引入的那些commit。

給發行版本簽名

你可以刪除上次發佈的版本並重新打標籤，也可以像第二章所說的那樣建立一個新的標籤。如果你決定以維護者的身份給發行版本簽名，應該這樣做：

```
$ 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
```

完成簽名之後，如何分發PGP公開金鑰（public key）是個問題。（譯者注：分發公開金鑰是為了驗證標籤）。還好，Git的設計者想到了解決辦法：可以把key（既公開金鑰）作為blob變數寫入Git庫，然後把它的內容直接寫在標籤裡。`gpg --list-keys` 命令可以顯示出你所擁有的key：

```
$ 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]
```

然後，匯出key的內容並經由管道符傳遞給`git hash-object`，之後鑰匙會以blob類型寫入Git中，最後返回這個blob量的SHA-1值：

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

現在你的Git已經包含了這個key的內容了，可以通過不同的SHA-1值指定不同的key來創建標籤。

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

在運行 `git push --tags` 命令之後，`maintainer-pgp-pub` 標籤就會公佈給所有人。如果有人想要校驗標籤，他可以使用如下命令導入你的key：

```
$ git show maintainer-pgp-pub | gpg --import
```

人們可以用這個key校驗你簽名的所有標籤。另外，你也可以在標籤資訊裡寫入一個操作嚮導，用戶只需要運行 `git show <tag>` 查看標籤資訊，然後按照你的嚮導就能完成校驗。

生成內部版本號

因為Git不會為每次提交自動附加類似'v123'的遞增序列，所以如果你想要得到一個便於理解的提交號可以運行 `git describe` 命令。Git將會返回一個字串，由三部分組成：最近一次標定的版本號，加上自那次標定之後的提交次數，再加上一段所描述的提交的SHA-1值：

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

這個字串可以作為快照的名字，方便人們理解。如果你的Git是你自己下載源碼然後編譯安裝的，你會發現 `git --version` 命令的輸出和這個字串差不多。如果在一個剛剛打完標籤的提交上運行 `describe` 命令，只會得到這次標定的版本號，而沒有後面兩項資訊。

`git describe` 命令只適用於有標注的標籤（通過 `-a` 或者 `-s` 選項創建的標籤），所以發行版本的標籤都應該是帶有標注的，以保證 `git describe` 能夠正確的執行。你也可以把這個字串作為 `checkout` 或者 `show` 命令的目標，因為他們最終都依賴於一個簡短的SHA-1值，當然如果這個SHA-1值失效他們也跟著失效。最近Linux內核為了保證SHA-1值的唯一性，將位數由8位擴展到10位，這就導致擴展之前的 `git describe` 輸出完全失效了。

準備發佈

現在可以發佈一個新的版本了。首先要將代碼的壓縮包歸檔，方便那些可憐的還沒有使用Git的人們。可以使用 `git archive`：

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

這個壓縮包解壓出來的是一個資料夾，裡面是你項目的最新代碼快照。你也可以用類似的方法建立一個zip壓縮包，在 `git archive` 加上 `--format=zip` 選項：

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

現在你有了一個tar.gz壓縮包和一個zip壓縮包，可以把他們上傳到你網站上或者用e-mail發給別人。

製作簡報

是時候通知郵寄清單裡的朋友們來檢驗你的成果了。使用 `git shortlog` 命令可以方便快捷的製作一份修改日誌（changelog），告訴大家上次發佈之後又增加了哪些特性和修復了哪些bug。實際上這個命令能夠統計給定範圍內的所有提交；假如你上一次發佈的版本是v1.0.1，下面的命令將給出自從上次發佈之後的所有提交的簡介：

```
$ 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
```

這就是自從v1.0.1版本以來的所有提交的簡介，內容按照作者分組，以便你能快速的發e-mail給他們。

小結

你學會了如何使用 Git 為專案做貢獻，也學會了如何使用 Git 維護你的專案。恭喜！你已經成為一名高效的開發者。在下一章你將學到更強大的工具來處理更加複雜的問題，之後你會變成一位 Git 大師。

Git 工具

現在，你已經學習了管理或者維護 Git 倉庫，實現代碼控制所需的大多數日常命令和工作流程。你已經完成了跟蹤和提交檔案的基本任務，並且發揮了暫存區(staging area)和羽量級的特性分支及合併的威力。

接下來你將領略到一些 Git 可以實現的非常強大的功能，這些功能你可能並不會在日常操作中使用，但在某些時候你也許會需要。

選擇修訂版本

Git 允許你通過幾種方法來指明特定的或者一定範圍內的提交。瞭解它們並不是必需的，但是瞭解一下總沒壞處。

單個修訂版本

顯然你可以使用給出的 SHA-1 值來指明一次提交，不過也有更加人性化的方法來做同樣的事。本節概述了指明單個提交的諸多方法。

簡短的 SHA

Git 很聰明，它能夠通過你提供的前幾個字元來識別你想要的那次提交，只要你提供的那部分 SHA-1 不短於四個字元，並且沒有歧義——也就是說，當前倉庫中只有一個物件以這段 SHA-1 開頭。

例如，想要查看一次指定的提交，假設你執行 `git log` 命令並找到你增加了功能的那次提交：

```
$ 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
```

假設是 `1c002dd`。如果你想 `git show` 這次提交，下面命令的作用是相同的（假設簡短的版本沒有歧義）：

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git 可以為你的 SHA-1 值生成出簡短且唯一的縮寫。如果你傳遞 `--abbrev-commit` 給 `git log` 命令，輸出結果裡就會使用簡短且唯一的值；它預設使用七個字元來表示，不過必要時為了避免 SHA-1 的歧義，會增加字元數：

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

通常在一個專案中，使用八到十個字元來避免 SHA-1 歧義已經足夠了。最大的 Git 專案之一，Linux 內核，目前也需要最長 40 個字元中的 12 個字元來保持唯一性。

關於 **SHA-1** 的簡短說明

許多人可能會擔心一個問題：在隨機的偶然情況下，在他們的倉庫裡會出現兩個具有相同 SHA-1 值的物件。那會怎麼樣呢？

如果你真的向倉庫裡提交了一個跟之前的某個物件具有相同 SHA-1 值的物件，Git 將會發現之前的那個物件已經存在在 Git 資料庫中，並認為它已經被寫入了。如果什麼時候你想再次檢出那個物件時，你會總是得到先前的那個物件的資料。

不過，你應該瞭解到，這種情況發生的概率是多麼微小。SHA-1 摘要長度是 20 位元組，也就是 160 位元。為了保證有 50% 的概率出現一次衝突，需要 2^{80} 個隨機雜湊的物件（計算衝突機率的公式是 $p = (n(n-1)/2) * (1/2^{160})$ ）。 2^{80} 是 1.2×10^{24} ，也就是一億億億，那是地球上沙粒總數的 1200 倍。

現在舉例說一下怎樣才能產生一次 SHA-1 衝突。如果地球上 65 億的人類都在程式設計，每人每秒都在產生相當於整個 Linux 內核歷史（一百萬個 Git 物件）的代碼，並將之提交到一個巨大的 Git 倉庫裡面，那將花費 5 年的時間才會產生足夠的物件，使其擁有 50% 的概率產生一次 SHA-1 物件衝突。這要比你程式設計團隊的成員同一個晚上在互不相干的意外中被狼襲擊並殺死的機率還要小。

分支引用 (**Branch References**)

指明一次提交的最直接的方法是有一個指向它的分支引用。這樣，你就可以在任何需要一個提交物件或者 SHA-1 值的 Git 命令中使用該分支名稱了。如果你想要顯示一個分支的最後一次提交的物件，例如假設 `topic1` 分支指向 `ca82a6d`，那麼下面的命令是相等的：

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

如果你想知道某個分支指向哪個特定的 SHA，或者想看任何一個例子中被簡寫的 SHA-1，你可以使用一個叫做 `rev-parse` 的 Git plumbing 工具。在第 9 章你可以看到關於 plumbing 工具的更多信息；簡單來說，`rev-parse` 是為了底層操作而不是日常操作設計的。不過，有時你想看 Git 現在到底處於什麼狀態時，它可能會很有用。現在，你可以對你的分支執行 `rev-parse`：

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

引用日誌(RefLog)裡的簡稱

在你工作的同時，Git 在後臺的工作之一就是保存一份引用日誌(reflog)——一份記錄最近幾個月你的 HEAD 和分支引用的日誌。

你可以使用 `git reflog` 來查看引用日誌：

```
$ 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
```

每次你的分支頂端因為某些原因被修改時，Git 就會為你將資訊保存在這個臨時歷史記錄裡面。你也可以使用這份資料來指明更早的分支。如果你想查看倉庫中 HEAD 在五次前的值，你可以使用引用日誌的輸出中的 `@{n}` 引用：

```
$ git show HEAD@{5}
```

你也可以使用這個語法來查看一定時間前分支指向哪裡。例如，想看你的 `master` 分支昨天在哪，你可以輸入

```
$ git show master@{yesterday}
```

它就會顯示昨天分支的頂端在哪。這項技術只對還在你引用日誌裡的資料有用，所以不能用來查看比幾個月前還早的提交。

想要看類似於 `git log` 輸出格式的引用日誌資訊，你可以執行 `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'
```

需要注意的是，日誌引用資訊只存在於本地——這是一個你在倉庫裡做過什麼的日誌。這些引用不會和其他人的倉庫拷貝裡的相同；當你新 `clone` 一個倉庫的時候，引用日誌是空的，因為你在倉庫裡還沒有操作。只有你克隆了一個專案至少兩個月，`git show HEAD@{2.months.ago}` 才會有用——如果你是五分鐘前克隆的倉庫，將不會有結果回傳。

祖先引用 (Ancestry References)

另一種指明某次提交的常用方法是通過它的祖先。如果你在引用最後加上一個 `^`，Git 將其理解為此次提交的父提交。假設你的專案歷史是這樣的：

```
$ 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
```

那麼，想看上一次提交，你可以使用 `HEAD^`，意思是「HEAD 的父提交」：

```
$ 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'
```

你也可以在 `^` 後添加一個數字——例如，`d921970^2` 意思是「`d921970` 的第二父提交」。這種語法只在合併提交時有用，因為合併提交可能有多個父提交。第一父提交是你合併時所在分支，而第二父提交是你所合併進來的分支：

```
$ 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
```

另外一個指明祖先提交的方法是 `~`。這也是指向第一父提交，所以 `HEAD~` 和 `HEAD^` 是相等的。當你指定數字的時候就明顯不一樣了。`HEAD~2` 是指「第一父提交的第一父提交」，也就是「祖父提交」——它會根據你指定的次數檢索第一父提交。例如，在上面列出的歷史記錄裡面，`HEAD~3` 會是

```
$ git show HEAD~3
commit 1c3618887afb5fbcbbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500

ignore *.gem
```

也可以寫成 `HEAD^{~3}`，同樣是第一父提交的第一父提交：

```
$ git show HEAD^{~3}
commit 1c3618887afb5fbcbbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500

ignore *.gem
```

你也可以混合使用這些語法——你可以通過 `HEAD~3^2` 指明先前引用的第二父提交（假設它是一個合併提交），依此類推。

提交範圍

現在你已經可以指明單次的提交，讓我們來看看怎樣指明一定範圍的提交。這在你管理分支的時候尤顯重要——如果你有很多分支，你可以指明範圍來圈定一些問題的答案，比如：「這個分支上我有哪些工作還沒合併到主分支的？」

雙點

最常用的指明範圍的方法是雙點的語法。這種語法主要是讓 Git 區分出可從一個分支中獲得而不能從另一個分支中獲得的提交。例如，假設你有類似於圖 6-1 的提交歷史。

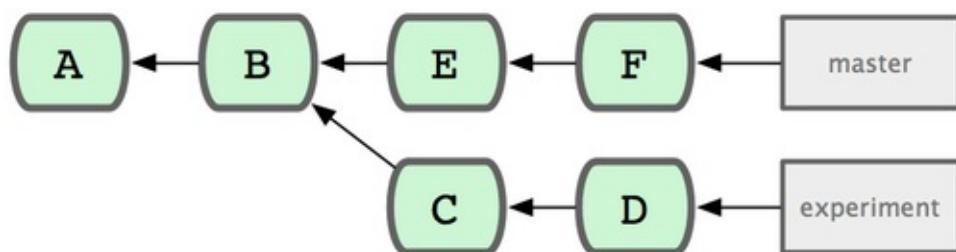


Figure 6-1. 範圍選擇的提交歷史實例

你想要查看你的試驗分支(experiment)上哪些沒有被提交到主分支，那麼你就可以使用 `master..experiment` 來讓 Git 顯示這些提交的日誌——這句話的意思是「所有可從 experiment 分支中獲得而不能從 master 分支中獲得的提交」。為了使例子簡單明瞭，我使用了圖示中提交物件的字母，來代替它們在實際的日誌輸出裏的顯示順序：

```
$ git log master..experiment
D
C
```

另一方面，如果你想看相反的——所有在 `master` 而不在 `experiment` 中的分支——你可以交換分支的名字。`experiment..master` 顯示所有可在 `master` 獲得而在 `experiment` 中不能獲得的提交：

```
$ git log experiment..master
F
E
```

這在你想將 `experiment` 分支維持在最新狀態，並預覽你將合併的提交的時候特別有用。這個語法的另一種常見用途是查看你將把什麼推送到遠端：

```
$ git log origin/master..HEAD
```

這條命令顯示任何在你當前分支上而在遠端 `origin` 上的 `master` 分支上的提交。如果你執行 `git push` 並且你的當前分支正在追蹤 `origin/master`，被 `git log origin/master..HEAD` 列出的提交就是將被傳輸到伺服器上的提交。你也可以省略語法中的一邊讓 Git 來假定它是 `HEAD`。例如，輸入 `git log origin/master..` 將得到和上面的例子一樣的結果——Git 使用 `HEAD` 來代替不存在的一邊。

多點

雙點語法就像速記一樣有用；但是你也許會想針對兩個以上的分支來指明修訂版本，比如查看哪些提交被包含在某些分支中的一個，但是不在你當前的分支上。Git 允許你在引用前使用 `^` 字元或者 `--not` 指明你不希望提交被包含其中的分支。因此下面三個命令是等同的：

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

這樣很好，因為它允許你在查詢中指定多於兩個的引用，而這是雙點語法所做不到的。例如，如果你想查找所有從 `refA` 或 `refB` 包含的但是不被 `refC` 包含的提交，你可以輸入下面中的一個

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

這建立了一個非常強大的修訂版本查詢系統，應該可以幫助你了解你的分支裡有些什麼東西。

三點

最後一種主要的範圍選擇語法是三點語法，這個可以指定被兩個引用中的一個包含但又不被兩者同時包含的分支。回過頭來看一下圖6-1裡所列的提交歷史的例子。如果你想查看 `master` 或者 `experiment` 中包含的但不是兩者共有的引用，你可以執行

```
$ git log master...experiment
F
E
D
C
```

這個再次給出你普通的 `log` 輸出但是只顯示那四次提交的資訊，按照傳統的提交日期排列。

這種情形下，`log` 命令的一個常用參數是 `--left-right`，它會顯示每個提交到底處於哪一側的分支。這使得資料更加有用。

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

有了以上工具，讓 Git 知道你要察看哪些提交就容易得多了。

互動式暫存

Git 提供了很多腳本來輔助某些命令列任務。這裡，你將看到一些互動式命令，它們幫助你方便地構建只包含特定組合和部分檔案的提交。在你修改了一大批檔案然後決定將這些變更分佈在幾個有聚焦的提交而不是單個又大又亂的提交時，這些工具非常有用。用這種方法，你可以確保你的提交在邏輯上劃分為相應的變更集合，以便於和你一起工作的開發者審閱。如果你執行 `git add` 時加上 `-i` 或者 `--interactive` 選項，Git 就進入了一個互動式的 shell 模式，顯示一些類似於下面的資訊：

```
$ 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>
```

你會看到這個命令以一個完全不同的視圖顯示了你的暫存區——主要是你通過 `git status` 得到的那些資訊但是稍微簡潔但資訊更加豐富一些。它在左側列出了你暫存的變更，在右側列出了未被暫存的變更。

在這之後是一個命令區。這裡你可以做很多事情，包括暫存檔案(stage)、撤回檔案(unstage)、暫存部分檔案、加入未被追蹤的文件、查看暫存文件的差別。

暫存和撤回檔案

如果你在 `What now>` 的提示後輸入 `2` 或者 `u`，這個腳本會提示你那些檔你想要暫存：

```
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>>
```

如果想暫存 `TODO` 和 `index.html`，你可以輸入相應的編號：

```
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>>
```

每個檔旁邊的 `*` 表示選中的檔將被暫存。如果你在 `update>>` 提示後直接敲入 `Enter`，Git 會替你把所有選中的內容暫存：

```
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
```

現在你可以看到 `TODO` 和 `index.html` 檔被暫存了，同時 `simplegit.rb` 檔仍然未被暫存。如果這時你想要撤回 `TODO` 檔，就使用 `3` 或者 `r`（代表 `revert`，恢復）選項：

```
*** 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
```

再次查看 Git 的狀態，你會看到你已經撤回了 `TODO` 檔

```
*** 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
```

要查看你暫存內容的差異，你可以使用 `6` 或者 `d`（表示`diff`）命令。它會顯示你暫存檔的列表，你可以選擇其中的幾個，顯示其被暫存的差異。這跟你在命令列下指定 `git diff --cached` 非常相似：

```
*** 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">
```

通過這些基本命令，你可以使用互動式增加模式更加方便地處理暫存區。

暫存補丁 (Staging Patches)

只讓 Git 將檔案的某些部分暫存，而忽略其他部份也是有可能的。例如，你對 `simplegit.rb` 檔作了兩處修改但是只想暫存其中一個而忽略另一個，在 Git 中實現這一點非常容易。在互動式的提示符下，輸入 `5` 或者 `p`（表示 `patch`，補丁）。Git 會詢問哪些檔你希望部分暫存；然後對於被選中檔案的每一節，他會逐個顯示檔案的差異區塊並詢問你是否希望暫存他們：

```

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,?]?

```

此處你有很多選擇。輸入 `?` 可以顯示清單：

```

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

```

如果你想暫存各個區塊，通常你會輸入 `y` 或者 `n`，但是暫存特定檔案裡的全部區塊或者暫時跳過對一個區塊的處理同樣也很有用。如果你暫存了檔案的一個部分而保留另外一個部分不被暫存，你的狀態輸出看起來會是這樣：

```

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

```

`simplegit.rb` 的狀態非常有意思。它顯示有幾行被暫存了，有幾行沒有。你部分地暫存了這個檔。這時候，你可以退出互動式腳本然後執行 `git commit` 來提交部分暫存的檔。

最後，你也可以不通過互動式增加的模式來實現檔案部分暫存——你可以在命令列下使用 `git add -p` 或者 `git add --patch` 來啟動同樣的腳本。

儲藏 (Stashing)

經常有這樣的事情發生，當你正在進行專案中某一部分的工作，裡面的東西處於一個比較雜亂的狀態，而你想轉到其他分支上進行一些工作。問題是，你不想只為了待會要回到這個工作點，就把做到一半的工作進行提交。解決這個問題的辦法就是 `git stash` 命令。

「儲藏」可以獲取你工作目錄的 `dirty state`——也就是你修改過的被追蹤檔和暫存的變更——並將它保存到一個未完成變更的堆疊(stack)中，隨時可以重新應用。

儲藏你的工作

為了演示這一功能，你可以進入你的專案，在一些檔上進行工作，有可能還暫存其中一個變更。如果你執行 `git status`，你可以看到你的 `dirty state`：

```
$ git status
# 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)
#
#       modified:   lib/simplegit.rb
#
```

現在你想切換分支，但是你還不想提交你正在進行中的工作；所以你儲藏這些變更。為了往堆疊推送一個新的儲藏，執行 `git stash`：

```
$ 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")
```

你的工作目錄就乾淨了：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

這時，你可以方便地切換到其他分支工作；你的變更都保存在堆疊上。要查看現有的儲藏，你可以使用 `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
```

在這個案例中，之前已經進行了兩次儲藏，所以你可以取得三個不同的儲藏。你可以重新應用你剛剛的儲藏，所採用的命令就是原本 `stash` 命令輸出的輔助訊息裡提示的：`git stash apply`。如果你想應用較舊的儲藏，你可以通過名字指定它，像這樣：`git stash apply stash@{2}`。如果你不指明，Git 預設使用最近的儲藏並嘗試應用它：

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

你可以看到 Git 重新修改了你所儲藏的那些當時尚未提交的檔。在這個案例裡，你嘗試應用儲藏的工作目錄是乾淨的，並且屬於同一分支；但是一個乾淨的工作目錄和應用到相同的分支上並不是應用儲藏的必要條件。你可以在其中一個分支上保留一份儲藏，隨後切換到另外一個分支，再重新應用這些變更。在工作目錄裡包含已修改、未提交的檔時，你也可以應用儲藏——Git 會給出合併衝突，如果有任何變更無法乾淨地被應用。

對檔案的變更被重新應用，但是被暫存的檔沒有重新被暫存。想那樣的話，你必須在執行 `git stash apply` 命令時帶上一個 `--index` 的選項來重新應用被暫存的變更。如果你是這麼做的，你應該已經回到你原來的位置：

```
$ 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)
#
#       modified:   lib/simplegit.rb
#
```

`apply` 選項只嘗試應用儲藏的工作——儲藏的內容仍然在堆疊上。要移除它，你可以執行 `git stash drop`，加上你希望移除的儲藏的名字：

```
$ 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)
```

你也可以執行 `git stash pop` 來重新應用儲藏，同時立刻將其從堆疊中移走。

取消儲藏 (Un-applying a Stash)

在某些使用情境下，你可能想要應用儲藏的變更，做一些工作，然後又要把來自原儲藏的變更取消。Git 並未提供類似 `stash unapply` 的命令，但是達成相同效果是可能的，只要取得該儲藏關連的補丁然後反向應用它就行了：

```
$ git stash show -p stash@{0} | git apply -R
```

同樣的，如果你沒有指定某個儲藏，Git 會預設為最近的儲藏：

```
$ git stash show -p | git apply -R
```

你可能會想要新建一個別名，在你的 `git` 增加一個 `stash-unapply` 命令，這樣更有效率。例如：

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... work work work
$ git stash-unapply
```

從儲藏中創建分支

如果你儲藏了一些工作，暫時不去理會，然後繼續在你儲藏工作的分支上工作，你在重新應用工作時可能會碰到一些問題。如果嘗試應用的變更是針對一個你那之後修改過的檔，你會碰到一個合併衝突並且必須去化解它。如果你想用更方便的方法來重新檢驗你儲藏的變更，你可以執行 `git stash branch`，這會創建一個新的分支，檢出你儲藏工作時所處的提交，重新應用你的工作，如果成功，將會丟棄儲藏。

```
$ git stash branch testchanges
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)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

這是一個很棒的捷徑來恢復儲藏的工作然後在新的分支上繼續當時的工作。

重寫歷史

很多時候，在 Git 上工作的時候，你也許會由於某種原因想要修訂你的提交歷史。Git 的一個卓越之處就是它允許你在最後可能的時刻再作決定。你可以在你即將提交暫存區時決定什麼檔歸入哪一次提交，你可以使用 `stash` 命令來決定你暫時擱置的工作，你可以重寫已經發生的提交以使它們看起來是另外一種樣子。這個包括改變提交的次序、改變說明或者修改提交中包含的檔，將提交歸併(squash)、拆分或者完全刪除——這一切在你尚未開始將你的工作和別人共用前都是可以的。

