45 個Git 經典操作場景，專治不會合代碼

架構師社區 2022-04-01 11:30

git對於大家應該都不太陌生，熟練使用git已經成為程序員的一項基本技能，儘管在工作中有諸如 `Sourcetree` 這樣牛X的客戶端工具，使得合併代碼變的很方便。但找工作面試和一些需彰顯個人實力的場景，仍然需要我們掌握足夠多的git命令。

下邊我們整理了45個日常用git合代碼的經典操作場景，基本覆蓋了工作中的需求。

我剛才提交了什麼?

如果你用 `git commit -a` 提交了一次變化(changes)，而你又不確定到底這次提交了哪些內容。你就可以用下面的命令顯示當前 `HEAD` 上的最近一次的提交(commit):

```
(main)$ git show
```

或者

```
$ git log -n1 -p
```

我的提交信息(commit message)寫錯了

如果你的提交信息(commit message)寫錯了且這次提交(commit)還沒有推(push)，你可以通過下面的方法來修改提交信息(commit message):

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

這會打開你的默認編輯器，在這裡你可以編輯信息．另一方面，你也可以用一條命令一次完成:

```
$ git commit --amend --only -m 'xxxxxxx'
```

如果你已經推(push)了這次提交(commit)，你可以修改這次提交(commit)然後強推(force push)，但是不推薦這麼做。

我提交(commit)裡的用戶名和郵箱不對

如果這只是單個提交(commit)，修改它：

```
$ git commit --amend --author "New Authorname <authoremail@mydomain.com>"
```

如果你需要修改所有歷史，參考'git filter-branch'的指南頁.

我想從一個提交(commit)裡移除一個文件

通過下面的方法，從一個提交(commit)裡移除一個文件:

```
$ git checkout HEAD^ myfile
$ git add -A
$ git commit --amend
```

這將非常有用，當你有一個開放的補丁(open patch)，你往上面提交了一個不必要的文件，你需要強推(force push)去更新這個遠程補丁。

我想刪除我的的最後一次提交(commit)

如果你需要刪除推了的提交(pushed commits)，你可以使用下面的方法。可是，這會不可逆的改變你的歷史，也會搞亂那些已經從該倉庫拉取(pulled)了的人的歷史。簡而言之，如果你不是很確定，千萬不要這麼做。

```
$ git reset HEAD^ --hard
$ git push -f [remote] [branch]
```

如果你還沒有推到遠程，把Git重置(reset)到你最後一次提交前的狀態就可以了(同時保存暫存的變化):

```
(my-branch*)$ git reset --soft HEAD@{1}
```

這只能在沒有推送之前有用. 如果你已經推了，唯一安全能做的是 `git revert SHAofBadCommit` ， 那會創建一個新的提交(commit)用於撤消前一個提交的所有變化(changes)；或者，如果你推的這個分支是rebase-safe的(例如：其它開發者不會從這個分支拉)，只需要使用 `git push -f` 。

刪除任意提交(commit)

同樣的警告：不到萬不得已的時候不要這麼做.

```
$ git rebase --onto SHA1_OF_BAD_COMMIT^ SHA1_OF_BAD_COMMIT
$ git push -f [remote] [branch]
```

或者做一個交互式rebase 刪除那些你想要刪除的提交(commit)裡所對應的行。

我嘗試推一個修正後的提交(amended commit)到遠程，但是報錯：

```
To https://github.com/yourusername/repo.git
! [rejected]        mybranch -> mybranch (non-fast-forward)
error: failed to push some refs to 'https://github.com/tanay1337/webmaker.org.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

注意，rebasing(見下面)和修正(amending)會用一個**新的提交(commit)代替舊的**，所以如果之前你已經往遠程倉庫上推過一次修正前的提交(commit)，那你現在就必須強推(force push) (`-f`)。注意– *總是* 確保你指明一個分支!