在這一節中，你會學到如何完成這些很有用的任務，使得你的提交歷史在你將其共用給別人之前變成你想要的樣子。

改變最後一次提交

改變最後一次提交也許是最常見的重寫歷史的行爲。對於你的最近一次提交，你經常想做兩件基本事情：改變提交說明，或者經由增加、改變、移除檔案而改變你剛記錄的快照。

如果你只想修改最近一次提交說明，這非常簡單：

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

這會把你帶入文字編輯器，裡面包含了你最近一次提交的說明訊息，供你修改。當你保存並退出編輯器，這個編輯器會寫入一個新的提交，裡面包含了那個說明，並且讓它成為你的新的最後提交。

如果你完成提交後又想修改被提交的快照，增加或者修改其中的檔案，可能因為你最初提交時，忘了添加一個新建的檔，這個過程基本上一樣。你通過修改檔案然後對其執行 `git add` 或對一個已被記錄的檔執行 `git rm`，隨後的 `git commit --amend` 會獲取你當前的暫存區並將它作為新提交對應的快照。

使用這項技術的時候你必須小心，因為修正會改變提交的SHA-1值。這個很像是一次非常小的 `rebase`——不要在你最近一次提交被推送後還去修正它。

修改多個提交訊息

要修改歷史中更早的提交，你必須採用更複雜的工具。Git 沒有一個修改歷史的工具，但是你可以使用 `rebase` 工具來衍合一系列的提交到它們原來所在的 `HEAD` 上而不是移到新的上。依靠這個互動式的 `rebase` 工具，你就可以停留在每一次提交後，如果你想修改或改變說明、

增加檔案或做任何事情。在 `git rebase` 增加 `-i` 選項可以對話模式執行 `rebase`。你必須告訴 `rebase` 命令要衍合到哪次提交，來指明你想要重寫的提交要回溯到多遠。

例如，你想修改最近三次的提交說明，或者其中任意一次，你必須給 `git rebase -i` 提供一個參數，指明你想要修改的提交的父提交，例如 `HEAD~2^` 或者 `HEAD~3`。可能記住 `-3` 更加容易，因為你想修改最近三次提交；但是請記住你事實上所指的是四次提交之前，即你想修改的提交的父提交。

```
$ git rebase -i HEAD~3
```

再次提醒這是一個衍合命令——`HEAD~3..HEAD` 範圍內的每一次提交都會被重寫，無論你是否修改說明。不要涵蓋你已經推送到中心伺服器的提交——這麼做會使其他開發者產生混亂，因為你提供了同樣變更的不同版本。

執行這個命令會在你的文字編輯器提供一個提交列表，看起來像下面這樣：

```
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
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

很重要的一點是你得注意這些提交的順序與你通常通過 `log` 命令看到的是相反的。如果你執行 `log`，你會看到下面這樣的結果：

```
$ 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
```

請注意這裡的順序是相反的。互動式的 `rebase` 給了你一個即將執行的腳本。它會從你在命令列上指明的提交開始(`HEAD~3`)然後自上至下重播每次提交裡引入的變更。它將最早的列在頂上而不是最近的，因為這是第一個需要重播的。

你需要修改這個腳本來讓它停留在你想修改的變更上。要做到這一點，你只要將你想修改的每一次提交前面的 `pick` 改為 `edit`。例如，只想修改第三次提交說明的話，你就像下面這樣修改文件：

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

當你存檔並退出編輯器，Git 會倒回至列表中的最後一次提交，然後把你送到命令列中，同時顯示以下資訊：

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

這些指示很明確地告訴了你該幹什麼。輸入

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

修改提交說明，退出編輯器。然後，執行

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

這個命令會自動應用其他兩次提交，你就完成任務了。如果你將更多行的 `pick` 改為 `edit`，你就能對你想修改的提交重複這些步驟。Git 每次都會停下，讓你修正提交，完成後繼續執行。

重排(Reordering) 提交

你也可以使用互動式的衍合來徹底重排或刪除提交。如果你想刪除 "added cat-file" 這個提交並且修改其他兩次提交引入的順序，你將 `rebase` 腳本從這個

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

改為這個：

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

當你存檔並退出編輯器，Git 將分支倒回至這些提交的父提交，應用 `310154e`，然後 `f7f3f6d`，接著停止。你有效地修改了這些提交的順序並且徹底刪除了 "added cat-file" 這次提交。

擠壓(Squashing) 提交

互動式的衍合工具還可以將一系列提交擠壓為單一提交。腳本在 `rebase` 的資訊裡放了一些有用的指示：

```
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

如果不用 "pick" 或者 "edit"，而是指定 "squash"，Git 會同時應用那個變更和它之前的變更並將提交說明歸併。因此，如果你想將這三個提交合併為單一提交，你可以將腳本修改成這樣：

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

當你儲存並退出編輯器，Git 會應用全部三次變更然後將你送回編輯器來歸併三次提交說明。

```
# 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
```

當你儲存之後，你就擁有了一个包含前三次提交的全部變更的單一提交。

拆分(Splitting) 提交

拆分提交就是撤銷一次提交，然後多次部分地暫存或提交直到結束。例如，假設你想將三次提交中的中間一次拆分。將「updated README formatting and added blame」拆分成兩次提交：第一次為「updated README formatting」，第二次為「added blame」。你可以在 `rebase -i` 腳本中修改你想拆分的提交前的指令為 "edit"：

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

然後，這個腳本就將你帶入命令列，你重置那次提交，提取被重置的變更，從中創建多次提交。當你儲存並退出編輯器，Git 倒回到列表中第一次提交的父提交，應用第一次提交（`f7f3f6d`），應用第二次提交（`310154e`），然後將你帶到控制台。那裡你可以用 `git reset HEAD^` 對那次提交進行一次混合的重置，這將撤銷那次提交並且將修改的檔從暫存區撤回。此時你可以暫存並提交檔案，直到你擁有多次提交，結束後，執行 `git rebase --continue`。

```
$ 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 在腳本中應用了最後一次提交（`a5f4a0d`），你的歷史看起來就像這樣了：

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

再次提醒，這會修改你列表中的提交的 SHA 值，所以請確保這個列表裡不包含你已經推送到共用倉庫的提交。

核彈級選項: filter-branch

如果你想用腳本的方式修改大量的提交，還有一個重寫歷史的選項可以用——例如，全域性地修改電子郵件地址或者將一個檔從所有提交中刪除。這個命令是 `filter-branch`，這會大面積地修改你的歷史，所以你很有可能不該去用它，除非你的專案尚未公開，沒有其他人在你準備修改的提交的基礎上工作。儘管如此，這個可以非常有用。你會學習一些常見用法，借此對它的能力有所認識。

從所有提交中刪除一個檔

這個經常發生。有些人不經思考使用 `git add .`，意外地提交了一個巨大的二進位檔案，你想將它從所有地方刪除。也許你不小心提交了一個包含密碼的檔，而你想讓你的專案成為 `open source`。`filter-branch` 大概會是你用來清理整個歷史的工具。要從整個歷史中刪除一個名叫 `password.txt` 的檔，你可以在 `filter-branch` 上使用 `--tree-filter` 選項：

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

`--tree-filter` 選項會在每次 `checkout` 專案時先執行指定的命令然後重新提交結果。在這個例子中，你會在所有快照中刪除一個名叫 `password.txt` 的檔，無論它是否存在。如果你想刪除所有不小心提交上去的編輯器備份檔案，你可以執行類似 `git filter-branch --tree-filter "find * -type f -name '*~' -delete" HEAD` 的命令。

你可以觀察到 Git 重寫目錄樹並且提交，然後將分支指標移到末尾。一個比較好的辦法是在一個測試分支上做這件事，然後在你確定結果真的是你所要的之後，再 `hard-reset` 你的主分支。要在你所有的分支上運行 `filter-branch` 的話，你可以傳遞一個 `--all` 參數給該命令。

將一個子目錄設置為新的根目錄

假設你完成了從另外一個代碼控制系統的導入工作，得到了一些沒有意義的子目錄（`trunk`, `tags` 等等）。如果你想讓 `trunk` 子目錄成為每一次提交的新的專案根目錄，`filter-branch` 也可以幫你做到：

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cd8e8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

現在你的專案根目錄就是 `trunk` 子目錄了。Git 會自動地刪除不對這個子目錄產生影響的提交。

全域性地更換電子郵件地址

另一個常見的案例是你在開始時忘了執行 `git config` 來設置你的姓名和電子郵件地址，也許你想開源一個專案，把你所有的工作電子郵件地址修改為個人位址。無論哪種情況你都可以用 `filter-branch` 來更換多次提交裡的電子郵件地址。你必須小心一些，只改變屬於你的電子郵件地址，所以你使用 `--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
```

這樣會巡迴並重寫所有提交使之擁有你的新地址。因為提交裡包含了它們的父提交的 **SHA-1** 值，這個命令會修改你的歷史中的所有提交，而不僅僅是包含了匹配的電子郵件地址的那些。

使用 Git 做 Debug

Git 也提供了一些工具來幫助你 debug 專案中遇到的問題。由於 Git 被設計為可應用於幾乎任何類型的專案，這些工具是通用型的，但是在遇到問題時經常可以幫助你找到 bug 在哪裏。

檔案標注 (File Annotation)

如果你在追查程式碼中的 bug，想要知道這是什麼時候、為什麼被引進來的，檔案標注會是你最佳的工具。它會顯示檔案中對每一行進行修改的最近一次提交。因此，如果你發現自己程式碼中的一個 method 有 bug，你可以用 `git blame` 來標注該檔案，查看那個 method 的每一行分別是由誰在哪一天修改的。下面這個例子使用了 `-L` 選項來限制輸出範圍在第12至22行：

```
$ 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
```

請注意第一欄是最後一次修改該行的那次提交的 SHA-1 部份值。接下去的兩欄是從那次提交中取出的值——作者姓名和日期——所以你可以方便地獲知誰在什麼時候修改了這一行。在這後面是行號和檔案內容。請注意 `^4832fe2` 提交的那些行，這些指的是檔案最初提交 (original commit) 的那些行。那個提交是檔案第一次被加入這個專案時存在的，自那以後未被修改過。這會帶來小小的困惑，因為你已經至少看到了 Git 使用 `^` 來修飾一個提交的 SHA 值的三種不同的意義，但這裡確實就是這個意思。

另一件很酷的事情是，Git 並不會明確地記錄對檔案所做的重命名(rename)動作。它會記錄快照，然後根據實際狀況嘗試找出隱藏在背後的重命名動作。這其中有一個很有意思的特性，就是你可以讓它找出所有的程式碼移動。如果你在 `git blame` 後加上 `-c`，Git 會分析你所標注的檔案，然後嘗試找出其中代碼片段的原始出處，如果它是從其他地方拷貝過來的話。最近，我在對 `GITServerHandler.m` 這個檔案做程式碼重構(code refactoring)，將它分解為多個檔案，其中一個是 `GITPackUpload.m`。通過對 `GITPackUpload.m` 執行帶 `-c` 參數的 `blame` 命令，我可以看到程式碼片段的原始出處：

```
$ 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 COMM
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 COMM
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)
```

這真的非常有用。通常，你會把你拷貝代碼的那次提交作為原始提交，因為這是你在這個檔案中第一次接觸到那幾行。Git可以告訴你編寫那些行的原始提交，即便是在另一個檔案裡。

二分法查找 (Binary Search)

當你知道問題在哪裡的時候，標注檔案會有幫助。如果你不知道，並且自從上次程式碼可用的狀態之後已經經歷了上百次的提交，你可能就要求助於 `git bisect` 命令了。`bisect` 會在你的提交歷史中進行二分查找，來儘快地確定哪一次提交引入了錯誤。

例如你剛剛推送了一個代碼發佈版本到產品環境中，得到一些在你開發環境中沒有發生的錯誤報告，而你對代碼為什麼會表現成那樣百思不得其解。你回到你的代碼中，還好你可以重現那個錯誤，但是找不到問題在哪裡。你可以對代碼執行 `bisect` 來尋找。首先你執行 `git bisect start` 啓動，然後你用 `git bisect bad` 來告訴系統當前的提交已經有問題了。然後你必須告訴 `bisect` 已知的最後一次正常狀態是哪次提交，使用 `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 發現在你標記為正常的提交(v1.0)和當前的錯誤版本之間有大約12次提交，於是它 `checkout` 中間的一個。在這裡，你可以進行測試，檢查問題是否存在於這次提交。如果是，那麼它是在這個中間提交之前的某一次引入的；如果否，那麼問題是在中間提交之後引入的。假設這裡是沒有錯誤的，那麼你就通過 `git bisect good` 來告訴 Git 然後繼續你的旅程：

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

現在你在另外一個提交上了，在你剛剛測試通過的和一個錯誤提交的中點處。你再次執行測試然後發現這次提交是錯誤的，因此你通過 `git bisect bad` 來告訴 Git：

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

這次提交是好的，那麼 Git 就獲得了確定問題引入位置所需的所有資訊。它告訴你第一個錯誤提交的 SHA-1 值，並且顯示一些提交說明，以及哪些檔在那次提交裡被修改過，這樣你可以找出 bug 被引入的根源：

```
$ 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
```

當你完成之後，你應該執行 `git bisect reset` 來重設你的 HEAD 到你開始前的地方，否則你會處於一個詭異的狀態：

```
$ git bisect reset
```

這是個強大的工具，可以幫助你檢查上百的提交，在幾分鐘內找出 bug 引入的位置。事實上，如果你有一個腳本程式會在專案工作正常時返回0，錯誤時返回非0的話，你可以完全自動地執行 `git bisect`。首先你需要提供已知的錯誤和正確提交來告訴它二分查找的範圍。你可以通過 `bisect start` 命令來列出它們，先列出已知的錯誤提交再列出已知的正確提交：

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

這樣會自動地在每一個 checked-out 提交裡執行 `test-error.sh` 直到 Git 找出第一個破損的提交。你也可以執行像 `make` 或者 `make tests`，或者任何你所能執行的自動化測試。

子模組 (Submodules)

經常有這樣的事情，當你在一個專案上工作時，你需要在其中使用另外一個專案。也許它是一個協力廠商開發的程式庫(Library)，或者是你另外開發給多個父專案使用的子專案。在這個情境下產生了一個常見的問題：你想將這兩個專案分開處理，但是又需要在其中一個中使用另外一個。

這裡有一個例子。假設你在開發一個網站，並提供 Atom 訂閱(Atom feeds)。你不想自己編寫產生 Atom 的程式，而是決定使用一個 Library。你可能必須從 CPAN install 或者 Ruby gem 之類的共用庫(shared library)將那段程式 include 進來，或者將原始程式碼複製到你的專案樹中。如果採用包含程式庫的辦法，那麼不管用什麼辦法都很難對這個程式庫做客製化(customize)，部署它就更加困難了，因為你必須確保每個客戶都擁有那個程式庫。把程式碼包含到你自己的專案中帶來的問題是，當上游被修改時，任何你進行的客製化的修改都很難歸併(merge)。

Git 通過子模組處理這個問題。子模組允許你將一個 Git 倉庫當作另外一個 Git 倉庫的子目錄。這允許你 clone 另外一個倉庫到你的專案中並且保持你的提交相對獨立。

子模組初步

假設你想把 Rack library (一個 Ruby 的 web 伺服器閘道介面) 加入到你的專案中，可能既要保持你自己的變更，又要延續上游的變更。首先你要把外部的倉庫 clone 到你的子目錄中。

你通過 `git submodule add` 命令將外部專案加為子模組：

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

現在你就在專案裡的 `rack` 子目錄下有了一個 Rack 專案。你可以進入那個子目錄，進行變更，加入你自己的遠端可寫倉庫來推送你的變更，從原始倉庫拉取(pull)和歸併等等。如果你在加入子模組後立刻運行 `git status`，你會看到下面兩項：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitmodules
#       new file:   rack
#
```

首先你注意到有一個 `.gitmodules` 文件。這是一個設定檔，保存了專案 URL 和你拉取到的本地子目錄

```
$ cat .gitmodules
[submodule "rack"]
  path = rack
  url = git://github.com/chneukirchen/rack.git
```

如果你有多個子模組，這個檔裡會有多個條目。很重要的一點是這個文件跟其他文件一樣也是處於版本控制之下的，就像你的 `.gitignore` 檔一樣。它跟專案裡的其他檔一樣可以被推送和拉取。這是其他 `clone` 此專案的人獲知子模組專案來源的途徑。

`git status` 的輸出裡所列的另一項 `rack`。如果你在它上面執行 `git diff`，會發現一些有趣的東西：

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbebe7821e37fa53e69afc433
```

儘管 `rack` 是你工作目錄裡的子目錄，但 Git 把它視作一個子模組，當你不在那個目錄裡的時候，Git 並不會追蹤記錄它的內容。取而代之的是，Git 將它記錄成來自那個倉庫的一個特殊的提交。當你在那個子目錄裡修改並提交時，子專案會通知那裡的 HEAD 已經發生變更並記錄你當前正在工作的那個提交；通過那樣的方法，當其他人 `clone` 此專案，他們可以重新創建一致的環境。

這是關於子模組的重要一點：你記錄他們當前確切所處的提交。你不能記錄一個子模組的 `master` 或者其他的符號引用(sympolic reference)。

當你提交時，會看到類似如下：

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

注意 `rack` 條目的 `160000` 模式。這在 Git 中是一個特殊模式，基本意思是將一個提交記錄為一個目錄項而不是子目錄或者檔案。

你可以將 `rack` 目錄當作一個獨立的專案，保持一個指向子目錄的最新提交的指標然後反復地更新上層專案。所有的 Git 命令都在兩個子目錄裡獨立工作：

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:03:56 2009 -0700

  first commit with submodule rack
$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbebe7821e37fa53e69afc433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date:   Wed Mar 25 14:49:04 2009 +0100

  Document version change
```

Clone 一個帶子模組的專案

這裡你將 `clone` 一個帶子模組的專案。當你接收到這樣一個專案，你將得到了包含子專案的目錄，但裡面沒有檔案：

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r-- 1 schacon admin 3 Apr 9 09:11 README
drwxr-xr-x 2 schacon admin 68 Apr 9 09:11 rack
$ ls rack/
$
```

`rack` 目錄存在了，但是是空的。你必須執行兩個命令：`git submodule init` 來初始化你的本地設定檔，`git submodule update` 來從那個專案拉取所有資料並 `check out` 你上層專案裡所列的合適的提交：

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbebe7821e37fa53e69afcf433'
```

現在你的 `rack` 子目錄就處於你先前提交的確切狀態了。如果另外一個開發者變更了 `rack` 的代碼並提交，你拉取那個引用然後歸併之，你會得到有點怪怪的東西：

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +-.
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# 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:   rack
#
```

你歸併進來的僅僅是一個指向你的子模組的指標；但是它並不更新你子模組目錄裡的代碼，所以看起來你的工作目錄處於一個臨時狀態(*dirty state*)：

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1 +1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbebe7821e37fa53e69afcf433
```

事情就是這樣，因為你所擁有的子模組的指標，並沒有對應到子模組目錄的真實狀態。為了修復這一點，你必須再次運行 `git submodule update`：

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
  08d709f..6c5e70b  master      -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

每次你從主專案中拉取一個子模組的變更都必須這樣做。看起來很怪但是管用。

一個常見問題是當開發者對子模組做了一個本地的變更但是並沒有推送到公共伺服器。然後他們提交了一個指向那個非公開狀態的指標然後推送上層專案。當其他開發者試圖運行 `git submodule update`，那個子模組系統會找不到所引用的提交，因為它只存在於第一個開發者的系統中。如果發生那種情況，你會看到類似這樣的錯誤：

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

你不得不去查看誰最後變更子模組：

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

然後，你給那個傢伙發電子郵件說他一通。

上層專案

有時候，開發者想按照他們的分組獲取一個大專案的子目錄的子集。如果你是從 CVS 或者 Subversion 遷移過來的話這個很常見，在那些系統中你已經定義了一個模組或者子目錄的集合，而你想延續這種類型的工作流程。

在 Git 中實現這個的一個好辦法是你將每一個子目錄都做成獨立的 Git 倉庫，然後創建一個上層專案的 Git 倉庫包含多個子模組。這個辦法的一個優勢是，你可以在上層專案中通過標籤和分支更為明確地定義專案之間的關係。

子模組的問題

使用子模組並非沒有任何缺點。首先，你在子模組目錄中工作時必須相對小心。當你執行 `git submodule update`，它會 `check out` 專案的指定版本，但是不在分支內。這叫做獲得一個分離的頭(*detached head*)——這意味著 HEAD 檔直接指向一次提交，而不是一個符號引用(*symbolic reference*)。問題在於你通常並不想在一個分離的頭的環境下工作，因為太容易丟失變更了。如果你先執行了一次 `submodule update`，然後在那個子模組目錄裡不創建分支就進行提交，然後再次從上層專案裡執行 `git submodule update` 同時不進行提交，Git 會毫無提示地覆蓋你的變更。技術上講你不會丟失工作，但是你將失去指向它的分支，因此會很難取得。

為了避免這個問題，當你在子模組目錄裡工作時應使用 `git checkout -b work` 或類似的命令來創建一個分支。當你再次在子模組裡更新的時候，它仍然會覆蓋你的工作，但是至少你擁有一個可以回溯的指標。

切換帶有子模組的分支同樣也很有技巧。如果你創建一個新的分支，增加了一個子模組，然後切換回不帶該子模組的分支，你仍然會擁有一個未被追蹤的子模組的目錄：

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       rack/
```

你將不得不將它移走或者刪除，這樣的話當你切換回去的時候必須重新 `clone` 它——你可能會丟失你未推送的本地的變更或分支。

最後一個需要注意的是關於從子目錄切換到子模組。如果你已經追蹤了你專案中的一些檔案，但是想把它們移到子模組去，你必須非常小心，否則 Git 會生你的氣。假設你的專案中有一個子目錄裡放了 `rack` 的檔，然後你想將它轉換為子模組。如果你刪除子目錄然後執行 `submodule add`，Git 會向你大吼：

```
$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

你必須先將 `rack` 目錄撤回(unstage)。然後你才能加入子模組：

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
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 | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

現在假設你在一個分支裡那樣做了。如果你嘗試切換回一個仍然在目錄裡保留那些檔而不是子模組的分支時——你會得到下面的錯誤：

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

你必須先移除 `rack` 子模組的目錄才能切換到不包含它的分支：

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README      rack
```

然後，當你切換回來，你會得到一個空的 `rack` 目錄。你可以執行 `git submodule update` 重新 clone，也可以將 `/tmp/rack` 目錄重新移回空目錄。

子樹合併

現在你已經看到了子模組系統的麻煩之處，讓我們來看一下解決相同問題的另一途徑。當 Git 歸併(merge)時，它會檢查需要歸併的內容然後選擇一個合適的歸併策略。如果你歸併的分支是兩個，Git 使用一個「遞迴(recursive)」策略。如果你歸併的分支超過兩個，Git 採用「章魚」策略。這些策略是自動選擇的，因為遞迴策略可以處理複雜的三路歸併情況——比如多於一個共同祖先——但是它只能處理兩個分支的歸併。章魚歸併可以處理多個分支，但是必須更加小心以避免衝突帶來的麻煩，因此它被選中作為歸併兩個以上分支的預設策略。

實際上，你也可以選擇其他策略。其中的一個就是「子樹歸併(subtree merge)」，你可以用它來處理子專案問題。這裡你會看到如何換用子樹歸併的方法來實現前一節裡所做的 rack 的嵌入。

子樹歸併的想法是你擁有兩個專案，其中一個專案映射到另外一個專案的子目錄中，反過來也一樣。當你指定一個子樹歸併，Git 可以聰明地探知其中一個是另外一個的子樹從而實現正確的歸併——這相當神奇。

首先你將 Rack 應用加入到專案中。你將 Rack 專案當作你專案中的一個遠端參照，然後將它 check out 到它自身的分支：

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
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 git@github.com:schacon/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"
```

現在在你的 `rack_branch` 分支中就有了 Rack 專案的根目錄，而你自己的專案在 `master` 分支中。如果你先 check out 其中一個然後另外一個，你會看到它們有不同的專案根目錄：

```
$ ls
AUTHORS      KNOWN-ISSUES   Rakefile     contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

要將 Rack 專案當作子目錄拉取到你的 master 專案中。你可以在 Git 中用 `git read-tree` 來實現。你會在第9章學到更多與 `read-tree` 和它的朋友相關的東西，目前你會知道它讀取一個分支的根目錄樹到當前的暫存區和工作目錄。你只要切換回你的 `master` 分支，然後拉取 `rack` 分支到你主專案的 `master` 分支的 `rack` 子目錄：

```
$ git read-tree --prefix=rack/ -u rack_branch
```

當你提交的時候，看起來就像你在那個子目錄下擁有全部 Rack 的檔案——就像你從一個 `tarball` 裡拷貝的一樣。有意思的是你可以比較容易地歸併其中一個分支的變更到另外一個。因此，如果 Rack 專案更新了，你可以通過切換到那個分支並執行拉取來獲得上游的變更：

```
$ git checkout rack_branch
$ git pull
```

然後，你可以將那些變更歸併回你的 `master` 分支。你可以使用 `git merge -s subtree`，它會工作的很好；但是 Git 同時會把歷史歸併到一起，這可能不是你想要的。為了拉取變更並預置提交說明，需要在 `-s subtree` 策略選項的同時使用 `--squash` 和 `--no-commit` 選項。

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

所有 Rack 專案的變更都被歸併進來，而且可以進行本地提交。你也可以做相反的事情——在你主分支的 `rack` 目錄裡進行變更然後歸併回 `rack_branch` 分支，然後將它們提交給維護者或者推送到上游。

為了得到 `rack` 子目錄和你 `rack_branch` 分支的區別——以決定你是否需要歸併它們——你不能使用一般的 `diff` 命令。而是對你想比較的分支執行 `git diff-tree`：

```
$ git diff-tree -p rack_branch
```

或者，為了比較你的 `rack` 子目錄和伺服器上你拉取時的 `master` 分支，你可以執行

```
$ git diff-tree -p rack_remote/master
```

總結

你已經看到了很多高級的工具，允許你更加精確地操控你的提交和暫存區(staging area)。當你碰到問題時，你應該可以很容易找出是哪個分支、什麼時候、由誰引入了它們。如果你想在專案中使用子專案，你也已經學會了一些方法來滿足這些需求。到此，你應該能夠在 Git 命令列下完成大部分的日常事務，並且感到比較順手。

Git 客製化

到目前為止，我闡述了 Git 基本的運作機制和使用方式，介紹了 Git 提供的許多工具來幫助你簡單且有效地使用它。在本章，我將會介紹 Git 的一些重要的組態設定(**configuration**)和鉤子(**hooks**)機制以滿足自訂的要求。通過這些工具，它能夠更容易地使 Git 按照你、你的公司或團隊所需要的方式去運作。

Git 配置

如第一章所言，用 `git config` 配置 Git，要做的第一件事就是設置名字和郵箱地址：

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

從現在開始，你會瞭解到一些類似以上但更為有趣的設置選項來自訂 Git。

先過一遍第一章中提到的 Git 配置細節。Git 使用一系列的設定檔來儲存你定義的偏好，它首先會查找 `/etc/gitconfig` 檔，該檔所含的設定值對系統上所有使用者都有效，也對他們所擁有的倉庫都有效（譯注：`gitconfig` 是全域設定檔），如果傳遞 `--system` 選項給 `git config` 命令，Git 會讀寫這個檔。

接下來 Git 會尋找每個用戶的 `~/.gitconfig` 檔，它是針對個別使用者的，你可以傳遞 `--global` 選項讓 Git 讀寫該檔。

最後，Git 會尋找你目前使用中的倉庫 Git 目錄下的設定檔（`.git/config`），該文件中的設定值只對這個倉庫有效。以上闡述的三層配置從一般到特殊層層推進，如果定義的值有衝突，後面層級的設定會覆寫前面層級的設定值，例如：在 `.git/config` 和 `/etc/gitconfig` 的較量中，`.git/config` 取得了勝利。雖然你也可以直接手動編輯這些設定檔，但是執行 `git config` 命令將會來得簡單些。

用戶端基本配置

Git 能夠識別的配置項被分爲了兩大類：用戶端和伺服器端，其中大部分屬於用戶端配置，這是基於你個人工作偏好所做的設定。儘管有數不盡的選項，但我只闡述其中經常使用、或者會對你的工作流程產生巨大影響的選項。許多選項只在極端的情況下有用，這裏就不多做介紹了。如果你想觀察你的 Git 版本能識別的選項清單，請執行

```
$ git config --help
```

`git config` 的手冊頁（譯注：以 `man` 命令的顯示方式）非常細緻地羅列了所有可用的配置項。

core.editor

預設情況下，Git 會使用你所設定的「預設文字編輯器」，否則會使用 Vi 來創建和編輯提交以及標籤資訊，你可以使用 `core.editor` 改變預設編輯器：

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

現在無論你的環境變數 editor 被定義成什麼，Git 都會觸發 Emacs 來編輯相關訊息。

commit.template

如果把這個項目指定為你系統上的一個檔，當你提交的時候，Git 會預設使用該檔定義的內容做為預設的提交訊息。例如：你創建了一個範本檔 `$HOME/.gitmessage.txt`，它看起來像這樣：

```
subject line

what happened

[ticket: X]
```

如下設定 `commit.template` 可以告訴 Git，把上列內容做為預設訊息，當你執行 `git commit` 的時候，在你的編輯器中顯示：

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

然後當你提交時，在編輯器中顯示的提交資訊如下：

```
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
```

如果你對於提交訊息有特定的政策，那就在系統上創建一個範本檔，設定 Git 使用它做為預設值，這樣可以幫助提升你的政策經常被遵守的機會。

core.pager

`core.pager` 指定 Git 執行諸如 `log`、`diff` 等命令時所使用的分頁器，你可以設成用 `more` 或者任何你喜歡的分頁器（預設用的是 `less`），當然你也可以不用分頁器，只要把它設成空字串：

```
$ git config --global core.pager ''
```

這樣不管命令的輸出量多少，都會在一頁顯示所有內容。

user.signingkey

如果你要創建經簽署的含附注的標籤(signed annotated tags)（正如第二章所述），那麼把你的 GPG 簽署金鑰設為配置項會更好，設置金鑰 ID 如下：

```
$ git config --global user.signingkey <gpg-key-id>
```

現在你能夠簽署標籤，從而不必每次執行 `git tag` 命令時定義金鑰：

```
$ git tag -s <tag-name>
```

core.excludesfile

正如第二章所述，你能在專案倉庫的 `.gitignore` 檔裡頭用模式(pattern)來定義那些無需納入 Git 管理的檔案，這樣它們不會出現在未追蹤列表，也不會在你執行 `git add` 後被暫存。然而，如果你想用專案倉庫之外的檔案來定義那些需被忽略的檔的話，可以用 `core.excludesfile` 來通知 Git 該檔所在的位置，檔案內容則和 `.gitignore` 類似。

help.autocorrect

這個選置項只在 Git 1.6.1 以上(含)版本有效，假如你在 Git 1.6 中錯打了一條命令，它會像這樣顯示：

```
$ git com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
    commit
```

如果你把 `help.autocorrect` 設置成1（譯注：啟動自動修正），那麼在只有一個命令符合的情況下，Git 會自動執行該命令。

Git 中的著色

Git 能夠為輸出到你終端(terminal)的內容著色，以便你可以憑直觀進行快速、簡單地分析，有許多選項能幫你將顏色調成你喜好的。

color.ui

Git 會按照你的需要，自動為大部分的輸出加上顏色。你能明確地規定哪些需要著色、以及怎樣著色，設置 `color.ui` 為 `true` 來打開所有的預設終端著色：

```
$ git config --global color.ui true
```

設置好以後，當輸出到終端時，Git 會為之加上顏色。其他的參數還有 `false` 和 `always`，`false` 意味著不為輸出著色，而 `always` 則表示在任何情況下都要著色 -- 即使 Git 命令被重定向到文件或 pipe 到另一個命令。Git 1.5.5 版本引進了此項配置，如果你的版本更舊，你必須對顏色有關選項各自進行詳細地設置。

你會很少用到 `color.ui = always`，在大多數情況下，如果你想在被重定向的輸出中插入顏色碼，你可以傳遞 `--color` 旗標給 Git 命令來迫使它這麼做，`color.ui = true` 應該是你的首選。

color.*

想要具體指定哪些命令輸出需要被著色，以及怎樣著色，或者 Git 的版本很舊，你就需要用到和具體命令有關的顏色配置選項，它們都能被設為 `true`、`false` 或 `always`：

```
color.branch  
color.diff  
color.interactive  
color.status
```

除此之外，以上每個選項都有子選項，可以被用來覆蓋其父設置，以達到為輸出的各個部分著色的目的。例如，要讓 `diff` 輸出的改變資訊 (meta information) 以粗體、藍色前景和黑色背景的形式顯示，你可以執行：

```
$ git config --global color.diff.meta "blue black bold"
```

你能設置的顏色值如下：`normal`、`black`、`red`、`green`、`yellow`、`blue`、`magenta`、`cyan`、`white`。正如以上例子設置的粗體屬性，想要設置字體屬性的話，你的選擇有：`bold`、`dim`、`ul`、`blink`、`reverse`。

如果你想配置子選項的話，可以參考 `git config` 幫助頁。

外部的合併與比較工具

雖然 Git 自己實做了 `diff`，而且到目前為止你一直在使用它，但你能夠設定一個外部的工具來替代它。你還可以設定用一個圖形化的工具來合併和解決衝突，而不必自己手動解決。有一個不錯且免費的工具可以被用來做比較和合併工作，它就是 P4Merge（譯注：Perforce 圖形化合併工具），我會展示它的安裝過程。

所以如果你想試試看的話，因為 P4Merge 可以在所有主流平臺上運行，所以你應該可以嘗試看看。對於向你展示的例子，在 Mac 和 Linux 系統上，我會使用路徑名；在 Windows 上，`/usr/local/bin` 應該被改為你環境中的可執行路徑。

你可以在這裏下載 P4Merge：

```
http://www.perforce.com/perforce/downloads/component.html
```

首先，你要設定一個外部包裝腳本(**external wrapper scripts**)來執行你要的命令，我會使用 Mac 系統上的路徑來指定該腳本的位置；在其他系統上，它應該被放置在二進位檔案 `p4merge` 所在的目錄中。創建一個 `merge` 包裝腳本，名字叫作 `extMerge`，讓它附帶所有參數呼叫 `p4merge` 二進位檔案：

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

`diff` 包裝腳本首先確定傳遞過來7個參數，隨後把其中2個傳遞給你的 `merge` 包裝腳本，預設情況下，Git 會傳遞以下參數給 `diff`：

```
path old-file old-hex old-mode new-file new-hex new-mode
```

由於你僅僅需要 `old-file` 和 `new-file` 參數，用 `diff` 包裝腳本來傳遞它們吧。

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

你還需要確認一下這兩個腳本是可執行的：

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

現在來設定使用你自訂的比較和合併工具吧。這需要許多自訂設置：`merge.tool` 通知 Git 使用哪個合併工具；`mergetool.*.cmd` 規定命令如何執行；`mergetool.trustExitCode` 會通知 Git 該程式的退出碼(exit code)是否指示合併操作成功；`diff.external` 通知 Git 用什麼命令做比較。因此，你可以執行以下4條配置命令：

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

或者直接編輯 `~/.gitconfig` 文件如下：

```
[merge]
tool = extMerge
[mergetool "extMerge"]
cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
trustExitCode = false
[diff]
external = extDiff
```

設置完畢後，如果你像這樣執行 `diff` 命令：

```
$ git diff 32d1776b1^ 32d1776b1
```

不同於在命令列得到 `diff` 命令的輸出，Git 觸發了剛剛設置的 P4Merge，它看起來像圖7-1這樣：

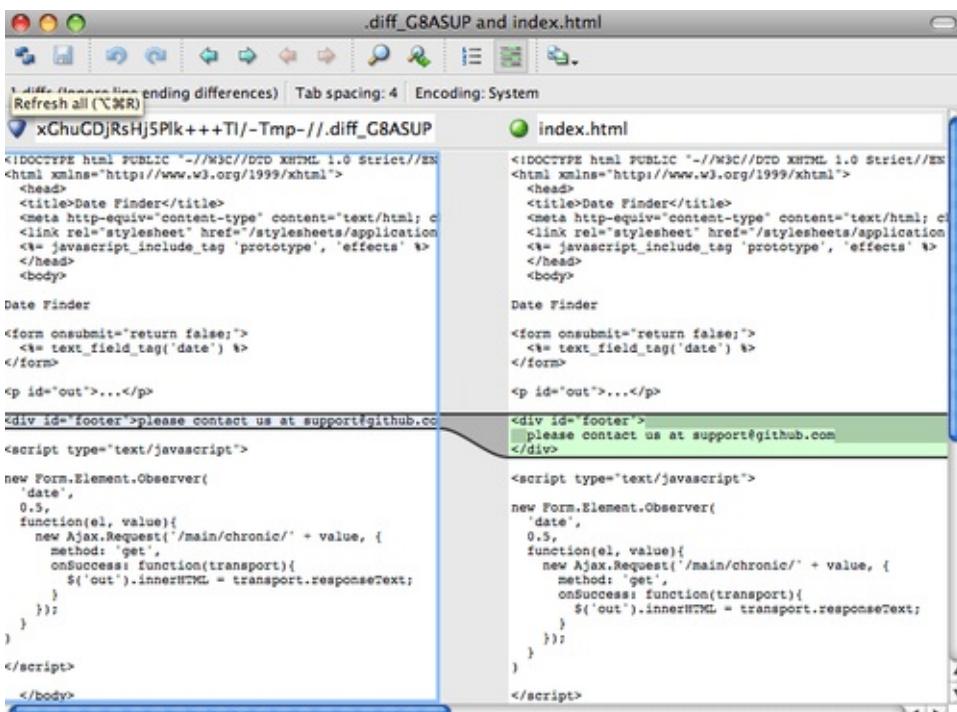


Figure 7-1. P4Merge.

當你設法合併兩個分支，結果卻有衝突時，執行 `git mergetool`，Git 會啓用 P4Merge 讓你通過圖形介面來解決衝突。

設置包裝腳本的好處是你能簡單地改變 `diff` 和 `merge` 工具，例如把 `extDiff` 和 `extMerge` 改成 `KDiff3`，要做的僅僅是編輯 `extMerge` 指令檔：

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

現在 Git 會使用 `KDiff3` 來做比較、合併和解決衝突。

Git 預先設置了許多其他的合併和解決衝突的工具，而你不必設置 `cmd`。可以把合併工具設置為：`kdiff3`、`opendiff`、`tkdiff`、`meld`、`xxdiff`、`emerge`、`vimdiff`、`gvimdiff`。如果你不想用 `KDiff3` 來做 `diff`，只是想用它來合併，而且 `kdiff3` 命令也在你的路徑裏，那麼你可以執行：

```
$ git config --global merge.tool kdiff3
```

如果執行了以上命令，沒有設置 `extMerge` 和 `extDiff` 檔，Git 會用 `KDiff3` 做合併，讓平常內建的比較工具來做比較。

格式化與空格

格式化與空格是許多開發人員在協同工作時，特別是在跨平臺情況下，遇到的令人頭疼的細小問題。在一些大家合作的工作或提交的補丁中，很容易因為編輯器安靜無聲地加入一些小空格，或者 Windows 程式師在跨平臺專案中的檔案行尾加入了回車分行符號(carriage return)。Git 的一些配置選項可以幫助解決這些問題。

core.autocrlf

如果你在 Windows 上寫程式，或者你不是用 Windows，但和其他在 Windows 上寫程式的人合作，在這些情況下，你可能會遇到換行符號的問題。這是因為 Windows 使用回車(carriage-return)和換行(linefeed)兩個字元來結束一行，而 Mac 和 Linux 只使用一個換行字元。雖然這是小問題，但它會極大地擾亂跨平臺協作。

Git 可以在你提交時自動地把換行符號 CRLF 轉換成 LF，而在簽出代碼時把 LF 轉換成 CRLF。用 `core.autocrlf` 來打開此項功能，如果是在 Windows 系統上，把它設置成 `true`，這樣當 `check out` 程式的時候，LF 會被轉換成 CRLF：

```
$ git config --global core.autocrlf true
```

Linux 或 Mac 系統使用 LF 作為行結束符，因此你不希望 Git 在 check out 檔案時進行自動的轉換；但是，當一個以 CRLF 做為換行符號的檔案不小心被引入時，你肯定希望 Git 可以修正它。你可以把 `core.autocrlf` 設置成 `input` 來告訴 Git 在提交時把 CRLF 轉換成 LF，check out 時不轉換：

```
$ git config --global core.autocrlf input
```

這樣會在 Windows 系統上的 check out 檔案中保留 CRLF，而在 Mac 和 Linux 系統上，以及倉庫中保留 LF。

如果你是 Windows 程式師，且正在開發僅運行在 Windows 上的專案，可以設置 `false` 取消此功能，把 carriage returns 記錄在倉庫中：

```
$ git config --global core.autocrlf false
```

core.whitespace

Git 預先設置了一些選項來探測和修正空格問題，其中有四個主要選項，有2個預設開啓，2個預設關閉，你可以自由地打開或關閉它們。

預設開啓的2個選項是：`trailing-space` 會查找每行結尾的空格，`space-before-tab` 會查找每行開頭的定位字元前的空格。

預設關閉的2個選項是：`indent-with-non-tab` 會查找8個以上空格（非定位字元）開頭的行，`cr-at-eol` 告訴 Git carriage returns 是合法的。

設置 `core.whitespace`，按照你的需要來打開或關閉選項，設定值之間以逗號分隔。從設定字符串裏把設定值去掉，就可以關閉這個設定，或是在設定值前面加上減號 `-` 也可以。例如，如果你想要打開除了 `cr-at-eol` 之外的所有選項，你可以這麼做：

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

當你執行 `git diff` 命令且為輸出著色時，Git 會偵測這些問題，因此你有可能在提交前修復它們。當你用 `git apply` 打補丁時，它也同樣會使用這些設定值來幫助你。你可以要 Git 警告你，如果正準備運用的補丁有特別的空白問題：

```
$ git apply --whitespace=warn <patch>
```

或者讓 Git 在打上補丁嘗試自動修正此問題：

```
$ git apply --whitespace=fix <patch>
```

這些選項也能運用於衍合。如果提交了有空格問題的檔但還沒推送到上游，你可以執行帶有 `--whitespace=fix` 選項的 `rebase` 來讓 Git 在重寫補丁時自動修正它們。

伺服器端配置

Git 伺服器端的配置選項並不多，但仍有一些有趣的選項值得你一看。

`receive.fsckObjects`

Git 預設情況下不會在推送期間檢查所有物件的一致性。Git 雖然會檢查確認每個物件仍然符合它的 SHA-1 checksum，所指向的物件也都是有效的，但是預設 Git 不會在每次推送時都做這種檢查。對於 Git 來說，倉庫或推送的檔越大，這個操作代價就相對越高，每次推送會消耗更多時間。如果想讓 Git 在每次推送時都檢查物件一致性，可以設定 `receive.fsckObjects` 為 `true` 來強迫它這麼做：

```
$ git config --system receive.fsckObjects true
```

現在 Git 會在每次推送被接受前檢查庫的完整性，確保有問題的用戶端沒有引入破壞性的資料。

`receive.denyNonFastForwards`

如果對已經被推送的提交歷史做衍合，繼而再推送；或是要將某個提交推送到遠端分支，而該提交歷史未包含這個遠端分支目前指向的 commit，這樣的推送會被拒絕。這通常是個很好的禁止策略，但有時你在做衍合的時候，你可能很確定自己在做什麼，那就可以在 `push` 命令後加 `-f` 旗標來強制更新遠端分支。

要禁用這樣的強制更新遠端分支 non-fast-forward references 的功能，可以如下設定

`receive.denyNonFastForwards`：

```
$ git config --system receive.denyNonFastForwards true
```

稍後你會看到，用伺服器端的 `receive hooks` 也能達到同樣的目的。這個方法可以做更細緻的控制，例如：拒絕某些特定的使用者強制更新 non-fast-forwards。

`receive.denyDeletes`

避開 `denyNonFastForwards` 策略的方法之一就是使用者刪除分支，然後推回新的引用 (reference)。在更新的 Git 版本中（從 1.6.1 版本開始），你可以把 `receive.denyDeletes` 設置為 `true`：

```
$ git config --system receive.denyDeletes true
```

這樣會在推送過程中阻止刪除分支和標籤 — 沒有使用者能夠這麼做。要刪除遠端分支，必須從伺服器手動刪除引用檔(ref files)。通過用戶存取控制清單也能這麼做，在本章結尾將會介紹這些有趣的方式。

Git 屬性

一些設定值(settings)也能指定到特定的路徑，這樣，Git 只對這個特定的子目錄或某些檔案應用這些設定值。這些針對特定路徑的設定值被稱為 Git 屬性(attributes)，可以在你目錄中的 `.gitattributes` 檔內進行設置（通常是你專案的根目錄），當你不想讓這些屬性檔和專案檔案一同提交時，也可以在 `.git/info/attributes` 檔進行設置。

使用屬性，你可以對個別檔案或目錄定義不同的合併策略，讓 Git 知道怎樣比較非文字檔，在你提交或簽出(check out)前讓 Git 過濾內容。你將在這個章節裏瞭解到能在自己的專案中使用的屬性，以及一些實例。

二進位檔案

你可以用 Git 屬性讓其知道哪些是二進位檔案（以防 Git 沒有識別出來），以及指示怎樣處理這些檔，這點很酷。例如，一些文字檔是由機器產生的，而且無法比較，而一些二進位檔案可以比較 — 你將會瞭解到怎樣讓 Git 識別這些檔。

識別二進位檔案

某些檔案看起來像是文字檔，但其實是看做為二進位資料。例如，在 Mac 上的 Xcode 專案含有一個以 `.pbxproj` 結尾的檔，它是由記錄設置項的 IDE 寫到磁碟的 JSON 資料集（純文字 javascript 資料類型）。雖然技術上看它是由 ASCII 字元組成的文字檔，但是你並不想這麼看它，因為它確實是一個輕量級資料庫 — 如果有兩個人改變了它，你沒辦法合併它們，`diff` 通常也幫不上忙，只有機器才能進行識別和操作，於是，你想把它當成二進位檔案。

讓 Git 把所有 `.pbxproj` 檔當成二進位檔案，在 `.gitattributes` 文件中加上下面這行：

```
*.pbxproj -crlf -diff
```

現在，Git 不會嘗試轉換和修正 CRLF (回車換行) 問題；也不會當你在專案中執行 `git show` 或 `git diff` 時，嘗試比較不同的內容。在 Git 1.6 及之後的版本中，可以用一個巨集代替 `-crlf -diff`：

```
*.pbxproj binary
```

Diffing Binary Files

在 Git 1.6 及以上版本中，你能利用 Git 屬性來有效地比較二進位檔案。可以設置 Git 把二進位資料轉換成文本格式，然後用一般 `diff` 來做比較。