```
(my-branch)$ git push origin mybranch -f
```

一般來說，**要避免強推**. 最好是創建和推(push)一個新的提交(commit)，而不是強推一個修正後的提交。後者會使那些與該分支或該分支的子分支工作的開發者，在源歷史中產生衝突。

我意外的做了一次硬重置(hard reset)，我想找回我的內容

如果你意外的做了 `git reset --hard`，你通常能找回你的提交(commit)，因為Git對每件事都會有日誌，且都會保存幾天。

```
(main)$ git reflog
```

你將會看到一個你過去提交(commit)的列表，和一個重置的提交。選擇你想要回到的提交(commit)的SHA，再重置一次:

```
(main)$ git reset --hard SHA1234
```

這樣就完成了。

暫存(Staging)

我需要把暫存的內容添加到上一次的提交(commit)

```
(my-branch*)$ git commit --amend
```

我想要暫存一個新文件的一部分，而不是這個文件的全部

一般來說，如果你想暫存一個文件的一部分，你可這樣做:

```
$ git add --patch filename.x
```

`-p` 簡寫。這會打開交互模式， 你將能夠用 `s` 選項來分隔提交(commit)；然而，如果這個文件是新的，會沒有這個選擇， 添加一個新文件時，這樣做:

```
$ git add -N filename.x
```

然後，你需要用 `e` 選項來手動選擇需要添加的行，執行 `git diff --cached` 將會顯示哪些行暫存了哪些行只是保存在本地了。

我想把在一個文件裡的變化(changes)加到兩個提交(commit)裡

`git add` 會把整個文件加入到一個提交. `git add -p` 允許交互式的選擇你想要提交的部分.

我想把暫存的內容變成未暫存，把未暫存的內容暫存起來

多數情況下，你應該將所有的內容變為未暫存，然後再選擇你想要的內容進行commit。但假定你就是想要這麼做，這裡你可以創建一個臨時的commit來保存你已暫存的內容，然後暫存你的未暫存的內容並進行stash。然後reset最後一個commit將原本暫存的內容變為未暫存，最後stash pop回來。

```
$ git commit -m "WIP"
$ git add .
$ git stash
$ git reset HEAD^
$ git stash pop --index 0
```

注意1: 這裡使用 `pop` 僅僅是因為想盡可能保持冪等。注意2: 假如你不加上 `--index` 你會把暫存的文件標記為為存儲。

未暫存(Unstaged)的內容

我想把未暫存的內容移動到一個新分支

```
$ git checkout -b my-branch
```

我想把未暫存的內容移動到另一個已存在的分支

```
$ git stash
$ git checkout my-branch
$ git stash pop
```

我想丟棄本地未提交的變化(uncommitted changes)

如果你只是想重置源(origin)和你本地(local)之間的一些提交(commit)，你可以：

```
# one commit
(my-branch)$ git reset --hard HEAD^
# two commits
(my-branch)$ git reset --hard HEAD^^
# four commits
(my-branch)$ git reset --hard HEAD~4
# or
(main)$ git checkout -f
```

重置某個特殊的文件，你可以用文件名做為參數:

```
$ git reset filename
```

我想丟棄某些未暫存的內容

如果你想丟棄工作拷貝中的一部分內容，而不是全部。

簽出(checkout)不需要的內容，保留需要的。

```
$ git checkout -p
# Answer y to all of the snippets you want to drop
```

另外一個方法是使用 `stash`， Stash所有要保留下的內容，重置工作拷貝，重新應用保留的部分。

```
$ git stash -p
# Select all of the snippets you want to save
```

```
$ git reset --hard
$ git stash pop
```

或者，stash 你不需要的部分，然後stash drop。

```
$ git stash -p
# Select all of the snippets you don't want to save
$ git stash drop
```

分支(Branches)

我從錯誤的分支拉取了內容，或把內容拉取到了錯誤的分支

這是另外一種使用 `git reflog` 情況，找到在這次錯誤拉(pull) 之前HEAD的指向。

```
(main)$ git reflog
ab7555f HEAD@{0}: pull origin wrong-branch: Fast-forward
c5bc55a HEAD@{1}: checkout: checkout message goes here
```

重置分支到你所需的提交(desired commit):