MS Word files

這個特性很酷，而且鮮為人知，因此我會結合實例來講解。首先，你將使用這項技術來解決最令人頭疼的問題之一：對 Word 文檔進行版本控制。每個人都知道 Word 是最可怕的編輯器，奇怪的是，每個人都在使用它。如果想對 Word 文件進行版本控制，你可以把檔案加入到 Git 倉庫中，每次修改後提交即可。但這樣做有什麼好處？如果你像平常一樣執行 `git diff` 命令，你只能得到如下的結果：

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

你不能直接比較兩個 Word 文件版本，除非人工細看，對吧？Git 屬性能很好地解決此問題，把下面這行加到 `.gitattributes` 文件：

```
*.doc diff=word
```

當你要看比較結果時，如果檔副檔名是 "doc"，Git 會使用 "word" 篩檢程式(filter)。什麼是 "word" 篩檢程式呢？你必須設置它。下面你將設定 Git 使用 `strings` 程式，把 Word 文檔轉換成可讀的文字檔，之後再進行比較：

```
$ git config diff.word.textconv catdoc
```

This command adds a section to your `.git/config` that looks like this:

```
[diff "word"]
textconv = catdoc
```

現在 Git 知道了，如果它要在兩個快照之間做比較，而其中任何一個檔案名是以 `.doc` 結尾，它應該要對這些檔執行 "word" 篩檢程式，也就是定義為執行 `strings` 程式。這樣就可以在比較前把 Word 檔轉換成文字檔。

下面展示了一個實例，我把此書的第一章納入 Git 管理，在一個段落中加入了一些文字後保存，之後執行 `git diff` 命令，得到結果如下：

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -128,7 +128,7 @@ and data size)
Since its birth in 2005, Git has evolved and matured to be easy to use
and yet retain these initial qualities. It's incredibly fast, it's
very efficient with large projects, and it has an incredible branching
-system for non-linear development.
+system for non-linear development (See Chapter 3).
```

Git 成功且簡潔地顯示出我增加的文字 "(See Chapter 3)"。雖然有些瑕疵 -- 在末尾顯示了一些隨機的內容 -- 但確實可以比較了。如果你能找到或自己寫個 Word 到純文字的轉換器的話，效果可能會更好。不過因為 `strings` 可以在大部分 Mac 和 Linux 系統上運行，所以在初次嘗試對各種二進位格式檔進行類似的處理，它是個不錯的選擇。

OpenDocument Text files

The same approach that we used for MS Word files (`*.doc`) can be used for OpenDocument Text files (`*.odt`) created by OpenOffice.org.

Add the following line to your `.gitattributes` file:

```
*.odt diff=odt
```

Now set up the `odt diff` filter in `.git/config`:

```
[diff "odt"]
binary = true
textconv = /usr/local/bin/odt-to-txt
```

OpenDocument files are actually zip'ped directories containing multiple files (the content in an XML format, stylesheets, images, etc.). We'll need to write a script to extract the content and return it as plain text. Create a file `/usr/local/bin/odt-to-txt` (you are free to put it into a different directory) with the following content:

```

#!/usr/bin/env perl
# Simplistic OpenDocument Text (.odt) to plain text converter.
# Author: Philipp Kempgen

if (! defined($ARGV[0])) {
    print STDERR "No filename given!\n";
    print STDERR "Usage: $0 filename\n";
    exit 1;
}

my $content = '';
open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;
{
    local $/ = undef; # slurp mode
    $content = <$fh>;
}
close $fh;
$_ = $content;
s/<text:span\b[^>]*>/g;           # remove spans
s/<text:h\b[^>]*>/\n\n***** /g;   # headers
s/<text:list-item\b[^>]*>\s*<text:p\b[^>]*>/\n      -- /g; # list items
s/<text:list\b[^>]*>/\n\n/g;        # lists
s/<text:p\b[^>]*>/\n /g;          # paragraphs
s/<[^>]+>/g;                   # remove all XML tags
s/\n{2,}/\n\n/g;                 # remove multiple blank lines
s/^A\n+//;                      # remove leading blank lines
print "\n", $_, "\n\n";

```

And make it executable

```
chmod +x /usr/local/bin/odt-to-txt
```

Now `git diff` will be able to tell you what changed in `.odt` files.

Image files

你還能用這個方法解決另一個有趣的問題：比較影像檔。方法之一是對 JPEG 檔執行一個篩檢程式，把 EXIF 資訊提取出來 — EXIF 資訊是記錄在大部分圖像格式裏面的 metadata。如果你下載並安裝了 `exiftool` 程式，可以用它把圖檔的 metadata 轉換成文本，於是至少 `diff` 可以用文字呈現的方式向你展示發生了哪些修改：

```
$ echo '*.*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

如果你把專案中的一個影像檔替換成另一個，然後執行 `git diff` 命令的結果如下：

```

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: 2009:04:17 10:12:35-07:00
+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

```

你可以很容易看出來，檔案的大小跟影像的尺寸都發生了改變。

關鍵字擴展

使用 SVN 或 CVS 的開發人員經常要求關鍵字擴展。這在 Git 中主要的問題是，你無法在一個檔案被提交後再修改它，因為 Git 會先對該檔計算 checksum。然而，你可以在檔案 check out 之後注入(inject)一些文字，然後在提交前再把它移除。Git 屬性提供了兩種方式來進行。

首先，你可以把某個 blob 的 SHA-1 checksum 自動注入檔案的 \$Id\$ 欄位。如果在一個或多個檔案上設置了此欄位，當下次你 check out 該分支的時候，Git 會用 blob 的 SHA-1 值替換那個欄位。注意，這不是 commit 物件的 SHA，而是 blob 本身的：

```

$ echo '* .txt ident' >> .gitattributes
$ echo '$Id$' > test.txt

```

下次 check out 這個檔案的時候，Git 注入了 blob 的 SHA 值：

```

$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $

```

然而，這個結果的用處有限。如果你在 CVS 或 Subversion 中用過關鍵字替換，你可以包含一個日期值 -- 而這個 SHA 值沒什麼幫助，因為它相當地隨機，也無法區分某個 SHA 跟另一個 SHA 比起來是比較新或是比較舊。

因此，你可以撰寫自己的篩檢程式，在提交或 checkout 文件時替換關鍵字。有兩種篩檢程式，“clean”和“smudge”。在 `.gitattributes` 檔中，你能對特定的路徑設置一個篩檢程式，然後設置處理檔案的腳本，這些腳本會在檔案 check out 前（“smudge”，見圖 7-2）和提交前（“clean”，見圖 7-3）被執行。這些篩檢程式能夠做各種有趣的事。

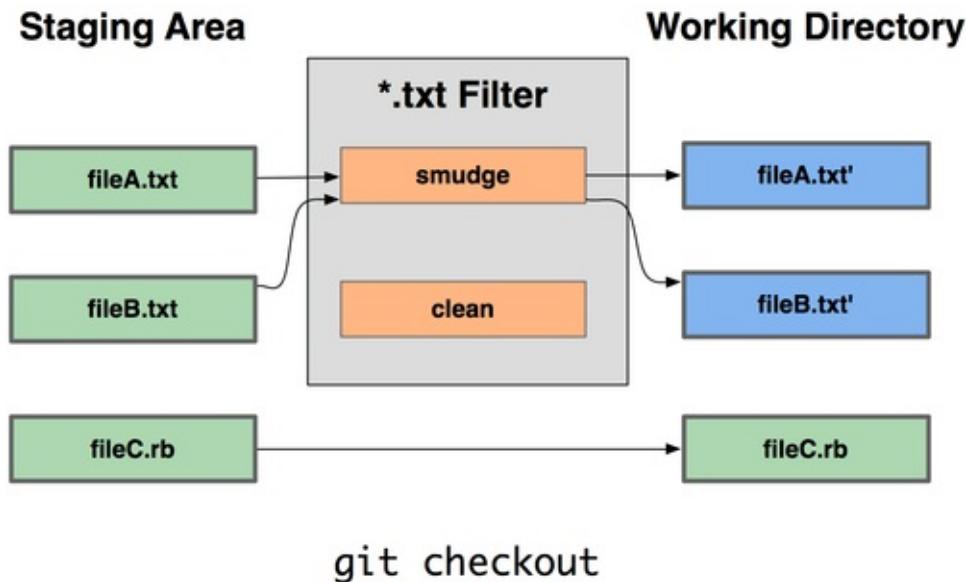


Figure 7-2. “smudge” filter 在 checkout 時執行

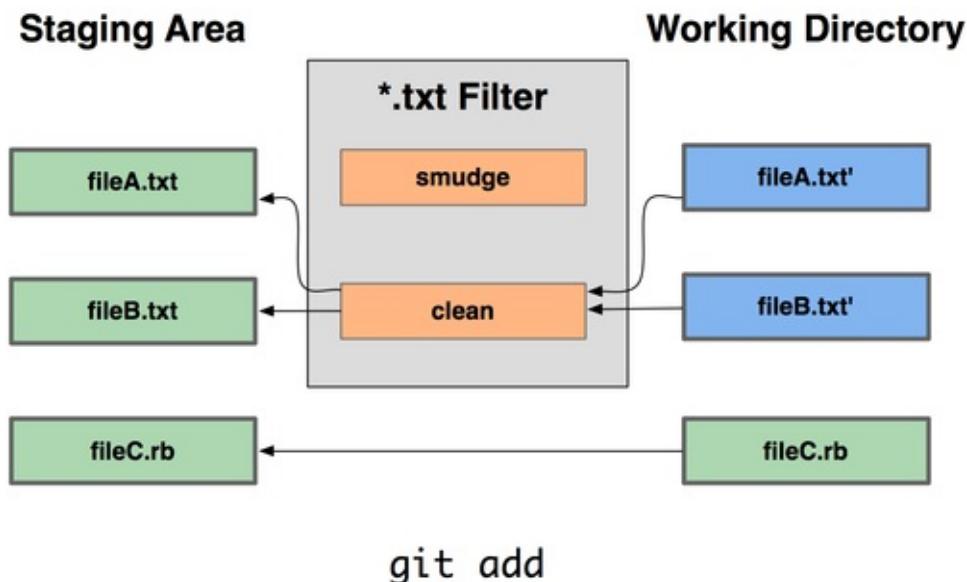


Figure 7-3. “clean” filter 在檔案被 staged 的時候執行

這裡舉一個簡單的例子：在提交前，用 indent（縮進）程式過濾所有 C 原始程式碼。在 `.gitattributes` 檔中設置 “indent” 篩檢程式過濾 `*.c` 文件：

```
*.c      filter=indent
```

然後，通過以下配置，讓 Git 知道 “indent” 篩檢程式在遇到 “smudge” 和 “clean” 時分別該做什麼：

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

於是，當你提交 `*.c` 檔時，`indent` 程式會被觸發，在把它們 `check out` 之前，`cat` 程式會被觸發。但 `cat` 程式在這裡沒什麼實際作用。這樣的組合，使 C 原始程式碼在提交前被 `indent` 程式過濾，非常有效。

另一個有趣的例子是類似 RCS 的 `$Date$` 關鍵字擴展。為了演示，需要一個小腳本，接受檔案名參數，得到專案的最新提交日期，最後把日期寫入該檔。下面用 Ruby 腳本來實現：

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

該腳本從 `git log` 命令中得到最新提交日期，找到檔案中所有的 `$Date$` 字串，最後把該日期填到 `$Date$` 字串中 — 此腳本很簡單，你可以選擇你喜歡的程式設計語言來實現。把該腳本命名為 `expand_date`，放到正確的路徑中，之後需要在 Git 中設置一個篩檢程式（`dater`），讓它在 `check out` 檔案時使用 `expand_date`，在提交時用 Perl 清除之：

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\$\$Date[^\\\$]*\\\$/\$\$Date\\\$/"'
```

這個 Perl 小程式會刪除 `$Date$` 字串裡多餘的字元，恢復 `$Date$` 原貌。到目前為止，你的篩檢程式已經設置完畢，可以開始測試了。打開一個檔，在檔中輸入 `$Date$` 關鍵字，然後設置 Git 屬性：

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

如果把這些修改提交，之後再 `check out`，你會發現關鍵字被替換了：

```
$ 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
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

雖說這項技術對自訂應用來說很有用，但還是要小心，因為 `.gitattributes` 檔會隨著專案一起提交，而篩檢程式（例如：`dater`）不會，所以，它不會在所有地方都成功運作。當你在設計這些篩檢程式時要注意，即使它們無法正常工作，也要讓整個專案運作下去。

匯出倉庫

Git 屬性在將專案匯出歸檔(archive)時也能發揮作用。

export-ignore

當產生一個 archive 時，可以告訴 Git 不要匯出某些檔案或目錄。如果你不想在 archive 中包含一個子目錄或檔案，但想將他們納入專案的版本管理中，你能對應地設置 `export-ignore` 屬性。

例如，在 `test/` 子目錄中有一些測試檔，在專案的壓縮包中包含他們是沒有意義的。因此，可以增加下面這行到 Git 屬性檔中：

```
test/ export-ignore
```

現在，當運行 `git archive` 來創建專案的壓縮包時，那個目錄不會在 archive 中出現。

export-subst

還能對 archives 做一些簡單的關鍵字替換。在第2章中已經可以看到，可以以 `--pretty=format` 形式的簡碼在任何檔中放入 `$Format:$` 字串。例如，如果想在專案中包含一個叫作 `LAST_COMMIT` 的檔，當運行 `git archive` 時，最後提交日期自動地注入進該檔，可以這樣設置：

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

執行 `git archive` 後，打開該檔，會發現其內容如下：

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

合併策略

通過 Git 屬性，還能對專案中的特定檔案使用不同的合併策略。一個非常有用的選項就是，當一些特定檔案發生衝突，Git 不會嘗試合併他們，而使用你這邊的來覆蓋別人的。

如果專案的一個分支有歧義或比較特別，但你想從該分支合併，而且需要忽略其中某些檔，這樣的合併策略是有用的。例如，你有一個資料庫設置檔 `database.xml`，在兩個分支中他們是不同的，你想合併一個分支到另一個，而不弄亂該資料庫檔，可以設置屬性如下：

```
database.xml merge=ours
```

如果合併到另一個分支，`database.xml` 檔不會有合併衝突，顯示如下：

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

這樣，`database.xml` 會保持原樣。

Git Hooks

和其他版本控制系統一樣，當某些重要事件發生時，Git 有方法可以觸發自訂腳本。有兩組掛鉤(hooks)：用戶端和伺服器端。用戶端掛鉤用於用戶端的操作，如提交和合併。伺服器端掛鉤用於 Git 伺服器端的操作，如接收被推送的提交。你可以為了各種不同的原因使用這些掛鉤，下面會講解其中一些。

安裝一個 Hook

掛鉤都被儲存在 Git 目錄下的 `hooks` 子目錄中，即大部分專案中預設的 `.git/hooks`。Git 預設會放置一些腳本範例在這個目錄中，除了可以作為掛鉤使用，這些樣本本身是可以獨立使用的。所有的樣本都是 shell 腳本，其中一些還包含了 Perl 的腳本，不過，任何正確命名的可執行腳本都可以正常使用 — 可以用 Ruby 或 Python，或其他。在 Git 1.6 版本之後，這些樣本檔名都是以 `.sample` 結尾，因此，你必須重新命名。在 Git 1.6 版本之前，這些樣本名都是正確的，但這些樣本不是可執行檔。

把一個正確命名且可執行的檔放入 Git 目錄下的 `hooks` 子目錄中，可以啓動該掛鉤腳本，之後他一直會被 Git 呼叫。隨後會講解主要的掛鉤腳本。

用戶端掛鉤

有許多用戶端掛鉤，以下把他們分為：提交工作流程掛鉤、電子郵件工作流程掛鉤及其他用戶端掛鉤。

提交工作流程掛鉤

有四個掛鉤被用來處理提交的過程。`pre-commit` 掛鉤在鍵入提交資訊前運行，被用來檢查即將提交的快照，例如，檢查是否有東西被遺漏，確認測試是否運行，以及檢查代碼。當從該掛鉤返回非零值時，Git 放棄此次提交，但可以用 `git commit --no-verify` 來忽略。該掛鉤可以被用來檢查程式碼樣式（運行類似 `lint` 的程式），檢查尾部空白（預設掛鉤是這麼做的），檢查新方法（簡體中文版譯注：程式的函數）的說明。

`prepare-commit-msg` 掛鉤在提交資訊編輯器顯示之前，預設資訊被創建之後執行。因此，可以有機會在提交作者看到預設資訊前進行編輯。該掛鉤接收一些選項：擁有提交資訊的檔案路徑，提交類型，以及提交的 SHA-1 (如果這是一個 `amended` 提交)。該掛鉤對通常的提交來說不是很有用，只在自動產生的預設提交資訊的情況下有作用，如提交資訊範本、合併、壓縮和 `amended` 提交等。可以和提交範本配合使用，以程式設計的方式插入資訊。

`commit-msg` 掛鉤接收一個參數，此參數是包含最近提交資訊的暫存檔路徑。如果該掛鉤腳本以非零退出，Git 會放棄提交，因此，可以用來在提交通過前驗證專案狀態或提交資訊。本章上一小節已經展示了使用該掛鉤核對提交資訊是否符合特定的模式。

`post-commit` 掛鉤在整個提交過程完成後運行，他不會接收任何參數，但可以執行 `git log -1 HEAD` 來獲得最後的提交資訊。總之，該掛鉤是作為通知之類使用的。

提交工作流程的用戶端掛鉤腳本可以在任何工作流程中使用，他們經常被用來實施某些策略，但值得注意的是，這些腳本在 `clone` 期間不會被傳送。可以在伺服器端實施策略來拒絕不符合某些策略的推送，但這完全取決於開發者在用戶端使用這些腳本的情況。所以，這些腳本對開發者是有用的，由他們自己設置和維護，而且在任何时候都可以覆蓋或修改這些腳本。

E-mail 工作流掛鉤

有三個可用的用戶端掛鉤用於 e-mail 工作流。當運行 `git am` 命令時，會呼叫他們，因此，如果你沒有在工作流中用到此命令，可以跳過本節。如果你通過 e-mail 接收由 `git format-patch` 產生的補丁，這些掛鉤也許對你有用。

首先執行的是 `applypatch-msg` 掛鉤，他接收一個參數：包含被建議提交資訊的暫存檔案名。如果該腳本以非零值退出，Git 將放棄此補丁。可以使用這個腳本確認提交資訊是否被正確格式化，或讓腳本把提交訊息編輯為正規化。

下一個當透過 `git am` 應用補丁時執行的是 `pre-applypatch` 掛鉤。該掛鉤不接收參數，在補丁被應用之後執行，因此，可以被用來在提交前檢查快照。你能用此腳本執行測試，檢查工作樹。如果有些什麼遺漏，或測試沒通過，腳本會以非零退出，放棄此次 `git am` 的運行，補丁不會被提交。

最後在 `git am` 操作期間執行的掛鉤是 `post-applypatch`。你可以用他來通知一個小組或該補丁的作者，但無法使用此腳本阻止打補丁的過程。

其他用戶端掛鉤

`pre-rebase` 掛鉤在衍合前執行，腳本以非零退出可以中止衍合的過程。你可以使用這個掛鉤來禁止衍合已經推送的提交物件，Git 所安裝的 `pre-rebase` 掛鉤範例就是這麼做的，不過它假定 `next` 是你定義的分支名。因此，你可能要修改樣本，把 `next` 改成你定義過且穩定的分支名。

在 `git checkout` 成功執行後會執行 `post-checkout` 掛鉤。他可以用來為你的專案環境設置合適的工作目錄。例如：放入大的二進位檔案、自動產生的文檔或其他一切你不想納入版本控制的檔。

最後，在 `merge` 命令成功執行後會執行 `post-merge` 掛鉤。他可以用來在 Git 無法跟蹤的工作樹中恢復資料，例如許可權資料。該掛鉤同樣能夠驗證在 Git 控制之外的檔是否存在，當工作樹改變時，你希望可以複製進來的檔案。

伺服器端掛鉤

除了用戶端掛鉤，作為系統管理員，你還可以使用兩個伺服器端的掛鉤對專案實施各種類型的策略。這些掛鉤腳本可以在提交物件推送到伺服器前執行，也可以在推送到伺服器後執行。推送到伺服器前執行的掛鉤(`pre hooks`)可以在任何時候以非零退出，拒絕推送、傳回錯誤訊息給用戶端；還可以如你所願設置足夠複雜的推送策略。

pre-receive and post-receive

處理來自用戶端的推送 (`push`) 操作時最先執行的腳本就是 `pre-receive`。它從標準輸入 (`stdin`) 獲取被推送的引用(`references`)列表；如果它退出時的返回值不是0，那麼所有推送內容都不會被接受。利用此掛鉤腳本可以實現類似保證被更新的索引(`references`)都不是 `non-fast-forward` 類型；抑或檢查執行推送操作的用戶擁有創建、刪除或者推送的許可權，或者他是否對將要修改的每一個檔都有存取權限。

`post-receive` 掛鉤在整個過程完結以後執行，可以用來更新其他系統服務或者通知使用者。它接受與 `pre-receive` 相同的標準輸入資料。應用實例包括給某郵寄清單發信，通知即時整合資料的伺服器，或者更新軟體專案的問題追蹤系統——甚至可以通過分析提交資訊來決定某個問題是否應該被開啓、修改或結案。該腳本無法停止推送程序，不過用戶端在它完成之前將保持連接狀態；所以在用它作一些長時間的操作之前請三思。

update

`update` 腳本和 `pre-receive` 腳本十分類似，除了它會為推送者更新的每一個分支運行一次。假如推送者同時向多個分支推送內容，`pre-receive` 只執行一次，相較之下 `update` 則會為每一個更新的分支運行一次。它不會從標準輸入讀取內容，而是接受三個參數：索引(`reference`)的名字（分支），推送前索引指向的內容的 `SHA-1` 值，以及使用者試圖推送內容的 `SHA-1` 值。如果 `update` 腳本退出時返回非零值，只有相應的那一個索引會被拒絕；其餘的依然會得到更新。

Git 強制策略實例

在本節中，我們應用前面學到的知識建立這樣一個 Git 工作流程：檢查提交資訊的格式，只接受純 fast-forward 內容的推送，並且指定專案中的某些特定用戶只能修改某些特定子目錄。我們將撰寫一個用戶端腳本來提示開發人員他們推送的內容是否會被拒絕，以及一個伺服端腳本來實際執行這些策略。

我使用 Ruby 來撰寫這些腳本，一方面因為它是我喜好的指令碼語言(scripting language)，也因為我覺得它是最接近偽代碼(pseudocode-looking)的指令碼語言；因而即便你不使用 Ruby 也能大致看懂。不過任何其他語言也一樣適用。所有 Git 自帶的範例腳本都是用 Perl 或 Bash 寫的，所以從這些腳本中能找到相當多的這兩種語言的掛鈎範例。

服務端掛鈎

所有服務端的工作都在 hooks (掛鈎) 目錄的 update (更新) 腳本中制定。update 腳本為每一個得到推送的分支運行一次；它接受推送目標的索引(reference)、該分支原來指向的位置、以及被推送的新內容。如果推送是通過 SSH 進行的，還可以獲取發出此次操作的用戶。如果設定所有操作都通過公鑰授權的單一帳號（比如“git”）進行，就有必要通過一個 shell 包裝(wrapper)依據公鑰來判斷用戶的身份，並且設定環境變數來表示該使用者的身份。下面假設嘗試連接的使用者儲存在 \$USER 環境變數裡，我們的 update 腳本首先搜集一切需要的資訊：

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies... \n(#{refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