```
$ git reset --hard c5bc55a
```

完成。

我想扔掉本地的提交(commit)，以便我的分支與遠程的保持一致

先確認你沒有推(push)你的內容到遠程。

`git status` 會顯示你領先(ahead)源(origin)多少個提交:

```
(my-branch)$ git status
# On branch my-branch
# Your branch is ahead of 'origin/my-branch' by 2 commits.
```

```
#   (use "git push" to publish your local commits)
#
```

一種方法是:

```
(main)$ git reset --hard origin/my-branch
```

我需要提交到一個新分支，但錯誤的提交到了main

在main下創建一個新分支，不切換到新分支,仍在main下:

```
(main)$ git branch my-branch
```

把main分支重置到前一個提交:

```
(main)$ git reset --hard HEAD^
```

`HEAD^` 是 `HEAD^1` 的簡寫，你可以通過指定要設置的`HEAD`來進一步重置。

或者，如果你不想使用 `HEAD^`，找到你想重置到的提交(commit)的hash(`git log` 能夠完成)， 然後重置到這個hash。使用`git push` 同步內容到遠程。

例如，main分支想重置到的提交的hash為`a13b85e`:

```
(main)$ git reset --hard a13b85e
HEAD is now at a13b85e
```

簽出(checkout)剛才新建的分支繼續工作:

```
(main)$ git checkout my-branch
```

我想保留來自另外一個ref-ish的整個文件

假設你正在做一個原型方案(原文為working spike (see note)), 有成百的內容，每個都工作得很好。現在, 你提交到了一個分支，保存工作內容:微信搜索公眾號：Java後端編程，回复：java 領取資料。

```
(solution)$ git add -A && git commit -m "Adding all changes from this spike into one big commi
```

當你想要把它放到一個分支裡(可能是 `feature`, 或者 `develop`), 你關心是保持整個文件的完整，你想要一個大的提交分隔成比較小。

假設你有:

- 分支 `solution`, 擁有原型方案， 領先 `develop` 分支。
- 分支 `develop`, 在這裡你應用原型方案的一些內容。

我去可以通過把內容拿到你的分支裡，來解決這個問題:

```
(develop)$ git checkout solution -- file1.txt
```

這會把這個文件內容從分支 `solution` 拿到分支 `develop` 裡來:

```
# On branch develop
# Your branch is up-to-date with 'origin/develop'.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   file1.txt
```

然後, 正常提交。

Note: Spike solutions are made to analyze or solve the problem. These solutions are used for estimation and discarded once everyone gets clear visualization of the problem.

我把幾個提交(commit)提交到了同一個分支，而這些提交應該分佈在不同的分支裡

假設你有一個 `main` 分支， 執行 `git log`，你看到你做過兩次提交:

```
(main)$ git log

commit e3851e817c451cc36f2e6f3049db528415e3c114
Author: Alex Lee <alexlee@example.com>
Date:   Tue Jul 22 15:39:27 2014 -0400

    Bug #21 - Added CSRF protection

commit 5ea51731d150f7ddc4a365437931cd8be3bf3131
Author: Alex Lee <alexlee@example.com>
Date:   Tue Jul 22 15:39:12 2014 -0400

    Bug #14 - Fixed spacing on title

commit a13b85e984171c6e2a1729bb061994525f626d14
Author: Aki Rose <akirose@example.com>
Date:   Tue Jul 21 01:12:48 2014 -0400

    First commit
```

讓我們用提交hash(commit hash)標記bug (`e3851e8` for #21, `5ea5173` for #14).

首先，我們把 `main` 分支重置到正確的提交(`a13b85e`):

```
(main)$ git reset --hard a13b85e
HEAD is now at a13b85e
```

現在，我們對bug #21 創建一個新的分支:

```
(main)$ git checkout -b 21
(21)$
```

接著，我們用 *cherry-pick* 把對bug #21的提交放入當前分支。這意味著我們將應用(apply)這個提交(commit)，僅僅這一個提交(commit)，直接在HEAD上面。