沒錯，我在用全域變數。別鄙視我——這樣比較利於演示過程。

強制特定的提交資訊格式

我們的第一項任務是指定每一條提交資訊都必須遵循某種特殊的格式。只是設定一個目標，假定每一條資訊必須包含一條形似“ref: 1234”這樣的字串，因為我們需要把每一次提交連結到專案問題追蹤系統裏面的工作項目。我們要逐一檢查每一條推送上來的提交內容，看看提

交資訊是否包含這麼一個字串，然後，如果該提交裡不包含這個字串，以非零返回值退出從而拒絕此次推送。

把 `$newrev` 和 `$oldrev` 變數的值傳給一個叫做 `git rev-list` 的 Git plumbing 命令可以獲取所有提交內容 SHA-1 值的列表。`git rev-list` 基本上是個 `git log` 命令，但它預設只輸出 SHA-1 值而已，沒有其他資訊。所以要獲取由 SHA 值表示的從一次提交到另一次提交之間的所有 SHA 值，可以執行：

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475
```

取得這些輸出內容，迴圈遍歷其中每一個提交的 SHA 值，找出與之對應的提交資訊，然後用規則運算式(*regular expression*)來測試該資訊是否符合某個 pattern。

下面要搞定如何從所有的提交內容中提取出提交資訊。使用另一個叫做 `git cat-file` 的 Git plumbing 工具可以獲得原始的提交資料。我們將在第九章瞭解到這些 plumbing 工具的細節；現在暫時先看一下這條命令會給你什麼：

```
$ 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
```

通過 SHA-1 值獲得提交內容中的提交資訊的一個簡單辦法是找到提交的第一個空白行，然後取出它之後的所有內容。可以使用 Unix 系統的 `sed` 命令來實現這個效果：

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

這條咒語從每一個待提交內容裡提取提交訊息，並且會在提交訊息不符合要求的情況下退出。為了退出腳本和拒絕此次推送，返回一個非零值。整個 method 大致如下：

```
$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
```

把這一段放在 `update` 腳本裡，所有包含不符合指定規則的提交都會遭到拒絕。

實現基於使用者的存取權限控制清單（ACL）系統

假設你需要添加一個使用存取權限控制列表 (access control list, ACL) 的機制來指定哪些使用者對專案的哪些部分有推送許可權。某些使用者具有全部的存取權，其他人只對某些子目錄或者某些特定的檔案具有推送許可權。要搞定這一點，所有的規則將被寫入一個位於伺服器的原始 Git 倉庫的 `acl` 檔。我們讓 `update` 掛鉤檢閱這些規則，審視推送的提交內容中需要修改的所有檔案，然後判定執行推送的用戶是否對所有這些檔案都有許可權。

我們首先要創建這個列表。這裡使用的格式和 CVS 的 ACL 機制十分類似：它由若干行構成，第一欄的內容是 `avail` 或者 `unavail`；下一欄是由逗號分隔的使用者清單，列出這條規則會對哪些使用者生效；最後一欄是這條規則會對哪個目錄生效（空白表示開放訪問）。這些欄位由 `pipe (|)` 字元隔開。

下例中，我們指定幾個管理員，幾個對 `doc` 目錄具有許可權的文件作者，以及一個只對 `lib` 和 `tests` 目錄具有許可權的開發人員，`ACL` 檔看起來像這樣：

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

首先把這些資料讀入到你所能使用的資料結構中。本例中，為保持簡潔，我們暫時只實做 `avail` 的規則（譯注：也就是省略了 `unavail` 部分）。下面這個 `method` 產生一個關聯式陣列，它的主鍵是用戶名，對應的值是一個該用戶有寫入許可權的所有目錄組成的陣列：

```

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

```

針對之前給出的 ACL 規則檔，這個 `get_acl_access_data` method 回傳的資料結構如下：

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

搞定了使用者許可權的資料，下面需要找出這次推送的提交之中，哪些位置被修改，從而確保試圖推送的使用者對這些位置都有許可權。

使用 `git log` 的 `--name-only` 選項（在第二章裡簡單的提過）我們可以輕而易舉的找出一次提交裡有哪些被修改的檔案：

```
$ git log -1 --name-only --pretty=format:'' 9f585d
README
lib/test.rb
```

使用 `get_acl_access_data` 回傳的 ACL 結構來一一核對每一次提交修改的檔案列表，就能判定該用戶是否有許可權推送所有的提交內容：

```

# 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.index(access_path) == 0) # 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

```

以上的大部分內容應該還算容易理解。通過 `git rev-list` 獲取推送到伺服器的提交清單。然後，針對其中每一項，找出它試圖修改的檔案，然後確保執行推送的用戶對這些檔案具有許可權。一個不太容易理解的 Ruby 技巧是 `path.index(access_path) == 0` 這句，如果路徑以 `access_path` 開頭，它會回傳 `True`——這是為了確保 `access_path` 並不是只在允許的路徑之一，而是所有准許全選的目錄都在該目錄之下。

現在，如果提交資訊的格式不對的話，或是修改的檔案在允許的路徑之外的話，你的用戶就不能推送這些提交。

只允許 **Fast-Forward** 類型的推送

剩下的最後一項任務是指定只接受 `fast-forward` 的推送。在 Git 1.6 或者較新的版本裡，只需要設定 `receive.denyDeletes` 和 `receive.denyNonFastForwards` 選項就可以了。但是用掛鈎來實做這個功能，便可以在舊版本的 Git 上運作，並且通過一定的修改，它可以做到只針對某些用戶執行，或者更多以後可能用到的規則。

檢查的邏輯是看看是否有任何的提交在舊版本(revision)裡能找到、但在新版本裡卻找不到。如果沒有，那這是一次純 `fast-forward` 的推送；如果有，那我們拒絕此次推送：

```
# enforces fast-forward only pushes
def check_fast_forward
  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

一切都設定好了。如果現在執行 `chmod u+x .git/hooks/update` —— 這是包含以上內容的案，我們修改它的許可權，然後嘗試推送一個包含非 fast-forward 類型的索引，會得到類似如下：

```
$ 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] Cannot push a non fast-forward reference
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'
```

這裡有幾個有趣的資訊。首先，我們可以看到掛鉤執行的起點：

```
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
```

注意這是你在 `update` 腳本一開頭的地方印出到標準輸出的東西。所有從腳本印出到 `stdout` 的東西都會發送到用戶端，這點很重要。

下一個值得注意的部分是錯誤資訊。

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

第一行是我們的腳本輸出的，在往下是 Git 在告訴我們 `update` 腳本退出時傳回了非零值，因而推送遭到了拒絕。最後一點：

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

每一個被掛鈎拒絕的索引(reference)，你都會看到一條遠端拒絕訊息，解釋它被拒絕是因為一個掛鈎失敗的原因。

而且，如果 ref 標記字串(譯註：例如 ref: 1234)沒有包含在任何的提交裡，我們將看到前面腳本裡印出的錯誤資訊：

```
[POLICY] Your message is not formatted correctly
```

又或者某人想修改一個自己不具備許可權的檔，然後推送了一個包含它的提交，他將看到類似的提示。比如，一個文件作者嘗試推送一個修改了 lib 目錄下某些東西的提交，他會看到

```
[POLICY] You do not have access to push to lib/test.rb
```

都做好了。從這裡開始，只要 update 腳本存在並且可執行，我們的倉庫永遠都不會遭到回轉(rewound)，或者包含不符合要求資訊的提交內容，並且使用者都被鎖在了沙箱裡面。

用戶端掛鈎

這種方法的缺點在於使用者推送內容遭到拒絕後幾乎無法避免的抱怨。辛辛苦苦寫成的代碼在最後時刻慘遭拒絕是令人十分沮喪、迷惑的；更可憐的是他們不得不修改提交歷史來解決問題，有時候這可不是隨便哪個人都做得來的。

這種兩難境地的解答是提供一些用戶端的掛鈎，讓使用者可以用來在他們作出伺服器可能會拒絕的事情時給以警告。這樣的話，用戶們就能在提交--問題變得更難修正之前修正問題。由於掛鈎本身不跟隨 clone 的專案副本分發，所以必須通過其他途徑把這些掛鈎分發到用戶的 .git/hooks 目錄並設為可執行檔。雖然可以在專案裏或用另一個專案分發這些掛鈎，不過全自動的解決方案是不存在的。

首先，你應該在每次提交前檢查你的提交說明訊息，這樣你才能確保伺服器不會因為不合格式的提交說明訊息而拒絕你的更改。為了達到這個目的，你可以增加 commit-msg 掛鈎。如果你使用該掛鈎來讀取第一個參數傳遞的檔案裏的訊息，並且與規定的模式(pattern)作對比，你就可以使 Git 在提交說明訊息不符合條件的情況下，拒絕執行提交。

```
#!/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
```

如果這個腳本放在這個位置 (`.git/hooks/commit-msg`) 並且是可執行的，而你的提交說明訊息沒有做適當的格式化，你會看到：

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

在這個實例中，提交沒有成功。然而如果你的提交說明訊息符合要求的，Git 會允許你提交：

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 files changed, 1 insertions(+), 0 deletions(-)
```

接下來我們要保證沒有修改到 ACL 允許範圍之外的檔案。如果你的專案 `.git` 目錄裡有前面使用過的 `ACL` 檔，那麼以下的 `pre-commit` 脚本將執行裡面的限制規定：

```

#!/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

```

這和服務端的腳本幾乎一樣，除了兩個重要區別。第一，ACL 檔的位置不同，因為這個腳本在當前工作目錄執行，而非 Git 目錄。ACL 檔的目錄必須由這個

```
access = get_acl_access_data('acl')
```

改成這個：

```
access = get_acl_access_data('.git/acl')
```

另一個重要區別是獲取「被修改檔案清單」的方式。在服務端的時候使用了查看提交紀錄的方式，可是目前的提交都還沒被記錄下來呢，所以這個清單只能從暫存區域獲取。原來是這樣：

```
files_modified = `git log -1 --name-only --pretty=format:'#{ref}'`
```

現在要用這樣：

```
files_modified = `git diff-index --cached --name-only HEAD`
```

不同的就只有這兩點——除此之外，該腳本完全相同。一個小陷阱在於它假設在本地執行的帳戶和推送到遠端服務端的相同。如果這二者不一樣，則需要手動設置一下 `$user` 變數。

最後一項任務是檢查確認推送內容中不包含非 fast-forward 類型的索引(reference)，不過這個需求比較少見。以下情況會得到一個非 fast-forward 類型的索引，要麼在某個已經推送過的提交上做衍合，要麼從本地不同分支推送到遠端相同的分支上。

既然伺服器會告訴你不能推送非 fast-forward 內容，而且上面的掛鉤也能阻止強制的推送，唯一剩下的潛在問題就是衍合已經推送過的提交內容。

下面是一個檢查這個問題的 pre-rebase 腳本的例子。它獲取一個所有即將重寫的提交內容的清單，然後檢查它們是否在遠端的索引(reference)裡已經存在。一旦發現某個提交可以從遠端索引裡衍變過來，它就放棄衍合操作：

```
#!/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
```

這個腳本利用了一個第六章「修訂版本選擇」一節中不曾提到的語法。執行這個命令可以獲得一個所有已經完成推送的提交的列表：

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

`SHA^@` 語法解析該次提交的所有祖先。我們尋找任何一個提交，這個提交可以從遠端最後一次提交衍變獲得(reachable)，但從我們嘗試推送的任何一個提交的 SHA 值的任何一個祖先都無法衍變獲得——也就是 fast-forward 的內容。

這個解決方案的缺點在於它可能會很慢而且通常是沒有必要的——只要不用 `-f` 來強制推送，伺服器會自動給出警告並且拒絕推送內容。然而，這是個不錯的練習，而且理論上能幫助用戶避免一個將來不得不回頭修改的衍合操作。

總結

你已經見識過絕大多數通過自訂 Git 用戶端和服務端來適應自己工作流程和專案內容的方式了。你已經學到了各種配置設定(configuration settings)、以檔案為基礎的屬性(file-based attributes)、以及事件掛鉤，你也建置了一個執行強制政策的伺服器。現在，差不多任何你能想像到的工作流程，你應該都能讓 Git 切合你的需要。

Git 與其他系統

世界不是完美的。大多數時候，將所有接觸到的專案全部轉向 Git 是不可能的。有時我們不得不為某個專案使用其他的版本控制系統（VCS, Version Control System），其中比較常見的是 Subversion。你將在本章的第一部分學習使用 `git svn`，這是 Git 為 Subversion 附帶的雙向橋接工具。

或許現在你已經在考慮將先前的專案轉向 Git。本章的第二部分將介紹如何將專案遷移到 Git：先介紹從 Subversion 的遷移，然後是 Perforce，最後介紹如何使用自訂的腳本進行非標準的導入。

Git 與 Subversion

當前，大多數開發中的開源專案以及大量的商業專案都使用 Subversion 來管理源碼。作為最流行的開源版本控制系統，Subversion 已經存在了接近十年的時間。它在許多方面與 CVS 十分類似，後者是前者出現之前代碼控制世界的霸主。

Git 最為重要的特性之一是名為 `git svn` 的 Subversion 雙向橋接工具。該工具把 Git 變成了 Subversion 服務的用戶端，從而讓你在本地享受到 Git 所有的功能，而後直接向 Subversion 伺服器推送內容，彷彿在本地使用了 Subversion 用戶端。也就是說，在其他人忍受古董的同時，你可以在本地享受分支合併，使用暫存區域，衍合以及單項挑揀(cherry-picking)等等。這是個讓 Git 偷偷潛入合作開發環境的好東西，在幫助你的開發同伴們提高效率的同時，它還能幫你勸說團隊讓整個專案框架轉向對 Git 的支持。這個 Subversion 之橋是通向分散式版本控制系統（DVCS, Distributed VCS）世界的神奇隧道。

git svn

Git 中所有 Subversion 橋接命令的基礎是 `git svn`。所有的命令都從它開始。相關的命令數目不少，你將通過幾個簡單的工作流程瞭解到其中常見的一些。

值得注意的是，在使用 `git svn` 的時候，你實際是在與 Subversion 互動，Git 比它要高級複雜的多。儘管可以在本地隨意的進行分支和合併，最好還是通過衍合保持線性的提交歷史，儘量避免類似「與遠端 Git 倉庫同步互動」這樣的操縱。

避免修改歷史再重新推送的做法，也不要同時推送到並行的 Git 倉庫來試圖與其他 Git 用戶合作。Subversion 只能保存單一的線性提交歷史，一不小心就會被搞糊塗。合作團隊中同時有人用 SVN 和 Git，一定要確保所有人都使用 SVN 服務來協作——這會讓生活輕鬆很多。

初始設定

為了展示功能，先要一個具有寫入許可權的 SVN 倉庫。如果想嘗試這個範例，你必須複製一份其中的測試倉庫。比較簡單的做法是使用一個名為 `svnsync` 的工具。較新的 Subversion 版本中都帶有該工具，它將資料編碼為用於網路傳輸的格式。

要嘗試本例，先在本地新建一個 Subversion 倉庫：

```
$ mkdir /tmp/test-svn  
$ svnadmin create /tmp/test-svn
```

然後，允許所有用戶修改 `revprop`——簡單的做法是添加一個總是以 0 作為傳回值的 `pre-revprop-change` 腳本：

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

現在可以呼叫 `svnsync init`，參數加目標倉庫，再加來源倉庫，就可以把該專案同步到本地了：

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

這將建立進行同步所需的屬性(property)。可以通過執行以下命令來 `clone` 程式碼：

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

別看這個操作只花掉幾分鐘，要是你想把源倉庫複製到另一個遠端倉庫，而不是本地倉庫，那將花掉接近一個小時，儘管專案中只有不到 100 次的提交。Subversion 每次只複製一次修改，把它推送到另一個倉庫裡，然後周而復始——驚人的低效率，但是我們別無選擇。

入門

有了可以寫入的 Subversion 倉庫以後，就可以嘗試一下典型的工作流程了。我們從 `git svn clone` 命令開始，它會把整個 Subversion 倉庫導入到一個本地的 Git 倉庫中。提醒一下，這裡導入的是一個貨真價實的 Subversion 倉庫，所以應該把下面的 `file:///tmp/test-svn` 換成你所用的 Subversion 倉庫的 URL：

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
  A    m4/acx_pthread.m4
  A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcd59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcd59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/branches/my-calc-branch r76
```

這相當於針對所提供的 URL 運行了兩條命令—— `git svn init` 加上 `gitsvn fetch`。可能會花上一段時間。我們所用的測試專案僅僅包含 75 次提交並且它的代碼量不算大，所以只有幾分鐘而已。不過，Git 仍然需要提取每一個版本，每次一個，再逐個提交。對於一個包含成百上千次提交的專案，花掉的時間則可能是幾小時甚至數天。

`-T trunk -b branches -t tags` 告訴 Git 該 Subversion 倉庫遵循了基本的分支和標籤命名法則。如果你的主幹(譯注：`trunk`，相當於非分散式版本控制裡的 `master` 分支，代表開發的主線) 分支或者標籤以不同的方式命名，則應做出相應改變。由於該法則的常見性，可以使用 `-s` 來代替整條命令，它意味著標準佈局 (`s` 是 `Standard layout` 的首字母)，也就是前面選項的內容。下面的命令有相同的效果：

```
$ git svn clone file:///tmp/test-svn -s
```

現在，你有了一個有效的 Git 倉庫，包含著導入的分支和標籤：

```
$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

值得注意的是，該工具分配命名空間時和遠端參照的方式不盡相同。`clone` 普通的 Git 倉庫時，可以用 `origin/[branch]` 的形式獲取遠端伺服器上所有可用的分支——分配到遠端服務的名稱下。然而 `git svn` 假定不存在多個遠端伺服器，所以把所有指向遠端服務的引用不加區分(`no namespacing`)的保存下來。可以用 Git 底層(`plumbing`)命令 `show-ref` 來查看所有引用的全名：

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dc92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

而普通的 Git 倉庫應該是這個模樣：

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

這裡有兩個遠端伺服器：一個名為 `gitserver`，具有一個 `master` 分支；另一個叫 `origin`，具有 `master` 和 `testing` 兩個分支。

注意本例中通過 `git svn` 導入的遠端參照，(`Subversion` 的)標籤是當作遠端分支添加的，而不是真正的 Git 標籤。導入的 Subversion 倉庫彷彿是有一個帶有不同分支的 `tags` 遠端伺服器。

提交到 Subversion

有了可以開展工作的（本地）倉庫以後，你可以開始對該專案做出貢獻並向上游倉庫提交內容了，Git 這時相當於一個 SVN 用戶端。假如編輯了一個檔並進行提交，那麼這次提交僅存在於本地的 Git 而非 Subversion 伺服器上：

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
 1 files changed, 1 insertions(+), 1 deletions(-)
```

接下來，可以將作出的修改推送到上游。值得注意的是，Subversion 的使用流程也因此改變了——你可以在離線狀態下進行多次提交然後一次性的推送到 Subversion 的伺服器上。向 Subversion 伺服器推送的命令是 `git svn dcommit`：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      README.txt
Committed r79
    M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

所有在原 Subversion 資料基礎上提交的 commit 會一一提交到 Subversion，然後你本地 Git 的 commit 將被重寫，加入一個特別標識。這一步很重要，因為它意味著所有 commit 的 SHA-1 指都會發生變化。這也是同時使用 Git 和 Subversion 兩種服務作為遠端服務不是個好主意的原因之一。檢視以下最後一個 commit，你會找到新添加的 `git-svn-id`（譯注：即本段開頭所說的特別標識）：

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sat May 2 22:06:44 2009 +0000

    Adding git-svn instructions to the README

git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

注意看，原本以 `97031e5` 開頭的 SHA-1 校驗值在提交完成以後變成了 `938b1a5`。如果既想要向 Git 遠端伺服器推送內容，又要推送到 Subversion 遠端伺服器，則必須先向 Subversion 推送（`dcommit`），因為該操作會改變所提交的資料內容。

拉取最新進展

如果要與其他開發者協作，總有那麼一天你推送完畢之後，其他人發現他們推送自己修改的時候（與你推送的內容）產生衝突。這些修改在你合併之前將一直被拒絕。在 `git svn` 裡這種情況像這樣：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

為了解決該問題，可以執行 `git svn rebase`，它會拉取伺服器上所有最新的改變，再於此基礎上衍合你的修改：

```
$ git svn rebase
      M README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

現在，你做出的修改都在 Subversion 伺服器上，所以可以順利的運行 `dcommit`：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
      M README.txt
Committed r81
      M README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

需要牢記的一點是，Git 要求我們在推送之前先合併上游倉庫中最新的內容，而 `git svn` 只要求存在衝突的時候才這樣做。假如有人向一個檔推送了一些修改，這時你要向另一個文件推送一些修改，那麼 `dcommit` 將正常工作：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
      M configure.ac
Committed r84
      M autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
      M configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M  autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

這一點需要牢記，因為它的結果是推送之後專案處於一個不完整存在於任何主機上的狀態。如果做出的修改無法相容但沒有產生衝突，則可能造成一些很難確診的難題。這和使用 Git 伺服器是不同的——在 Git 世界裡，發佈之前，你可以在用戶端系統裡完整的測試專案的狀態，而在 SVN 永遠都沒法確保提交前後專案的狀態完全一樣。

即使還沒打算進行提交，你也應該用這個命令從 Subversion 伺服器拉取最新修改。你可以執行 `git svn fetch` 獲取最新的資料，不過 `git svn rebase` 才會在獲取之後在本地進行更新。

```
$ git svn rebase
      M generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

不時地執行一下 `git svn rebase` 可以確保你的代碼沒有過時。不過，執行該命令時需要確保工作目錄的整潔。如果在本地做了修改，則必須在執行 `git svn rebase` 之前暫存工作、或暫時提交內容——否則，該命令會發現衍合的結果包含著衝突因而終止。

Git 分支問題

習慣了 Git 的工作流程以後，你可能會創建一些特性分支，完成相關的開發工作，然後合併他們。如果要用 `git svn` 向 Subversion 推送內容，那麼最好是每次用衍合來併入一個單一分支，而不是直接合併。使用衍合的原因是 Subversion 只有一個線性的歷史而不像 Git 那樣處理合併，所以 `Git svn` 在把快照轉換為 Subversion 的 commit 時只能包含第一個祖先。

假設分支歷史如下：創建一個 `experiment` 分支，進行兩次提交，然後合併到 `master`。在 `dcommit` 的時候會得到如下輸出：

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
      M CHANGES.txt
Committed r85
      M CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
      M COPYING.txt
      M INSTALL.txt
Committed r86
      M INSTALL.txt
      M COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

在一個包含了合併歷史的分支上使用 `dcommit` 可以成功運行，不過在 Git 專案的歷史中，它沒有重寫你在 `experiment` 分支中的兩個 commit——取而代之的是，這些改變出現在了 SVN 版本中同一個合併 commit 中。

在別人 `clone` 該專案的時候，只能看到這個合併 commit 包含了所有發生過的修改；他們無法獲知修改的作者和時間等提交資訊。

Subversion 分支

Subversion 的分支和 Git 中的不盡相同；避免過多的使用可能是最好方案。不過，用 `git svn` 創建和提交不同的 Subversion 分支仍是可行的。

創建新的 SVN 分支

要在 Subversion 中建立一個新分支，可以執行 `git svn branch [分支名]`：

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

這相當於在 Subversion 中的 `svn copy trunk branches/opera` 命令並且對 Subversion 伺服器進行了相關操作。值得提醒的是它沒有檢出(check out)並轉換到那個分支；如果現在進行提交，將提交到伺服器上的 `trunk`，而非 `opera`。

切換當前分支

Git 通過搜尋提交歷史中 Subversion 分支的頭部(tip)來決定 `dcommit` 的目的地——而它應該只有一個，那就是當前分支歷史中最近一次包含 `git-svn-id` 的提交。