```
(21)$ git cherry-pick e3851e8
```

這時候，這裡可能會產生衝突， 參見交互式rebasing 章 **衝突節** 解決衝突.

再者， 我們為bug #14 創建一個新的分支，也基於 `main` 分支

```
(21)$ git checkout main
(main)$ git checkout -b 14
(14)$
```

最後，為bug #14 執行 `cherry-pick`:

```
(14)$ git cherry-pick 5ea5173
```

我想刪除上游(upstream)分支被刪除了的本地分支

一旦你在github 上面合併(merge)了一個pull request，你就可以刪除你fork裡被合併的分支。如果你不准備繼續在這個分支里工作，刪除這個分支的本地拷貝會更乾淨，使你不會陷入工作分支和一堆陳舊分支的混亂之中（IDEA 中玩轉Git）。

```
$ git fetch -p
```

我不小心刪除了我的分支

如果你定期推送到遠程，多數情況下應該是安全的，但有些時候還是可能刪除了還沒有推到遠程的分支。讓我們先創建一個分支和一個新的文件:

```
(main)$ git checkout -b my-branch
(my-branch)$ git branch
(my-branch)$ touch foo.txt
(my-branch)$ ls
README.md foo.txt
```

添加文件並做一次提交

```
(my-branch)$ git add .
(my-branch)$ git commit -m 'foo.txt added'
(my-branch)$ foo.txt added
 1 files changed, 1 insertions(+)
 create mode 100644 foo.txt
(my-branch)$ git log

commit 4e3cd85a670ced7cc17a2b5d8d3d809ac88d5012
Author: siemiatj <siemiatj@example.com>
Date:   Wed Jul 30 00:34:10 2014 +0200

    foo.txt added

commit 69204cdf0acbab201619d95ad8295928e7f411d5
Author: Kate Hudson <katehudson@example.com>
Date:   Tue Jul 29 13:14:46 2014 -0400

    Fixes #6: Force pushing after amending commits
```

現在我們切回到主(main)分支，'不小心的'刪除 `my-branch` 分支

```
(my-branch)$ git checkout main
Switched to branch 'main'
Your branch is up-to-date with 'origin/main'.
(main)$ git branch -D my-branch
Deleted branch my-branch (was 4e3cd85).
(main)$ echo oh noes, deleted my branch!
oh noes, deleted my branch!
```

在這時候你應該想起了 `reflog`，一個升級版的日誌，它存儲了倉庫(repo)裡面所有動作的歷史。

```
(main)$ git reflog
69204cd HEAD@{0}: checkout: moving from my-branch to main
4e3cd85 HEAD@{1}: commit: foo.txt added
69204cd HEAD@{2}: checkout: moving from main to my-branch
```

正如你所見，我們有一個來自刪除分支的提交hash(commit hash)，接下來看看是否能恢復刪除了的分支。

```
(main)$ git checkout -b my-branch-help
```

```
Switched to a new branch 'my-branch-help'
(my-branch-help)$ git reset --hard 4e3cd85
HEAD is now at 4e3cd85 foo.txt added
(my-branch-help)$ ls
README.md foo.txt
```

看！我們把刪除的文件找回來了。Git的 `reflog` 在rebasing出錯的時候也是同樣有用的。

我想刪除一個分支

刪除一個遠程分支:

```
(main)$ git push origin --delete my-branch
```

你也可以:

```
(main)$ git push origin :my-branch
```

刪除一個本地分支:

```
(main)$ git branch -D my-branch
```

我想從別人正在工作的遠程分支簽出(checkout)一個分支

首先，從遠程拉取(fetch) 所有分支:

```
(main)$ git fetch --all
```

假設你想要從遠程的 `daves` 分支簽出到本地的 `daves`

```
(main)$ git checkout --track origin/daves
Branch daves set up to track remote branch daves from origin.
```