如果需要同時在多個分支上提交，可以通過導入 Subversion 上某個其他分支的 `commit` 來建立以該分支為 `dcommit` 目的地的本地分支。比如你想擁有一個並行維護的 `opera` 分支，可以執行

```
$ git branch opera remotes/opera
```

然後，如果要把 `opera` 分支併入 `trunk`（本地的 `master` 分支），可以使用普通的 `git merge`。不過最好提供一條描述提交的資訊（通過 `-m`），否則這次合併的記錄會是「Merge branch `opera`」，而不是任何有用的東西。

記住，雖然使用了 `git merge` 來進行這次操作，並且合併過程可能比使用 Subversion 簡單一些（因為 Git 會自動找到適合的合併基礎），這並不是一次普通的 Git 合併提交。最終它將被推送回 Subversion 伺服器上，而 Subversion 伺服器上無法處理包含多個祖先的 `commit`；因而在推送之後，它將變成一個包含了所有在其他分支上做出的改變的單一 `commit`。把一個分支合併到另一個分支以後，你沒法像在 Git 中那樣輕易的回到那個分支上繼續工作。提交時

執行的 `dcommit` 命令擦掉了所有關於哪個分支被併入的資訊，因而以後的合併基礎計算將是不正確的——`dcommit` 讓 `git merge` 的結果變得類似於 `git merge --squash`。不幸的是，我們沒有什麼好辦法來避免該情況——Subversion 無法儲存這個資訊，所以在使用它作為伺服器的時候你將永遠為這個缺陷所困。為了不出現這種問題，在把本地分支（本例中的 `opera`）併入 `trunk` 以後應該立即將其刪除。

對應 Subversion 的命令

`git svn` 工具集合了若干個與 Subversion 類似的效果，對應的命令可以簡化向 Git 的轉化過程。下面這些命令能實現 Subversion 的這些功能。

SVN 風格的歷史紀錄

習慣了 Subversion 的人可能想以 SVN 的風格顯示歷史，運行 `git svn log` 可以讓提交歷史顯示為 SVN 格式：

```
$ git svn log
-----
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines
autogen change

-----
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines
Merge branch 'experiment'

-----
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines
updated the changelog
```

關於 `git svn log`，有兩點需要注意。首先，它可以離線工作，不像 `svn log` 命令，需要向 Subversion 伺服器索取資料。其次，它僅僅顯示已經提交到 Subversion 伺服器上的 `commit`。在本地尚未 `dcommit` 的 Git 資料不會出現在這裡；其他人向 Subversion 伺服器新提交的資料也不會顯示。等於說是顯示了最近已知 Subversion 伺服器上的狀態。

SVN Annotation

類似 `git svn log` 命令模擬了 `svn log` 命令的離線操作，`svn annotate` 的等效命令是 `git svn blame [檔案名]`。其輸出如下：

```
$ 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
```

同樣，它不顯示本地的 Git 提交以及 Subversion 上後來更新的內容。

SVN 伺服器資訊

還可以使用 `git svn info` 來獲取與執行 `svn info` 類似的資訊：

```
$ 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)
```

它與 `blame` 和 `log` 的相同點在於離線運行以及只更新到最後一次與 Subversion 伺服器通信的狀態。

忽略 Subversion 所忽略的

假如 `clone` 了一個包含了 `svn:ignore` 屬性的 Subversion 倉庫，就有必要建立對應的 `.gitignore` 文件來防止意外提交一些不應該提交的文件。`git svn` 有兩個有助於改善該問題的命令。第一個是 `git svn create-ignore`，它自動建立對應的 `.gitignore` 檔，以便下次提交的時候可以包含它。

第二個命令是 `git svn show-ignore`，它把需要放進 `.gitignore` 檔中的內容列印到標準輸出，方便我們把輸出重定向到專案的黑名單檔(exclude file)：

```
$ git svn show-ignore > .git/info/exclude
```

這樣一來，避免了 `.gitignore` 對專案的干擾。如果你是一個 Subversion 團隊裡唯一的 Git 用戶，而其他隊友不喜歡專案裏出現 `.gitignore` 檔案，該方法是你的不二之選。

Git-Svn 總結

`git svn` 工具集在當前不得不使用 Subversion 伺服器或者開發環境要求使用 Subversion 伺服器的時候格外有用。不妨把它看成一個跛腳的 Git，然而，你還是有可能在轉換過程中碰到一些困惑你和合作者們的謎題。為了避免麻煩，試著遵守如下守則：

- 保持一個不包含由 `git merge` 產生的 `commit` 的線性提交歷史。將在主線分支外進行的開發通通衍合回主線；避免直接合併。
- 不要單獨建立和使用一個 Git 服務來搞合作。可以為了加速新開發者的 `clone` 進程建立一個，但是不要向它提供任何不包含 `git-svn-id` 條目的內容。甚至可以添加一個 `pre-receive` 掛鉤，在每一個提交資訊中檢查 `git-svn-id`，並拒絕提交那些不包含它的 `commit`。

如果遵循這些守則，在 Subversion 上工作還可以接受。然而，如果能遷徙到真正的 Git 伺服器，則能為團隊帶來更多好處。

遷移到 Git

如果在其他版本控制系統(VCS)中保存了某專案的代碼而後決定轉而使用 Git，那麼該專案必須經歷某種形式的遷移。本節將介紹 Git 中包含的一些針對常見系統的導入腳本(importer)，並將展示編寫自訂的導入腳本的方法。

導入

你將學習到如何從專業重量級的版本控制系統(SCM)中匯入資料——Subversion 和 Perforce——因為據我所知這二者的用戶是（向 Git）轉換的主要群體，而且 Git 為此二者附帶了高品質的轉換工具。

Subversion

讀過前一節有關 `git svn` 的內容以後，你應該能輕而易舉的根據其中的指導來 `git svn clone` 一個倉庫了；然後，停止 Subversion 的使用，向一個新 Git server 推送，並開始使用它。想保留歷史記錄，所花的時間應該不過就是從 Subversion 伺服器拉取資料的時間（可能要等上好一會就是了）。

然而，這樣的匯入並不完美；而且還要花那麼多時間，不如乾脆一次把它做對！首當其衝的任務是作者資訊。在 Subversion，每個提交者都在主機上有一個用戶名，記錄在提交資訊中。上節例子中多處顯示了 schacon，比如 `blame` 的輸出以及 `git svn log`。如果想讓這條資訊更好的映射到 Git 作者資料裡，則需要從 Subversion 用戶名到 Git 作者的一個映射關係。建立一個叫做 `user.txt` 的檔，用如下格式表示映射關係：

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

通過以下命令可以獲得 SVN 作者的列表：

```
$ svn log ^/ --xml | grep -P "^<author" | sort -u | \
perl -pe 's/<author>(.*)</author>/\$1 = /' > users.txt
```

它將輸出 XML 格式的日誌——你可以找到作者，建立一個單獨的列表，然後從 XML 中抽出需要的資訊。（顯而易見，本方法要求主機上安裝了 `grep`，`sort` 和 `perl`。）然後把輸出重定向到 `user.txt` 檔，然後就可以在每一項的後面添加相應的 Git 使用者資料。

為 `git svn` 提供該檔可以讓它更精確的映射作者資料。你還可以在 `clone` 或者 `init` 後面添加 `--no-metadata` 來阻止 `git svn` 包含那些 Subversion 的附加資訊。這樣 `import` 命令就變成了：

```
$ git svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata -s my_project
```

現在 `my_project` 目錄下導入的 Subversion 應該比原來整潔多了。原來的 `commit` 看上去是這樣：

```
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
```

現在是這樣：

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk
```

不僅作者一項乾淨了不少，`git-svn-id` 也就此消失了。

你還需要一點 `post-import`（導入後）清理工作。最起碼的，應該清理一下 `git svn` 創建的那些怪異的索引結構。首先要移動標籤，把它們從奇怪的遠端分支變成實際的標籤，然後把剩下的分支移動到本地。

要把標籤變成合適的 Git 標籤，執行

```
$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep -v @ | while read tagname; do
```

該命令將原本以 `tag/` 開頭的遠端分支的索引變成真正的 (lightweight) 標籤。

接下來，把 `refs/remotes` 下面剩下的索引(reference)變成本地分支：

```
$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -v @ | while read branchname; do
```

現在所有的舊分支都變成真正的 Git 分支，所有的舊標籤也變成真正的 Git 標籤。最後一項工作就是把新建的 Git 伺服器添加為遠端伺服器並且向它推送。為了讓所有的分支和標籤都得到上傳，我們使用這條命令：

```
$ 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
```

所有的分支和標籤現在都應該整齊乾淨的躺在新的 Git 伺服器裡了。

Perforce

你將瞭解到的下一個被導入的系統是 Perforce. Git 發行的時候同時也附帶了一個 Perforce 導入腳本，不過它是包含在源碼的 `contrib` 部分——而不像 `git svn` 那樣預設就可以使用。執行它之前必須獲取 Git 的源碼，可以在 git.kernel.org 下載：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

在這個 `fast-import` 目錄下，應該有一個叫做 `git-p4` 的 Python 可執行腳本。主機上必須裝有 Python 和 `p4` 工具該導入才能正常進行。例如，你要從 Perforce 公共代碼倉庫（譯注：`Perforce Public Depot`，Perforce 官方提供的代碼寄存服務）導入 `Jam` 專案。為了設定用戶端，我們要把 `P4PORT` 環境變數 `export` 到 Perforce 倉庫：

```
$ export P4PORT=public.perforce.com:1666
```

執行 `git-p4 clone` 命令將從 Perforce 伺服器導入 `Jam` 專案，我們需要給出倉庫和專案的路徑以及導入的目標路徑：

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

現在去 `/opt/p4import` 目錄執行一下 `git log`，就能看到導入的成果：

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into
the main part of the document. Built new tar/zip balls.

Only 16 months later.

[git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

Update derived jamgram.c

[git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

每一個 commit 裡都有一個 git-p4 識別字。這個識別字可以保留，以防以後需要引用 Perforce 的修改版本號。然而，如果想刪除這些識別字，現在正是時候——開始在新倉庫上工作之前。可以通過 git filter-branch 來批量刪除這些識別字：

```
$ git filter-branch --msg-filter '
    sed -e "/^\\[git-p4:/d"
'

Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

現在執行一下 git log，你會發現這些 commit 的 SHA-1 校驗值都發生了改變，而那些 git-p4 字串則從提交資訊裡消失了：

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800

Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into
the main part of the document. Built new tar/zip balls.

Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800

Update derived jamgram.c
```

至此導入已經完成，可以開始向新的 Git 伺服器推送了。

自定導入腳本

如果你的系統不是 Subversion 或 Perforce 之一，先上網找一下有沒有與之對應的導入腳本——導入 CVS，Clear Case，Visual Source Safe，甚至存檔目錄的導入腳本已經存在。假如這些工具都不適用，或者使用的工具很少見，抑或你需要導入過程具有更多可制定性，則應該使用 `git fast-import`。該命令從標準輸入讀取簡單的指令來寫入具體的 Git 資料。這樣創建 Git 物件比執行純 Git 命令或者手動寫物件要簡單的多（更多相關內容見第九章）。通過它，你可以編寫一個導入腳本來從導入來源讀取必要的資訊，同時在標準輸出直接輸出相關指令(instructions)。你可以執行該腳本並把它的輸出管道連接(pipe)到 `git fast-import`。

下面演示一下如何編寫一個簡單的導入腳本。假設你在進行一項工作，並且按時通過把工作目錄複寫為以時間戳記 `back_YY_MM_DD` 命名的目錄來進行備份，現在你需要把它們導入 Git。目錄結構如下：

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

為了導入到一個 Git 目錄，我們首先回顧一下 Git 儲存資料的方式。你可能還記得，Git 本質上是一個 `commit` 物件的鏈表，每一個物件指向一個內容的快照。而這裡需要做的工作就是告訴 `fast-import` 內容快照的位置，什麼樣的 `commit` 資料指向它們，以及它們的順序。我們採取一次處理一個快照的策略，為每一個內容目錄建立對應的 `commit`，每一個 `commit` 與前一個 `commit` 建立連結。

正如在第七章“Git 執行策略一例”一節中一樣，我們將使用 Ruby 來編寫這個腳本，因為它是我日常使用的語言而且閱讀起來簡單一些。你可以用任何其他熟悉的語言來重寫這個例子——它僅需要把必要的資訊列印到標準輸出而已。同時，如果你在使用 Windows，這意味著你要特別留意不要在換行的時候引入回車符（譯注：`carriage returns`，Windows 換行時加入的符號，通常說的 `\r`）——Git 的 `fast-import` 對僅使用分行符號（LF）而非 Windows 的回車符（CRLF）要求非常嚴格。

首先，進入目標目錄並且找到所有子目錄，每一個子目錄將作為一個快照被導入為一個 `commit`。我們將依次進入每一個子目錄並列印所需的命令來匯出它們。腳本的主迴圈大致是這樣：

```

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

```

我們在每一個目錄裡執行 `print_export`，它會取出上一個快照的索引和標記並返回本次快照的索引和標記；由此我們就可以正確的把二者連接起來。”標記（mark）”是 `fast-import` 中對 `commit` 識別字的叫法；在創建 `commit` 的同時，我們逐一賦予一個標記以便以後在把它連接到其他 `commit` 時使用。因此，在 `print_export` 方法中要做的第一件事就是根據目錄名產生一個標記：

```
mark = convert_dir_to_mark(dir)
```

實現該函數的方法是建立一個目錄的陣列序列並使用陣列的索引值作為標記，因為標記必須是一個整數。這個方法大致是這樣的：

```

$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end

```

有了整數來代表每個 `commit`，我們現在需要提交附加資訊中的日期。由於日期是用目錄名表示的，我們就從中解析出來。`print_export` 文件的下一行將是：

```
date = convert_dir_to_date(dir)
```

而 `convert_dir_to_date` 則定義為

```

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

```

它為每個目錄回傳一個 `integer`。提交附加資訊裡最後一項所需的是提交者資料，我們在一個全域變數中直接定義之：

```
$author = 'Scott Chacon <schacon@example.com>'
```

我們差不多可以開始為導入腳本輸出提交資料了。第一項資訊指明我們定義的是一個 `commit` 物件以及它所在的分支，隨後是我們產生的標記、提交者資訊以及提交備註，然後是前一個 `commit` 的索引，如果有的話。程式碼大致像這樣：

```

# 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

```

為了簡化，時區寫死(hardcode)為 (-0700)。如果是從其他版本控制系統導入，則必須以變數的形式指明時區。提交訊息必須以特定格式給出：

```
data (size)\n(contents)
```

該格式包含了「`data`」這個字、所讀取資料的大小、一個分行符號，最後是資料本身。由於隨後指明檔案內容的時候要用到相同的格式，我們寫一個輔助方法，`export_data`：

```

def export_data(string)
  print "data #{string.size}\n#{string}"
end

```

唯一剩下的就是每一個快照的內容了。這簡單的很，因為它們分別處於一個目錄——你可以輸出 `deleeeall` 命令，隨後是目錄中每個檔的內容。`Git` 會正確的記錄每一個快照：

```

puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end

```

注意：由於很多系統把每次修訂看作一個 commit 到另一個 commit 的變化量，fast-import 也可以依據每次提交獲取一個命令來指出哪些檔被添加，刪除或者修改過，以及修改的內容。我們將需要計算快照之間的差別並且僅僅給出這項資料，不過該做法要複雜很多——還不如直接把所有資料丟給 Git 讓它自己搞清楚。假如前面這個方法更適用於你的資料，參考 `fast-import` 的 man 說明頁面來瞭解如何以這種方式提供資料。

列舉新檔內容或者指明帶有新內容的已修改檔的格式如下：

```

M 644 inline path/to/file
data (size)
(file contents)

```

這裡，644 是許可權模式（如果有執行檔，則需要偵測之並設定為 755），而 inline 說明我們在本行結束之後立即列出檔的內容。我們的 `inline_data` 方法大致是：

```

def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end

```

我們再次使用了前面定義過的 `export_data`，因為這裡和指明提交注釋的格式如出一轍。

最後一項工作是回傳當前的標記以便下次迴圈的使用。

```
return mark
```

注意：如果你是在 Windows 上執行，一定記得添加一項額外的步驟。前面提過，Windows 使用 CRLF 作為換行字元而 Git fast-import 只接受 LF。為了避開這個問題來滿足 git fast-import，你需要讓 ruby 用 LF 取代 CRLF：

```
$stdout.binmode
```

搞定了。現在執行該腳本，你將得到如下內容：

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

要執行導入腳本，在需要導入的目錄把該內容用管道定向(pipe)到 `git fast-import`。你可以建立一個空目錄然後執行 `git init` 作為起點，然後執行該腳本：

```
$ 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:        18 (      1 duplicates          )
    blobs :           7 (      1 duplicates          0 deltas)
    trees :           6 (      0 duplicates          1 deltas)
    commits:          5 (      0 duplicates          0 deltas)
    tags :            0 (      0 duplicates          0 deltas)
Total branches:       1 (      1 loads      )
    marks:           1024 (     5 unique      )
    atoms:            3
Memory total:         2255 KiB
    pools:            2098 KiB
    objects:          156 KiB
-----
pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 33554432
pack_report: core.packedGitLimit = 268435456
pack_report: pack_used_ctr = 9
pack_report: pack_mmap_calls = 5
pack_report: pack_open_windows = 1 /
pack_report: pack_mapped = 1356 / 1356
-----
```

你會發現，在它成功執行完畢以後，會給出一堆有關已完成工作的資料。上例在一個分支導入了5次提交資料，包含了18個物件。現在可以執行 `git log` 來檢視新的歷史：

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03
```

就這樣——一個乾淨整潔的 Git 倉庫。需要注意的是此時沒有任何內容被檢出(**checked out**)——剛開始目前的目錄裡沒有任何檔。要獲取它們，你得轉到 `master` 分支的所在：

```
$ ls  
$ git reset --hard master  
HEAD is now at 10bfe7d imported from current  
$ ls  
file.rb lib
```

`fast-import` 還可以做更多——處理不同的檔案模式、二進位檔案、多重分支與合併、標籤、進展標識(progress indicators)等等。一些更加複雜的實例可以在 Git 源碼的 `contrib/fast-import` 目錄裡找到；較佳的其中之一是前面提過的 `git-p4` 腳本。

總結

現在的你應該掌握了在 Subversion 上使用 Git，以及把幾乎任何現存倉庫在不遺漏資料的情況下導入為 Git 倉庫。下一章將介紹 Git 內部的原始資料格式，從而使你能親手鍛造其中的每一個位元組，如果需要的話。

Git 內部原理

不管是從前面的章節直接跳到了本章，還是讀完了其餘各章一直到這，你都將在本章見識 Git 的內部工作原理和實現方式。我個人發現學習這些內容對於理解 Git 的用處和強大是非常重要的，不過也有人認為這些內容對於初學者來說可能難以理解且過於複雜。正因如此我把這部分內容放在最後一章，你在學習過程中可以先閱讀這部分，也可以晚點閱讀這部分，這完全取決於你自己。

既然已經讀到這了，就讓我們開始吧。首先要弄明白一點，從根本上來講 Git 是一套內容定址 (content-addressable) 檔案系統，在此之上提供了一個 VCS 使用者介面。馬上你就會學到這意味著什麼。

早期的 Git (主要是 1.5 之前版本) 的使用者介面要比現在複雜得多，這是因為它更側重于成為檔案系統而不是一套更精緻的 VCS。最近幾年改進了 UI 從而使它跟其他任何系統一樣清晰易用。即便如此，還是經常會有一些陳腔濫調提到早期 Git 的 UI 複雜又難學。

內容定址檔案系統這一層相當酷，在本章中我會先講解這部分。隨後你會學到傳輸機制和最終要使用的各種倉庫管理任務。

底層命令 (Plumbing) 和高層命令 (Porcelain)

本書講解了使用 `checkout`, `branch`, `remote` 等共約 30 個 Git 命令。然而由於 Git 一開始被設計成供 VCS 使用的工具集，而不是一整套 user-friendly 的 VCS，它還包含了許多底層命令，這些命令用於以 UNIX 風格使用或由腳本呼叫。這些命令一般被稱為“plumbing”命令（底層命令），其他的更友好的命令則被稱為“porcelain”命令（高層命令）。

本書前八章主要專門討論高層命令。本章將主要討論底層命令以理解 Git 的內部工作機制、演示 Git 如何及為何要以這種方式工作。這些命令主要不是用來從命令列手工使用的，更多的是用來為其他工具和自訂腳本服務的。

當你在一個新目錄或已有目錄內執行 `git init` 時，Git 會創建一個 `.git` 目錄，幾乎所有 Git 儲存和操作的內容都位於該目錄下。如果你要備份或複製一個倉庫，基本上將這一目錄拷貝至其他地方就可以了。本章基本上都討論該目錄下的內容。該目錄結構如下：

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

該目錄下有可能還有其他檔，但這是一個全新的 `git init` 生成的倉庫，所以預設情況下這些就是你能看到的結構。新版本的 Git 不再使用 `branches` 目錄，`description` 檔僅供 GitWeb 程式使用，所以不用關心這些內容。`config` 檔包含了專案特有的配置選項，`info` 目錄保存了一份不希望在 `.gitignore` 檔中管理的忽略模式 (ignored patterns) 的全域可執行檔。`hooks` 目錄包含了第六章詳細介紹的用戶端或服務端鉤子腳本。

另外還有四個重要的檔案或目錄：`HEAD` 及 `index` 檔，`objects` 及 `refs` 目錄。這些是 Git 的核心部分。`objects` 目錄存放所有資料內容，`refs` 目錄存放指向資料 (分支) 的提交物件的指標，`HEAD` 檔指向當前分支，`index` 檔保存了暫存區域資訊。馬上你將詳細瞭解 Git 是如何操縱這些內容的。