```
Switched to a new branch 'daves'
```

(`--track` 是 `git checkout -b [branch] [remotename]/[branch]` 的簡寫)

這樣就得到了一個 `daves` 分支的本地拷貝，任何推過(pushed)的更新，遠程都能看到.

Rebasing 和合併(Merging)

我想撤銷rebase/merge

你可以合併(merge)或rebase了一個錯誤的分支，或者完成不了一個進行中的rebase/merge。Git 在進行危險操作的時候會把原始的HEAD保存在一個叫ORIG_HEAD的變量裡，所以要把分支恢復到rebase/merge前的狀態是很容易的。

```
(my-branch)$ git reset --hard ORIG_HEAD
```

我已經rebase過, 但是我不想強推(force push)

不幸的是，如果你想把這些變化(changes)反應到遠程分支上，你就必須得強推(force push)。是因你快進(Fast forward)了提交，改變了Git歷史，遠程分支不會接受變化(changes)，除非強推(force push)。這就是許多人使用merge 工作流，而不是rebasing工作流的主要原因之一， 開發者的強推(force push)會使大的團隊陷入麻煩。使用時需要注意，一種安全使用rebase 的方法是，不要把你的變化(changes)反映到遠程分支上，而是按下面的做:

```
(main)$ git checkout my-branch
(my-branch)$ git rebase -i main
(my-branch)$ git checkout main
(main)$ git merge --ff-only my-branch
```

我需要組合(combine)幾個提交(commit)

假設你的工作分支將會做對於 `main` 的pull-request。一般情況下你不關心提交(commit)的時間戳，只想組合 **所有** 提交(commit) 到一個單獨的里面，然後重置(reset)重提交(recommit)。確保主(main)分支是最新的和你的變化都已經提交了，然後:

```
(my-branch)$ git reset --soft main
(my-branch)$ git commit -am "New awesome feature"
```

如果你想要更多的控制，想要保留時間戳，你需要做交互式rebase (interactive rebase):

```
(my-branch)$ git rebase -i main
```

如果沒有相對的其它分支， 你將不得不相對自己的 `HEAD` 進行rebase。例如：你想組合最近的兩次提交(commit)，你將相對於 `HEAD~2` 進行rebase， 組合最近3次提交(commit)，相對於 `HEAD~3`，等等。

```
(main)$ git rebase -i HEAD~2
```

在你執行了交互式rebase的命令(interactive rebase command)後，你將在你的編輯器裡看到類似下面的內容:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
pick b729ad5 fixup
pick e3851e8 another fix

# Rebase 8074d12..b729ad5 onto 8074d12
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

所有以 `#` 開頭的行都是註釋，不會影響rebase.

然後，你可以用任何上面命令列表的命令替換 `pick`，你也可以通過刪除對應的行來刪除一個提交(commit)。

例如，如果你想 **單獨保留最舊(first)的提交(commit),組合所有剩下的到第二個里面**，你就應該編輯第二個提交(commit)後面的每個提交(commit) 前的單詞為 `f`:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
f b729ad5 fixup
f e3851e8 another fix
```

如果你想組合這些提交(commit) **並重命名這個提交(commit)**，你應該在第二個提交(commit)旁邊添加一個 `r`，或者更簡單的用 `s` 替代 `f`:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
s b729ad5 fixup
s e3851e8 another fix
```

你可以在接下來彈出的文本提示框裡重命名提交(commit)。

```
Newer, awesomer features

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# rebase in progress; onto 8074d12
# You are currently editing a commit while rebasing branch 'main' on '8074d12'.
#
# Changes to be committed:
#       modified:   README.md
#
```

如果成功了，你應該看到類似下面的內容:

```
(main)$ Successfully rebased and updated refs/heads/main.
```

安全合併(merging)策略

`--no-commit` 執行合併(merge)但不自動提交，給用戶在做提交前檢查和修改的機會。`no-ff` 會為特性分支(feature branch)的存在過留下證據，保持項目歷史一致（更多Git資料，參見IDEA 中如何完成Git 版本回退？）。

```
(main)$ git merge --no-ff --no-commit my-branch
```

我需要將一個分支合併成一個提交(commit)

```
(main)$ git merge --squash my-branch
```

我只想組合(combine)未推的提交(unpushed commit)

有時候，在將數據推向上游之前，你有幾個正在進行的工作提交(commit)。這時候不希望把已經推(push)過的組合進來，因為其他人可能已經有提交(commit)引用它們了。

```
(main)$ git rebase -i @{u}
```

這會產生一次交互式的rebase(interactive rebase)，只會列出沒有推(push)的提交(commit)， 在這個列表時進行reorder/fix/squash 都是安全的。

檢查是否分支上的所有提交(commit)都合併(merge)過了

檢查一個分支上的所有提交(commit)是否都已經合併(merge)到了其它分支，你應該在這些分支的head(或任何commits)之間做一次diff:

```
(main)$ git log --graph --left-right --cherry-pick --oneline HEAD...feature/120-on-scroll
```

這會告訴你在一個分支裡有而另一個分支沒有的所有提交(commit)，和分支之間不共享的提交(commit)的列表。另一個做法可以是:

```
```

```
(main)$ git log main ^feature/120-on-scroll --no-merges
```

交互式rebase(interactive rebase)可能出現的問題

這個rebase 編輯屏幕出現'noop'

如果你看到的是這樣:

```
noop
```

這意味著你rebase的分支和當前分支在同一個提交(commit)上，或者 *領先(ahead)* 當前分支。你可以嘗試:

- 檢查確保主(main)分支沒有問題
- rebase `HEAD~2` 或者更早

有衝突的情況

如果你不能成功的完成rebase，你可能必須要解決衝突。

首先执行 `git status` 找出哪些文件有冲突:

```
(my-branch)$ git status
On branch my-branch
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:   README.md
```

在这个例子里面，`README.md` 有冲突。打开这个文件找到类似下面的内容:

```
<<<<<<< HEAD
some code
=========
some code
>>>>>>> new-commit
```

```
>>>>>>> new-commit
```

你需要解决新提交的代码(示例里，从中间 `==` 线到 `new-commit` 的地方)与 `HEAD` 之间不一样的地方.

有时候这些合并非常复杂，你应该使用可视化的差异编辑器(visual diff editor):

```
(main*)$ git mergetool -t opendiff
```

在你解决完所有冲突和测试过后，`git add` 变化了的(changed)文件，然后用 `git rebase --continue` 继续rebase。

```
(my-branch)$ git add README.md
(my-branch)$ git rebase --continue
```

如果在解决完所有的冲突过后，得到了与提交前一样的结果，可以执行 `git rebase --skip` 。

任何时候你想结束整个rebase 过程，回来rebase前的分支状态，你可以做:

```
(my-branch)$ git rebase --abort
```

Stash

暂存所有改动

暂存你工作目录下的所有改动

```
$ git stash
```

你可以使用 `-u` 来排除一些文件

```
$ git stash -u
```

暂存指定文件

假设你只想暂存某一个文件

```
$ git stash push working-directory-path/filename.ext
```

假设你想暂存多个文件

```
$ git stash push working-directory-path/filename1.ext working-directory-path/filename2.ext
```

暂存时记录消息

这样你可以在 `list` 时看到它

```
$ git stash save <message>
```

或

```
$ git stash push -m <message>
```

使用某个指定暂存

首先你可以查看你的 `stash` 记录

```
$ git stash list
```

然后你可以 `apply` 某个 `stash`

```
$ git stash apply "stash@{n}"
```

此处， 'n'是 `stash` 在栈中的位置，最上层的 `stash` 会是0

除此之外，也可以使用时间标记(假如你能记得的话)。

```
$ git stash apply "stash@{2.hours.ago}"
```

暂存时保留未暂存的内容

你需要手动create一个 `stash commit` ， 然后使用 `git stash store` 。

```
$ git stash create
$ git stash store -m "commit-message" CREATED_SHA1
```

杂项(Miscellaneous Objects)

克隆所有子模块

```
$ git clone --recursive git://github.com/foo/bar.git
```

如果已经克隆了:

```
$ git submodule update --init --recursive
```

删除标签(tag)