Git 物件

Git 是一套內容定址檔案系統。很不錯。不過這是什麼意思呢？這種說法的意思是，從內部來看，Git 是簡單的 **key-value** 資料儲存。它允許插入任意類型的內容，並會回傳一個鍵值，通過該鍵值可以在任何時候再取出該內容。可以通過底層命令 `hash-object` 來示範這點，傳一些資料給該命令，它會將資料保存在 `.git` 目錄並回傳表示這些資料的鍵值。首先初使化一個 Git 倉庫並確認 `objects` 目錄是空的：

```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git 初始化了 `objects` 目錄，同時在該目錄下創建了 `pack` 和 `info` 子目錄，但是該目錄下沒有其他常規檔。我們往這個 Git 資料庫裡儲存一些文本：

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

參數 `-w` 指示 `hash-object` 命令儲存(資料)物件，若不指定這個參數該命令僅僅回傳鍵值。`--stdin` 指定從標準輸入裝置(`stdin`)來讀取內容，若不指定這個參數，`hash-object` 就需要指定一個要儲存的檔案路徑。該命令輸出長度為 40 個字元的校驗和(`checksum hash`)。這是個 **SHA-1** 雜湊值——其值為要儲存的資料加上你馬上會瞭解到的一種頭資訊(`header`)的校驗和。現在可以查看到 Git 已經儲存了資料：

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

可以在 `objects` 目錄下看到一個檔。這便是 Git 儲存資料內容的方式——為每份內容生成一個檔，取得該內容與頭資訊的 **SHA-1** 校驗和，創建以該校驗和前兩個字元為名稱的子目錄，並以(校驗和)剩下 38 個字元為檔命名(保存至子目錄下)。

通過 `cat-file` 命令可以將資料內容取回。該命令是查看 Git 對象的瑞士軍刀。傳入 `-p` 參數可以讓 `cat-file` 命令輸出資料內容的類型：

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

可以往 Git 中添加更多內容並取回了。也可以直接添加檔案。比方說可以對一個檔進行簡單的版本控制。首先，創建一個新檔，並把檔案內容儲存到資料庫中：

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

接著往該檔中寫入一些新內容並再次保存：

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

資料庫中已經將檔案的兩個新版本連同一開始的內容保存下來了：

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

再將檔案修復到第一個版本：

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

或恢復到第二個版本：

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

需要記住的是幾個版本的檔 SHA-1 值可能與實際的值不同，其次，儲存的並不是檔案名而僅僅是檔案內容。這種物件類型稱為 blob。通過傳遞 SHA-1 值給 `cat-file -t` 命令可以讓 Git 返回任何物件的類型：

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree 物件

接下去來看 tree 物件，tree 物件可以儲存檔案名，同時也允許將一組檔案儲存在一起。Git 以一種類似 UNIX 檔案系統但更簡單的方式來儲存內容。所有內容以 tree 或 blob 物件儲存，其中 tree 物件對應於 UNIX 中的目錄，blob 物件則大致對應於 inodes 或檔案內容。一個單獨的 tree 物件包含一條或多條 tree 記錄，每一條記錄含有一個指向 blob 或子 tree 物件的 SHA-1 指標，並附有該物件的許可權模式 (mode)、類型和檔案名資訊。以 simplegit 專案為例，最新的 tree 可能是這個樣子：

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

`master^{tree}` 表示 `master` 分支上最新提交指向的 tree 物件。請注意 `lib` 子目錄並非一個 blob 物件，而是一個指向別一個 tree 物件的指標：

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

從概念上來講，Git 保存的資料如圖 9-1 所示。

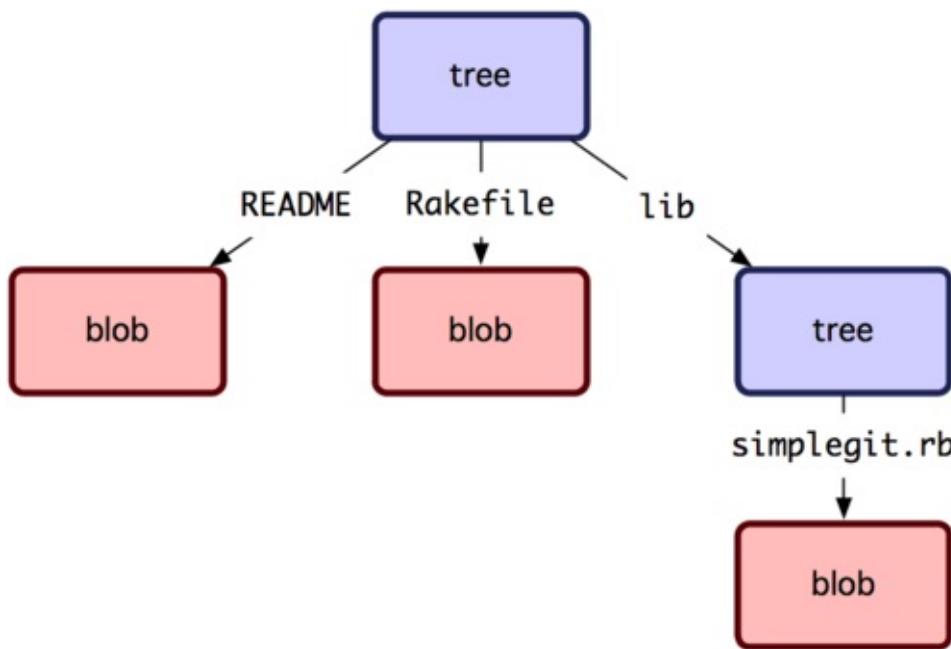


Figure 9-1. Git 物件模型的簡化版

你可以自己創建 tree。通常 Git 根據你的暫存區域或 index 來創建並寫入一個 tree。因此要創建一個 tree 物件的話首先要通過將一些檔暫存從而創建一個 index。可以使用 plumbing 命令 `update-index` 為一個單獨檔 — `test.txt` 檔的第一個版本 — 創建一個 index。通過該命令人工地將 `test.txt` 檔的首個版本加入到了一個新的暫存區域中。由於該檔原先並不在暫存區

域中 (甚至就連暫存區域也還沒被創建出來呢)，必須傳入 `--add` 參數；由於要添加的檔並不在目前的目錄下而是在資料庫中，必須傳入 `--cacheinfo` 參數。同時指定檔案模式，SHA-1 值和檔案名：

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

在本例中，指定了檔案模式為 `100644`，表明這是一個普通檔。其他可用的模式有：`100755` 表示可執行檔，`120000` 表示符號連結(symbolic link)。檔案模式是從一般的 UNIX 檔案模式中參考來的，但是沒有那麼靈活——上述三種模式僅對 Git 中的檔案 (blobs) 有效 (雖然也有其他模式用於目錄和子模組)。

現在可以用 `write-tree` 命令將暫存區域的內容寫到一個 `tree` 物件了。無需 `-w` 參數——如果目標 `tree` 不存在，呼叫 `write-tree` 會自動根據 `index` 狀態創建一個 `tree` 物件。

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

可以驗證這確實是一個 `tree` 物件：

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

再根據 `test.txt` 的第二個版本以及一個新檔創建一個新 `tree` 物件：

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

這時暫存區域中包含了 `test.txt` 的新版本及一個新檔 `new.txt`。創建(寫)該 `tree` 物件(將暫存區域或 `index` 狀態寫入到一個 `tree` 物件)，然後瞧瞧它的樣子：

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

請注意該 `tree` 物件包含了兩個檔案記錄，且 `test.txt` 的 SHA 值是早先值的“第二版”(`1f7a7a`)。來點更有趣的，你將把第一個 `tree` 物件作為一個子目錄加進該 `tree` 中。可以用 `read-tree` 命令將 `tree` 物件讀到暫存區域中去。在這時，通過傳一個 `--prefix` 參數給

`read-tree`，將一個已有的 tree 物件作為一個子 tree 讀到暫存區域中：

```
$ 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
```

如果從剛寫入的新 tree 物件創建一個工作目錄，將得到位於工作目錄頂級的兩個檔和一個名為 `bak` 的子目錄，該子目錄包含了 `test.txt` 檔的第一個版本。可以將 Git 用來包含這些內容的資料想像成如圖 9-2 所示的樣子。

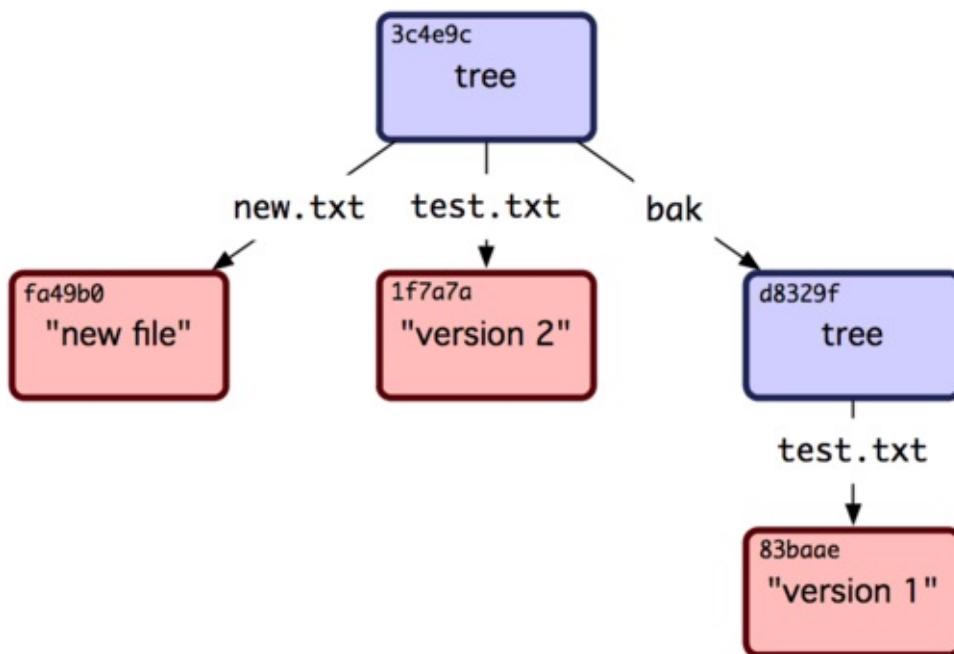


Figure 9-2. 當前 Git 資料的內容結構

Commit 物件

你現在有三個 tree 物件，它們指向下你要跟蹤的專案的不同快照，可是先前的問題依然存在：必須記住三個 SHA-1 值以獲得這些快照。你也沒有關於誰、何時以及為何保存了這些快照的資訊。`commit` 物件為你保存了這些基本資訊。

要創建一個 commit 物件，使用 `commit-tree` 命令，指定一個 tree 的 SHA-1，如果有任何前繼提交物件，也可以指定。從你寫的第一個 tree 開始：

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

通過 `cat-file` 查看這個新 commit 物件：

```
$ 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
```

commit 物件格式很簡單：指明了該時間點專案快照的頂層樹物件、作者/提交者資訊（從 Git 組態設定的 `user.name` 和 `user.email` 中獲得）以及當前時間戳記、一個空行，以及提交注釋資訊。

接著再寫入另外兩個 commit 物件，每一個都指定其之前的那個 commit 物件：

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

每一個 commit 物件都指向了你創建的樹物件快照。出乎意料的是，現在已經有了真實的 Git 歷史了，所以如果執行 `git log` 命令並指定最後那個 commit 物件的 SHA-1 便可以查看歷史：

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

bak/test.txt |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)

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 deletions(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

test.txt |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

真棒。你剛剛通過使用低級操作而不是那些普通命令創建了一個 Git 歷史。這基本上就是執行 `git add` 和 `git commit` 命令時 Git 進行的工作 —— 保存修改了的檔案的 blob，更新索引，創建 tree 物件，最後創建 commit 物件，這些 commit 物件指向了頂層 tree 物件以及先前的 commit 物件。這三類 Git 物件 —— blob，tree 以及 commit —— 都各自以檔案的方式保存在 `.git/objects` 目錄下。以下所列是目前為止範例目錄的所有物件，每個物件後面的注釋裡標明了它們保存的內容：

```
$ 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
```

如果你按照以上描述進行了操作，可以得到如圖 9-3 所示的物件圖。

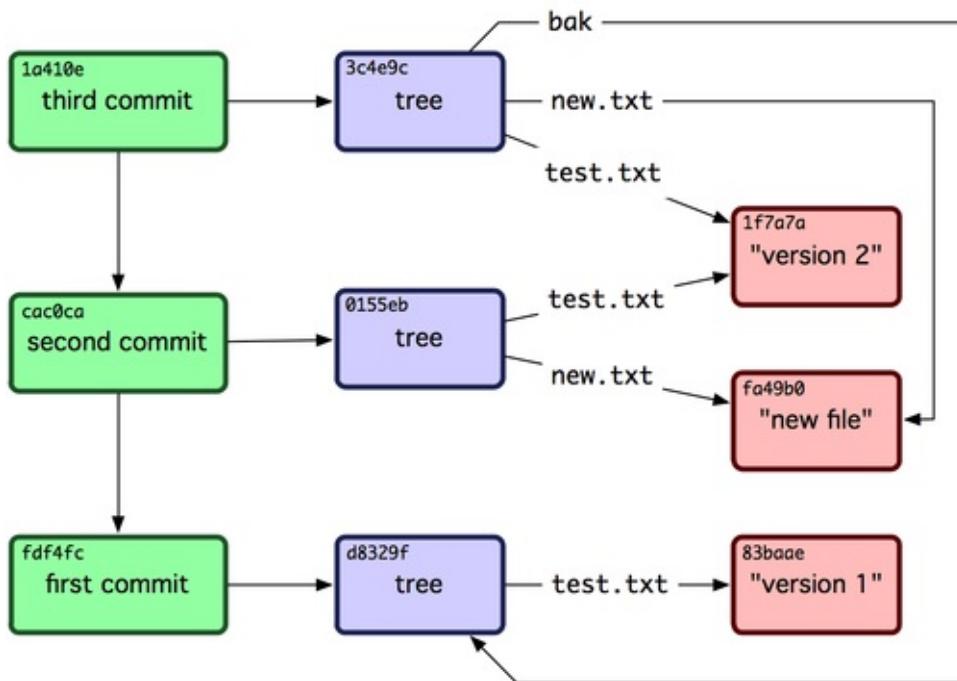


Figure 9-3. Git 目錄下的所有物件

物件儲存

之前我提到當儲存資料內容時，同時會有一個檔頭被儲存起來。我們花些時間來看看 Git 是如何儲存物件的。你將看到如何通過 Ruby 指令碼語言儲存一個 blob 物件（這裡以字串 “what is up, doc?” 為例）。使用 `irb` 命令進入 Ruby 互動式模式：

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git 以物件類型為起始內容構造一個檔頭，本例中是一個 blob。然後添加一個空格，接著是資料內容的長度，最後是一個空位元組 (null byte)：

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git 將檔頭與原始資料內容拼接起來，並計算拼接後的新內容的 SHA-1 校驗和。可以在 Ruby 中使用 `require` 語句導入 `SHA1 digest` 程式庫，然後呼叫 `Digest::SHA1hexdigest()` 方法計算字串的 SHA-1 值：

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git 用 zlib 對資料內容進行壓縮，在 Ruby 中可以用 zlib 程式庫來實現。首先需要導入該程式庫，然後用 `Zlib::Deflate.deflate()` 對資料進行壓縮：

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\3110R04c(\317H,Q\310,V(-\320QH\3110\266\000_\034\0235"
```

最後將用 zlib 壓縮後的內容寫入磁片。需要指定保存物件的路徑 (SHA-1 值的頭兩個字元作為子目錄名稱，剩餘 38 個字元作為檔案名保存至該子目錄中)。在 Ruby 中，如果子目錄不存在可以用 `FileUtils.mkdir_p()` 函數創建它。接著用 `File.open` 方法打開檔案，並用 `write()` 方法將之前壓縮的內容寫入該檔：

```
>> 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
```

這就行了 —— 你已經創建了一個正確的 blob 物件。所有的 Git 物件都以這種方式儲存，惟一的區別是類型不同 —— 除了字串 blob，檔頭起始內容還可以是 commit 或 tree。不過雖然 blob 幾乎可以是任意內容，commit 和 tree 的資料卻是有固定格式的。

Git References

你可以執行像 `git log 1a410e` 這樣的命令來查看完整的歷史，但是這樣你就要記得 `1a410e` 是你最後一次提交，這樣才能在提交歷史中找到這些物件。你需要一個檔來用一個簡單的名字來記錄這些 SHA-1 值，這樣你就可以用這些指標而不是原來的 SHA-1 值去檢索了。

在 Git 中，這些我們稱之為「引用」（references 或者 refs，譯者注）。你可以在 `.git/refs` 目錄下面找到這些包含 SHA-1 值的檔。在這個專案裡，這個目錄還沒不包含任何檔，但是包含這樣一個簡單的結構：

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

如果想要創建一個新的引用幫助你記住最後一次提交，技術上你可以這樣做：

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

現在，你就可以在 Git 命令中使用你剛才創建的引用而不是 SHA-1 值：

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769ccbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

當然，我們並不鼓勵你直接修改這些引用檔。如果你確實需要更新一個引用，Git 提供了一個比較安全的命令 `update-ref`：

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

基本上 Git 中的一個分支其實就是一個指向某個工作版本一條 HEAD 記錄的指標或引用。你可以用這條命令創建一個指向第二次提交的分支：

```
$ git update-ref refs/heads/test cac0ca
```

這樣你的分支將會只包含那次提交以及之前的工作：

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

現在，你的 Git 資料庫應該看起來像圖 9-4 一樣。

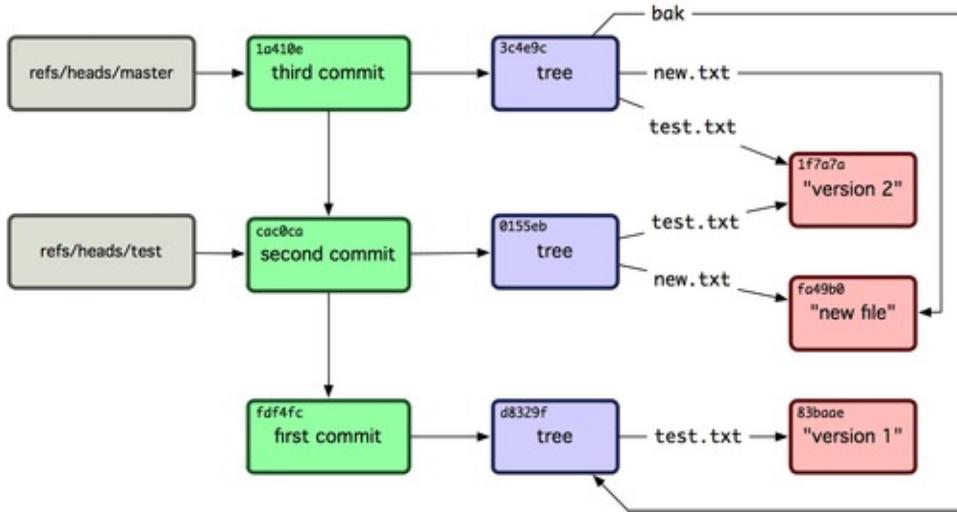


Figure 9-4. 包含分支引用的 Git 目錄物件

每當你執行 `git branch (分支名稱)` 這樣的命令，Git 基本上就是執行 `update-ref` 命令，把你現在所在分支中最後一次提交的 SHA-1 值，添加到你要創建的分支的引用。

HEAD 標記

現在的問題是，當你執行 `git branch (分支名稱)` 這條命令的時候，Git 怎麼知道最後一次提交的 SHA-1 值呢？答案就是 `HEAD` 檔。`HEAD` 檔是一個指向你當前所在分支的引用識別字。這樣的引用識別字——它看起來並不像一個普通的引用——其實並不包含 SHA-1 值，而是一個指向另外一個引用的指標。如果你看一下這個檔，通常你將會看到這樣的內容：

```
$ cat .git/HEAD
ref: refs/heads/master
```

如果你執行 `git checkout test`，Git 就會更新這個檔，看起來像這樣：

```
$ cat .git/HEAD
ref: refs/heads/test
```

當你再執行 `git commit` 命令，它就創建了一個 `commit` 物件，把這個 `commit` 物件的父級設置為 `HEAD` 指向的引用的 SHA-1 值。

你也可以手動編輯這個檔，但是同樣有一個更安全的方法可以這樣做：`symbolic-ref`。你可以用下面這條命令讀取 HEAD 的值：

```
$ git symbolic-ref HEAD
refs/heads/master
```

你也可以設置 HEAD 的值：

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

但是你不能設置成 `refs` 以外的形式：

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

你剛剛已經重溫過了 Git 的三個主要物件類型，現在這是第四種。Tag 物件非常像一個 commit 物件——包含一個標籤，一組資料，一個消息和一個指標。最主要的區別就是 Tag 物件指向一個 commit 而不是一個 tree。它就像是一個分支引用，但是不會變化——永遠指向同一個 commit，僅僅是提供一個更加友好的名字。

正如我們在第二章所討論的，Tag 有兩種類型：`annotated` 和 `lightweight`。你可以類似下面這樣的命令建立一個 `lightweight tag`：

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

這就是 `lightweight tag` 的全部——一個永遠不會發生變化的分支。`annotated tag` 要更複雜一點。如果你創建一個 `annotated tag`，Git 會創建一個 tag 物件，然後寫入一個 `reference` 指向這個 tag，而不是直接指向 commit。你可以這樣創建一個 `annotated tag`（`-a` 參數表明這是一個 `annotated tag`）：

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

這是所創建物件的 `SHA-1` 值：

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

現在你可以執行 `cat-file` 命令檢查這個 SHA-1 值：

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cf9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

值得注意的是這個物件指向你所標記的 `commit` 物件的 SHA-1 值。同時需要注意的是它並不是必須要指向一個 `commit` 物件；你可以標記任何 Git 物件。例如，在 Git 的原始程式碼裡，管理者添加了一個 GPG 公開金鑰（這是一個 `blob` 物件）對它做了一個標籤。你可以執行以下命令來查看

```
$ git cat-file blob junio-gpg-pub
```

Git 原始程式碼裡的公開金鑰. *Linux kernel* 也有一個不是指向 `commit` 物件的 `tag`——第一個 `tag` 是在導入原始程式碼的時候創建的，它指向初始 `tree`（`initial tree`，譯者注）。

Remotes

你將會看到的第三種 `reference` 是 `remote reference`（遠端參照，譯者注）。如果你添加了一個 `remote` 然後推送代碼過去，Git 會把你最後一次推送到這個 `remote` 的每個分支的值都記錄在 `refs/remotes` 目錄下。例如，你可以添加一個叫做 `origin` 的 `remote` 然後把你的 `master` 分支推送上去：

```
$ 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
```

然後查看 `refs/remotes/origin/master` 這個檔，你就會發現 `origin` `remote` 中的 `master` 分支就是你最後一次和伺服器的通信。

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references 和分支(refs/heads references)的主要區別在於他們是不能被 check out 的。Git 把他們當作是標記了這些分支在伺服器上最後狀態的一種書簽。

Packfiles

我們再來看一下 test Git 倉庫。目前為止，有 11 個物件 —— 4 個 blob，3 個 tree，3 個 commit 以及一個 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 用 zlib 壓縮檔案內容，因此這些檔並沒有佔用太多空間，所有檔加起來總共僅用了 925 位元組。接下去你將添加一些大檔以演示 Git 的一個很有意思的功能。將你之前用到過的 Grit 庫中的 repo.rb 檔加進去 —— 這個原始程式碼檔大小約為 12K：

```
$ curl https://raw.github.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 459 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

如果查看一下生成的 tree，可以看到 repo.rb 檔的 blob 物件的 SHA-1 值：

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dc51b4ac296e3e5b6e2a99cf44391e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

然後可以用 `git cat-file` 命令查看這個物件有多大：

```
$ du -b .git/objects/9b/c1dc421dc51b4ac296e3e5b6e2a99cf44391e
4102    .git/objects/9b/c1dc421dc51b4ac296e3e5b6e2a99cf44391e
```

稍微修改一下些檔，看會發生些什麼：

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
 1 files changed, 1 insertions(+), 0 deletions(-)
```

查看這個 commit 生成的 tree，可以看到一些有趣的東西：

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

blob 物件與之前的已經不同了。這說明雖然只是往一個 400 行的檔最後加入了一行內容，Git 却用一個全新的物件來保存新的檔案內容：

```
$ du -b .git/objects/05/408d195263d853f09dca71d55116663690c27c
4109   .git/objects/05/408d195263d853f09dca71d55116663690c27c
```

你的磁片上有了兩個幾乎完全相同的 12K 的物件。如果 Git 只完整保存其中一個，並保存另一個物件的差異內容，豈不更好？

事實上 Git 可以那樣做。Git 往磁片保存物件時預設使用的格式叫鬆散物件 (loose object) 格式。Git 時不時地將這些物件打包至一個叫 packfile 的二進位檔案以節省空間並提高效率。當倉庫中有太多的鬆散物件，或是手動執行 `git gc` 命令，或推送至遠端伺服器時，Git 都會這樣做。手動執行 `git gc` 命令讓 Git 將倉庫中的物件打包，並看看會發生些什麼：