```
$ git tag -d <tag_name>
$ git push <remote> :refs/tags/<tag_name>
```

恢复已删除标签(tag)

如果你想恢复一个已删除标签(tag)，可以按照下面的步骤：首先，需要找到无法访问的标签(unreachable tag):

```
$ git fsck --unreachable | grep tag
```

记下这个标签(tag)的hash，然后用Git的 update-ref

```
$ git update-ref refs/tags/<tag_name> <hash>
```

这时你的标签(tag)应该已经恢复了。

已删除补丁(patch)

如果某人在 GitHub 上给你发了一个pull request，但是然后他删除了他自己的原始fork，你将没法克隆他们的提交(commit)或使用 `git am` 。在这种情况下，最好手动的查看他们的提交(commit)，并把它们拷贝到一个本地新分支，然后做提交。

做完提交后，再修改作者，参见变更作者。然后，应用变化，再发起一个新的pull request。

跟踪文件(Tracking Files)

我只想改变一个文件名字的大小写，而不修改内容

```
(main)$ git mv --force myfile MyFile
```

我想从Git删除一个文件，但保留该文件

```
(main)$ git rm --cached log.txt
```

配置(Configuration)

我想给一些Git命令添加别名(alias)

在 OS X 和 Linux 下，你的 Git的配置文件储存在 `~/.gitconfig` 。我在 `[alias]` 部分添加了一些快捷别名(和一些我容易拼写错误的)，如下:

```
[alias]
    a = add
    amend = commit --amend
    c = commit
    ca = commit --amend
    ci = commit -a
    co = checkout
    d = diff
    dc = diff --changed
    ds = diff --staged
    f = fetch
    loll = log --graph --decorate --pretty=oneline --abbrev-commit
    m = merge
    one = log --pretty=oneline
    outstanding = rebase -i @{u}
    s = status
    unpushed = log @{u}
    wc = whatchanged
    wip = rebase -i @{u}
    zap = fetch -p
```

我想缓存一个仓库(repository)的用户名和密码

你可能有一个仓库需要授权，这时你可以缓存用户名和密码，而不用每次推/拉(push/pull)的时候都输入，Credential helper能帮你。

```
$ git config --global credential.helper cache
# Set git to use the credential memory cache
```

```
$ git config --global credential.helper 'cache --timeout=3600'
# Set the cache to timeout after 1 hour (setting is in seconds)
```

我不知道我做错了些什么

你把事情搞砸了：你 `重置(reset)` 了一些东西，或者你合并了错误的分支，亦或你强推了后找不到你自己的提交(commit)了。有些时候，你一直都做得很好，但你想回到以前的某个状态。

这就是 `git reflog` 的目的， `reflog` 记录对分支顶端(the tip of a branch)的任何改变，即使那个顶端没有被任何分支或标签引用。基本上，每次HEAD的改变，一条新的记录就会增加到 `reflog` 。遗憾的是，这只对本地分支起作用，且它只跟踪动作 (例如，不会跟踪一个没有被记录的文件的任何改变)。

```
(main)$ git reflog
0a2e358 HEAD@{0}: reset: moving to HEAD~2
0254ea7 HEAD@{1}: checkout: moving from 2.2 to main
c10f740 HEAD@{2}: checkout: moving from main to 2.2
```

上面的reflog展示了从main分支签出(checkout)到2.2 分支，然后再签回。那里，还有一个硬重置(hard reset)到一个较旧的提交。最新的动作出现在最上面以 `HEAD@{0}` 标识.

如果事实证明你不小心回移(move back)了提交(commit)，reflog 会包含你不小心回移前main上指向的提交(0254ea7)。

```
$ git reset --hard 0254ea7
```

然后使用git reset就可以把main改回到之前的commit，这提供了一个在历史被意外更改情况下的安全网。

文章转载于：江南一点雨

10个 Linux 命令，让你的操作更有效率

高效运维

当SQL注入遇到诡异的编码问题

HACK之道

技術大佬每次看新手寫代碼的樣子

程序員的幽默