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

查看一下 `objects` 目錄，會發現大部分物件都不在了，與此同時出現了兩個新檔：

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

仍保留著的幾個物件是未被任何 commit 引用的 blob — 在此例中是你之前創建的 “what is up, doc?” 和 “test content” 這兩個示例 blob。你從沒將他們添加至任何 commit，所以 Git 認為它們是懸而未決的，不會將它們打包進 packfile。

剩下的檔是新創建的 packfile 以及一個索引。packfile 檔包含了剛才從檔案系統中移除的所有物件。索引檔包含了 packfile 的偏移資訊(offset)，這樣就可以快速定位任意一個指定物件。有意思的是執行 `gc` 命令前磁片上的物件大小約為 8K，而這個新生成的 packfile 僅為 4K 大小。通過打包物件減少了一半磁片使用空間。

Git 是如何做到這點的？Git 打包物件時，會查找命名及尺寸相近的檔，並只保存檔案不同版本之間的差異內容。可以查看一下 packfile，觀察它是如何節省空間的。`git verify-pack` 命令用於顯示已打包的內容：

```
$ git verify-pack -v \
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 5400
05408d195263d853f09dca71d55116663690c27c blob 12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree 106 107 5086
1a410efbd13591db07496601ebc7a059dd55cf9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 101 105 5211
484a59275031909e19adb7c92262719cfcd919a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag 136 127 5476
9bc1dc421dc51b4ac296e3e5b6e2a99cf44391e blob 7 18 5193 1 \
05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob 1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree 106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

如果你還記得的話，`9bc1d` 這個 blob 是 `repo.rb` 檔的第一個版本，這個 blob 引用了 `05408` 這個 blob，即該檔的第二個版本。命令輸出內容的第三欄顯示的是物件大小，可以看到 `05408` 佔用了 12K 空間，而 `9bc1d` 僅為 7 位元組。非常有趣的是第二個版本才是完整保存檔案內容的物件，而第一個版本是以差異方式保存的 —— 這是因為大部分情況下需要快速訪問的是檔案的最新版本。

最妙的是可以隨時進行重新封包。Git 自動定期對倉庫進行重新封包以節省空間。當然也可以手動執行 `git gc` 命令來這麼做。

The Refspec

這本書讀到這裡，你已經使用過一些簡單的遠端分支到本地引用的映射方式了，這種映射可以更為複雜。假設你像這樣添加了一項遠端倉庫：

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

它在你的 `.git/config` 檔中添加了一節，指定了遠程的名稱 (`origin`)，遠程倉庫的 URL 地址，和用於獲取(`fetch`)操作的 Refspec:

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Refspec 的格式是一個可選的 `+` 號，接著是 `<src>:<dst>` 的格式，這裡 `<src>` 是遠端上的引用格式，`<dst>` 是將要記錄在本地的引用格式。可選的 `+` 號告訴 Git 在即使不能快速演進(`fast-forward`)的情況下，也去強制更新它。

預設情況下 `refspec` 會被 `git remote add` 命令所自動產生，Git 會獲取遠端伺服器上 `refs/heads/` 下面的所有引用，並將它寫入到本地的 `refs/remotes/origin/`。所以，如果遠端伺服器上有一個 `master` 分支，你在本地可以通過下面這種方式來取得它的歷史記錄：

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

它們的作用都是相同的，因為 Git 把它們都擴展成 `refs/remotes/origin/master`。

如果你想讓 Git 每次只拉取遠端的 `master` 分支，而不是遠端的所有分支，你可以把 `fetch` 這一行修改成這樣：

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

這是 `git fetch` 操作對這個遠端的預設 `refspec` 值。而如果你只想做一次該操作，也可以在命令列上指定這個 `refspec`。例如可以這樣拉取遠端的 `master` 分支到本地的 `origin/mymaster` 分支：

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

你也可以在命令列上指定多個 refspec. 像這樣可以一次獲取遠端的多個分支：

```
$ 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
```

在這個例子中， master 分支因為不是一個可以快速演進的引用而拉取操作被拒絕。你可以在 refspec 之前使用一個 + 號來 override 這種行為。

你也可以在設定檔中指定多個 refspec. 如你想在每次獲取時都獲取 master 和 experiment 分支，就添加兩行：

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

但是這裡不能使用部分萬用字元，像這樣就是不合法的：

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

但是你可以使用命名空間來達到這個目的。如果你有一個 QA 團隊，他們推送一系列分支，你想每次獲取 master 分支和 QA 團隊的所有分支，你可以使用這樣的配置段落(config section)：

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

如果你的工作流程很複雜，有QA團隊推送的分支、開發人員推送的分支、和集成人員推送的分支，並且他們在遠端分支上協作，你可以採用這種方式為他們創建各自的命名空間。

推送 Refspecs

採用命名空間的方式確實很棒，但QA團隊第一次是如何將他們的分支推送到 qa/ 空間裡面的呢？答案是你可以使用 refspec 來推送。

如果QA團隊想把他們的 master 分支推送到遠端的 qa/master 分支上，可以這樣執行：

```
$ git push origin master:refs/heads/qa/master
```

如果他們想讓 Git 每次運行 `git push origin` 時都這樣自動推送，他們可以在設定檔中添加 `push` 值：

```
[remote "origin"]
url = git@github.com:schacon/simplegit-progit.git
fetch = +refs/heads/*:refs/remotes/origin/*
push = refs/heads/master:refs/heads/qa/master
```

這樣，就會讓 `git push origin` 預設就把本地的 `master` 分支推送到遠端的 `qa/master` 分支上。

刪除 References

你也可以使用 `refspec` 來刪除遠端的引用(references)，是通過執行這樣的命令：

```
$ git push origin :topic
```

因為 `refspec` 的格式是 `<src>:<dst>`，通過把 `<src>` 部分留空的方式，這個意思是是把遠端的 `topic` 分支變成空，也就是刪除它。

傳輸協議

Git 可以用兩種主要的方式跨越兩個倉庫傳輸資料：基於 HTTP 協定之上，和 `file://`，`ssh://`，和 `git://` 等智慧傳輸協議。這一節帶你快速流覽這兩種主要的協議操作過程。

啞協議

Git 基於 HTTP 之上傳輸通常被稱為啞協議，這是因為它在服務端不需要有針對 Git 特有的代碼。這個獲取過程僅僅是一系列 GET 請求，用戶端可以假定服務端的 Git 倉庫中的佈局。讓我們以 `simplegit` 倉庫為例來看看 `http-fetch` 的過程：

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

它做的第一件事情就是獲取 `info/refs` 檔。這個檔是在服務端運行了 `update-server-info` 所產生的，這也解釋了為什麼在服務端要想使用 HTTP 傳輸，必須要開啓 `post-receive` 鈎子(hook)：

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

現在你有一個遠端引用和 SHA 值的列表。下一步是尋找 HEAD 引用，這樣你就知道了在完成後，什麼應該被檢出到工作目錄：

```
=> GET HEAD
ref: refs/heads/master
```

這說明在完成獲取後，需要檢出(check out) `master` 分支。這時，已經可以開始漫遊操作(walking process)了。因為你的起點是在 `info/refs` 檔中所提到的 `ca82a6` commit 物件，你的開始操作就是獲取它：

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

然後你收回了這個物件 — 這在服務端是一個鬆散格式的物件，你使用的是靜態的 HTTP GET 請求獲取的。可以使用 `zlib` 解壓縮它，去除檔頭，查看它的 commmit 內容：

```
$ 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
```

這樣，就得到了兩個需要進一步獲取的物件 — `cfda3b` 是這個 `commit` 物件所對應的 `tree` 物件，和 `085bb3` 是它的父物件：

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

這樣就取得了這它的下一步 `commit` 物件，再抓取 `tree` 物件：

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

哎呀！- 看起來這個 `tree` 物件在服務端並不以鬆散格式對象存在，所以得到了404回應，代表在 HTTP 服務端沒有找到該物件。這有好幾個原因 — 這個物件可能在替代倉庫裡面，或者在打包檔裡面，Git 會首先檢查任何列出的替代倉庫：

```
=> GET objects/info/http-alternates
(empty file)
```

如果這回傳了幾個替代倉庫列表，那麼它會去那些地方檢查鬆散格式物件和檔案 — 這是一種在軟體分叉(forks)之間共用物件以節省磁碟的好方法。然而，在這個例子中，沒有替代倉庫。所以你所需要的物件肯定在某個打包檔中。要檢查服務端有哪些打包格式檔，你需要獲取 `objects/info/packs` 檔，這裡面包含有打包檔列表（是的，它也是被 `update-server-info` 所產生的）：

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

這裡服務端只有一個打包檔，所以你要的物件顯然就在裡面。但是你可以先檢查它的索引檔以確認。這在服務端有多個打包檔時也很有用，因為這樣就可以先檢查你所需要的物件空間是在哪一個打包檔裡面了：

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

現在你有了這個打包檔的索引，你可以看看你要的物件是否在裡面 — 因為索引檔列出了這個打包檔所包含的所有物件的SHA值，和該物件存在於打包檔中的偏移量，所以你只需要簡單地獲取整個打包檔：

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

現在你也有了這個 tree 物件，你可以繼續在 commit 物件上漫遊。它們全部都在這個你已經下載到的打包檔裡面，所以你不用繼續向服務端請求更多下載了。在這完成之後，由於下載開始時已探明 HEAD 引用是指向 master 分支，Git 會將它檢出到工作目錄。

整個過程看起來就像這樣：

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

智慧協議

HTTP 方法是很簡單但效率不是很高。使用智慧協定是傳送資料的更常用的方法。這些協定在遠端都有 Git 智慧型程序(process)在服務 — 它可以讀出本地資料並計算出用戶端所需要的，並產生合適的資料給它，這有兩類傳輸資料的程序：一對用於上傳資料和一對用於下載。

上傳資料

為了上傳資料至遠端，Git 使用 send-pack 和 receive-pack 程序。這個 send-pack 程序運行在用戶端上，它連接至遠端運行的 receive-pack 程序。

舉例來說，你在你的專案上執行了 git push origin master，並且 origin 被定義為一個使用 SSH 協議的 URL。Git 會使用 send-pack 程序，它會啓動一個基於 SSH 的連接(connection)到伺服器。它嘗試像這樣透過 SSH 在服務端運行命令：

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"  
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs  
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic  
0000
```

這裡的 `git-receive-pack` 命令會立即對它所擁有的每一個引用回應一行 — 在這個例子中，只有 `master` 分支和它的SHA值。這裡第1行也包含了服務端的能力清單（這裡是 `report-status` 和 `delete-refs` ）。

每一行以4位元組的十六進位開始，用於指定整行的長度。你看到第1行以005b開始，這在十六進位中表示91，意味著第1行有91位元組長。下一行以003e起始，表示有62位元組長，所以需要讀剩下的62位元組。再下一行是0000開始，表示伺服器已完成了引用列表過程。

現在它知道了服務端的狀態，你的 `send-pack` 程序會判斷哪些 `commit` 是它所擁有但服務端沒有的。針對每個引用，這次推送都會告訴對端的 `receive-pack` 這個資訊。舉例說，如果你在更新 `master` 分支，並且增加 `experiment` 分支，這個 `send-pack` 將會是像這樣：

這裡全部是'0'的SHA-1值表示之前沒有遇這個物件 — 因為你是在添加新的 experiment 引用。如果你在刪除一個引用，你會看到相反的：就是右邊全部是'0'。

Git 對每個引用發送這樣一行資訊，就是舊的SHA值，新的SHA值，和將要更新的引用的名稱。第1行還會包含有用戶端的能力。下一步，用戶端會發送一個所有那些服務端所沒有的物件的一個打包檔。最後，服務端以成功(或者失敗)來回應：

000Unpack ok

下載資料

當你在下載資料時，`fetch-pack` 和 `upload-pack` 程序就起作用了。用戶端啓動 `fetch-pack` 程序，連接至遠端的 `upload-pack` 程序，以協商後續資料傳輸過程。

在遠端倉庫有不同的方式啓動 `upload-pack` 程序。你可以使用與 `receive-pack` 相同的透過 SSH 管道的方式，也可以通過 Git 後臺來啓動這個進程，它預設監聽在 9418 號埠上。這裡 `fetch-pack` 程序在連接後像這樣向後臺發送資料：

003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0

它也是以4位元組指定後續位元組長度的方式開始，然後是要執行的命令，和一個空位元組，然後是服務端的主機名稱，再跟隨一個最後的空位元組。Git 後臺程序會檢查這個命令是否可以執行，以及那個倉庫是否存在，以及是否具有公開許可權。如果所有檢查都通過了，它會啓動這個 `upload-pack` 程序並將用戶端的請求移交給它。

如果你透過 SSH 使用獲取(fetch)功能，`fetch-pack` 會像這樣運行：

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

不管哪種方式，在 `fetch-pack` 連接之後，`upload-pack` 都會以這種形式回傳：

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

這與 `receive-pack` 回應很類似，但是這裡指的能力是不同的。而且它還會指出 HEAD 引用，讓用戶端可以檢查是否是一份 clone。

在這裡，`fetch-pack` 程序檢查它自己所擁有的物件和所有它需要的物件，通過發送“want”和所需物件的 SHA 值，發送“have”和所有它已擁有的物件的 SHA 值。在列表完成時，再發送“done”通知 `upload-pack` 程序開始發送所需物件的打包檔。這個過程看起來像這樣：

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

這是傳輸協議的一個很基礎的例子，在更複雜的例子中，用戶端可能會支援 `multi_ack` 或者 `side-band` 能力；但是這個例子中展示了智慧協議的基本交互過程。

維護及資料復原

偶爾，你可能需要進行一些清理工作 —— 如減小一個倉庫的大小，清理導入的倉庫，或是恢復丟失的資料。本節將描述這類使用場景。

維護

Git 會不定時地自動執行稱為「auto gc」的命令。大部分情況下該命令什麼都不處理。不過要是存在太多鬆散物件 (loose object, 不在 packfile 中的物件) 或 packfile，Git 會執行 `git gc` 命令。`gc` 指垃圾收集 (garbage collect)，此命令會做很多工作：收集所有鬆散物件並將它們存入 packfile，合併這些 packfile 進一個大的 packfile，然後將不被任何 commit 引用並且已存在一段時間 (數月) 的物件刪除。

可以如下手動執行 auto gc 命令：

```
$ git gc --auto
```

再次強調，這個命令一般什麼都不幹。如果有 7,000 個左右的鬆散對象或是 50 個以上的 packfile，Git 才會真正觸發 `gc` 命令。你可以修改配置中的 `gc.auto` 和 `gc.autopacklimit` 來調整這兩個設定值。

`gc` 還會將所有引用 (references) 併入一個單獨檔。假設倉庫中包含以下分支和標籤：

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

這時如果執行 `git gc`，`refs` 下的所有檔都會消失。Git 會將這些檔挪到 `.git/packed-refs` 檔中去以提高效率，該檔是這個樣子的：

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cf9
```

當更新一個引用時，Git 不會修改這個檔，而是在 `refs/heads` 下寫入一個新檔。當查找一個引用的 SHA 時，Git 首先在 `refs` 目錄下查找，如果未找到則到 `packed-refs` 檔中去查找。因此如果在 `refs` 目錄下找不到一個引用，該引用可能存到 `packed-refs` 檔中去了。

請留意檔最後以 `\^` 開頭的那一行。這表示該行上一行的那個標籤是一個 `annotated` 標籤，而該行正是那個標籤所指向的 `commit`。

資料復原

在使用 Git 的過程中，有時會不小心丟失 `commit` 資訊。這一般出現在以下情況下：強制刪除了一個分支而後又想重新使用這個分支，`hard-reset` 了一個分支從而丟棄了分支的部分 `commit`。如果這真的發生了，有什麼辦法把丟失的 `commit` 找回來呢？

下面的例子演示了對 `test` 倉庫 `master` 分支進行 `hard-reset` 到一個老版本的 `commit` 的操作，然後恢復丟失的 `commit`。首先查看一下當前的倉庫狀態：

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbdbe608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

接著將 `master` 分支移回至中間的一個 `commit`：

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbdbe608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

這樣就丟棄了最新的兩個 `commit`——包含這兩個 `commit` 的分支不存在了。現在要做的是找出最新的那個 `commit` 的 SHA，然後添加一個指向它的分支。關鍵在於找出最新的 `commit` 的 SHA——你不大可能記住了這個 SHA，是吧？

通常最快捷的辦法是使用 `git reflog` 工具。當你（在一個倉庫下）工作時，Git 會在你每次修改了 `HEAD` 時悄悄地將改動記錄下來。當你提交或修改分支時，`reflog` 就會更新。`git update-ref` 命令也可以更新 `reflog`，這是在本章前面的“Git References”部分我們使用該命令而不是手工將 SHA 值寫入 `ref` 文件的理由。任何時間執行 `git reflog` 命令可以查看當前的狀態：

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

可以看到我們 `check out` 的兩個 `commit`，但沒有更多的相關資訊。執行 `git log -g` 會輸出 `reflog` 的正常日誌，從而顯示更多有用資訊：

```
$ 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 a bit
```

看起來弄丟了的 `commit` 是底下那個，這樣在那個 `commit` 上創建一個新分支就能把它恢復過來。比方說，可以在那個 `commit (ab1afef)` 上創建一個名為 `recover-branch` 的分支：

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769ccbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

酷！這樣有了一個跟原來 `master` 一樣的 `recover-branch` 分支，最新的兩個 `commit` 又找回來了。接著，假設引起 `commit` 丢失的原因並沒有記錄在 `reflog` 中——可以通過刪除 `recover-branch` 和 `reflog` 來類比這種情況。這樣最新的兩個 `commit` 不會被任何東西引用到：

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

因為 `reflog` 資料是保存在 `.git/logs/` 目錄下的，這樣就沒有 `reflog` 了。現在要怎樣恢復 `commit` 呢？辦法之一是使用 `git fsck` 工具，該工具會檢查倉庫的資料完整性。如果指定 `--full` 選項，該命令顯示所有未被其他物件引用（指向）的所有物件：

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afe80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

本例中，可以從 `dangling commit` 找到丟失了的 `commit`。用相同的方法就可以恢復它，即創建一個指向該 SHA 的分支。

移除物件

Git 有許多過人之處，不過有一個功能有時卻會帶來問題：`git clone` 會將包含每一個檔的所有歷史版本的整個專案下載下來。如果專案包含的僅僅是原始程式碼的話這並沒有什麼壞處，畢竟 Git 可以非常高效地壓縮此類資料。不過如果有人在某個時刻往專案中添加了一個非常大的檔，即便他在後來的提交中將此檔刪掉了，所有的簽出都會下載這個大檔。因為歷史記錄中引用了這個檔，它會一直存在著。

當你將 Subversion 或 Perforce 倉庫轉換導入至 Git 時這會成為一個很嚴重的問題。在此類系統中，(簽出時) 不會下載整個倉庫歷史，所以這種情形不大會有不良後果。如果你從其他系統導入了一個倉庫，或是發覺一個倉庫的尺寸遠超出預計，可以用下面的方法找到並移除大(尺寸) 物件。

警告：此方法會破壞提交歷史。為了移除對一個大檔的引用，從最早包含該引用的 `tree` 物件開始之後的所有 `commit` 物件都會被重寫。如果在剛導入一個倉庫並在其他人在此基礎上開始工作之前這麼做，那沒有什麼問題——否則你不得不通知所有協作者（貢獻者）去衍合你新修改的 `commit`。

為了演示這點，往 `test` 倉庫中加入一個大檔，然後在下次提交時將它刪除，接著找到並將這個檔從倉庫中永久刪除。首先，加一個大檔進去：

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

喔，你並不想往專案中加進一個這麼大的 `tar` 包。最後還是去掉它：

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

對倉庫進行 `gc` 操作，並查看佔用了空間：

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

可以執行 `count-objects` 以查看使用了多少空間：

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

`size-pack` 是以 KB 為單位表示的 `packfiles` 的大小，因此已經使用了 2MB。而在這次提交之前僅用了 2K 左右 — 顯然在這次提交時刪除檔並沒有真正將其從歷史記錄中刪除。每當有人複製這個倉庫去取得這個小專案時，都不得不複製所有 2MB 資料，而這僅僅因為你曾經不小心加了個大檔。讓我們來解決這個問題。

首先要找出這個檔。在本例中，你知道是哪個文件。假設你並不知道這一點，要如何找出哪個(些)文件佔用了這麼多的空間？如果執行 `git gc`，所有物件會存入一個 `packfile` 檔；執行另一個底層命令 `git verify-pack` 以識別出大物件，對輸出的第三欄資訊即檔案大小進行排序，還可以將輸出定向(pipe)到 `tail` 命令，因為你只關心排在最後的那幾個最大的檔：

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob    1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob    12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob    2056716 2056872 5401
```

最底下那個就是那個大檔：2MB。要查看這到底是哪個檔，可以使用第 7 章中已經簡單使用過的 `rev-list` 命令。若給 `rev-list` 命令傳入 `--objects` 選項，它會列出所有 commit SHA 值，blob SHA 值及相應的檔路徑。可以這樣查看 blob 的檔案名：

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

接下來要將該檔從歷史記錄的所有 `tree` 中移除。很容易找出哪些 commit 修改了這個檔：

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

必須重寫從 `6df76` 開始的所有 commit 才能將檔從 Git 歷史中完全移除。這麼做需要用到第 6 章中用過的 `filter-branch` 命令：

```
$ git filter-branch --index-filter \
  'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

`--index-filter` 選項類似於第 6 章中使用的 `--tree-filter` 選項，但這裡不是傳入一個命令去修改磁碟上 `checked out` 的檔，而是修改暫存區域或索引。不能用 `rm file` 命令來刪除一個特定檔，而是必須用 `git rm --cached` 來刪除它——也就是說，從索引而不是從磁碟上刪除它。這樣做是出於速度考慮——由於 Git 在執行你的 filter 之前無需將所有版本簽出到磁片上，這個操作會快得多。也可以用 `--tree-filter` 來完成相同的操作。`git rm` 的 `--ignore-unmatch` 選項指定當你試圖刪除的內容並不存在時不顯示錯誤。最後，因為你清楚問題是從哪個 commit 開始的，使用 `filter-branch` 重寫自 `6df7640` 這個 commit 開始的所有歷史記錄。不這麼做的話會重寫所有歷史記錄，花費不必要的更多時間。

現在歷史記錄中已經不包含對那個檔的引用了。不過 `reflog` 以及執行 `filter-branch` 時 Git 往 `.git/refs/original` 添加的一些 `refs` 中仍有對它的引用，因此需要將這些引用刪除並對倉庫進行 `repack` 操作。在進行 `repack` 前需要將所有對這些 commits 的引用去除：

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

看一下節省了多少空間。

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

repack 後倉庫的大小減小到了 7K，遠小於之前的 2MB。從 size 值可以看出大檔物件還在鬆散物件中，其實並沒有消失，不過這沒有關係，重要的是在再進行推送或複製，這個物件不會再傳送出去。如果真的要完全把這個物件刪除，可以運行 `git prune --expire` 命令。

總結

現在你應該對 Git 可以作什麼相當瞭解了，並且在一定程度上也知道了 Git 是如何實現的。本章涵蓋了許多 **plumbing** 命令——這些命令比較底層，且比你在本書其他部分學到的 **porcelain** 命令要來得簡單。從底層瞭解 Git 的工作原理可以幫助你更好地理解為何 Git 實現了目前的這些功能，也使你能夠針對你的工作流程寫出自己的工具和腳本。

Git 作為一套 **content-addressable** 的檔案系統，是一個非常強大的工具，而不僅僅只是一個 VCS。希望藉助於你新學到的 Git 內部原理的知識，你可以自己實做出有趣的應用，並以更進階的方式、更如魚得水的使用 Git。