45 个 Git 经典操作场景，专治不会合代码

Linux爱好者 2023-04-06 11:50 发表于浙江

> 作者：小富
> https://blog.csdn.net/xinzhifu1/article/details/123271097

git对于大家应该都不太陌生，熟练使用git已经成为程序员的一项基本技能，尽管在工作中有诸如 `Sourcetree` 这样牛X的客户端工具，使得合并代码变的很方便。但找工作面试和一些需彰显个人实力的场景，仍然需要我们掌握足够多的git命令。

下边我们整理了45个日常用git合代码的经典操作场景，基本覆盖了工作中的需求。

我刚才提交了什么?

如果你用 `git commit -a` 提交了一次变化(changes)，而你又不确定到底这次提交了哪些内容。你就可以用下面的命令显示当前 `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)写错了

如果你的提交信息(commit message)写错了且这次提交(commit)还没有推(push)，你可以通过下面的方法来修改提交信息(commit message):

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

这会打开你的默认编辑器，在这里你可以编辑信息．另一方面，你也可以用一条命令一次完成:

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

如果你已经推(push)了这次提交(commit)，你可以修改这次提交(commit)然后强推(force push)，但是不推荐这么做。

我提交(commit)里的用户名和邮箱不对

如果这只是单个提交(commit)，修改它:

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

如果你需要修改所有历史，参考 'git filter-branch'的指南页.

我想从一个提交(commit)里移除一个文件

通过下面的方法，从一个提交(commit)里移除一个文件:

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

这将非常有用，当你有一个开放的补丁(open patch)，你往上面提交了一个不必要的文件，你需要强推(force push)去更新这个远程补丁。

我想删除我的的最后一次提交(commit)

如果你需要删除推了的提交(pushed commits)，你可以使用下面的方法。可是，这会不可逆的改变你的历史，也会搞乱那些已经从该仓库拉取(pulled)了的人的历史。简而言之，如果你不是很确定，千万不要这么做。

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

如果你还没有推到远程，把Git重置(reset)到你最后一次提交前的状态就可以了(同时保存暂存的变化):

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

这只能在没有推送之前有用. 如果你已经推了，唯一安全能做的是 `git revert SHAofBadCommit` ， 那会创建一个新的提交(commit)用于撤消前一个提交的所有变化(changes)；或者，如果你推的这个分支是rebase-safe的（例如：其它开发者不会从这个分支拉)，只需要使用 `git push -f` 。

删除任意提交(commit)

同样的警告：不到万不得已的时候不要这么做.

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

或者做一个 交互式rebase 删除那些你想要删除的提交(commit)里所对应的行。

我尝试推一个修正后的提交(amended commit)到远程，但是报错：

```
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(见下面)和修正(amending)会用一个**新的提交(commit)代替旧的**，所以如果之前你已经往远程仓库上推过一次修正前的提交(commit)，那你现在就必须强推(force push)（`-f`）。注意 – *总是* 确保你指明一个分支!

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

一般来说，**要避免强推**．最好是创建和推(push)一个新的提交(commit)，而不是强推一个修正后的提交。后者会使那些与该分支或该分支的子分支工作的开发者，在源历史中产生冲突。

我意外的做了一次硬重置(hard reset)，我想找回我的内容

如果你意外的做了 `git reset --hard`，你通常能找回你的提交(commit)，因为Git对每件事都会有日志，且都会保存几天。

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

你将会看到一个你过去提交(commit)的列表，和一个重置的提交。选择你想要回到的提交(commit)的SHA，再重置一次:

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

这样就完成了。

暂存(Staging)

我需要把暂存的内容添加到上一次的提交(commit)

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

我想要暂存一个新文件的一部分，而不是这个文件的全部

一般来说，如果你想暂存一个文件的一部分，你可这样做:

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

`-p` 简写。这会打开交互模式， 你将能够用 `s` 选项来分隔提交(commit)；然而，如果这个文件是新的，会没有这个选择， 添加一个新文件时，这样做:

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

然后，你需要用 `e` 选项来手动选择需要添加的行，执行 `git diff --cached` 将会显示哪些行暂存了哪些行只是保存在本地了。

我想把在一个文件里的变化(changes)加到两个提交(commit)里

`git add` 会把整个文件加入到一个提交. `git add -p` 允许交互式的选择你想要提交的部分.

我想把暂存的内容变成未暂存，把未暂存的内容暂存起来

多数情况下，你应该将所有的内容变为未暂存，然后再选择你想要的内容进行commit。但假定你就是想要这么做，这里你可以创建一个临时的commit来保存你已暂存的内容，然后暂存你的未暂存的内容并进行stash。然后reset最后一个commit将原本暂存的内容变为未暂存，最后stash pop回来。

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

注意1：这里使用 `pop` 仅仅是因为想尽可能保持幂等。注意2：假如你不加上 `--index` 你会把暂存的文件标记为为存储。

未暂存(Unstaged)的内容

我想把未暂存的内容移动到一个新分支

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

我想把未暂存的内容移动到另一个已存在的分支

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

我想丢弃本地未提交的变化(uncommitted changes)

如果你只是想重置源(origin)和你本地(local)之间的一些提交(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
```

重置某个特殊的文件，你可以用文件名做为参数：

```
$ git reset filename
```

我想丢弃某些未暂存的内容

如果你想丢弃工作拷贝中的一部分内容，而不是全部。

签出(checkout)不需要的内容，保留需要的。

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

另外一个方法是使用 `stash`， Stash所有要保留下的内容，重置工作拷贝，重新应用保留的部分。

```
$ 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，然后stash drop。

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

分支(Branches)

我从错误的分支拉取了内容，或把内容拉取到了错误的分支

这是另外一种使用 `git reflog` 情况，找到在这次错误拉(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)，以便我的分支与远程的保持一致

先确认你没有推(push)你的内容到远程。

`git status` 会显示你领先(ahead)源(origin)多少个提交:

```
(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)
#
```

一种方法是:

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

我需要提交到一个新分支，但错误的提交到了main

在main下创建一个新分支，不切换到新分支,仍在main下:

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

把main分支重置到前一个提交:

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

`HEAD^` 是 `HEAD^1` 的简写，你可以通过指定要设置的 `HEAD` 来进一步重置。

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` 能够完成)， 然后重置到这个hash。使用 `git push` 同步内容到远程。

例如，main分支想重置到的提交的hash为 `a13b85e`：

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

签出(checkout)刚才新建的分支继续工作:

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

我想保留来自另外一个ref-ish的整个文件

假设你正在做一个原型方案(原文为working spike (see note))，有成百的内容，每个都工作得很好。现在，你提交到了一个分支，保存工作内容:

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

当你想要把它放到一个分支里（可能是 `feature`，或者 `develop`），你关心是保持整个文件的完整，你想要一个大的提交分隔成比较小。

假设你有:

- 分支 `solution`，拥有原型方案， 领先 `develop` 分支。
- 分支 `develop`，在这里你应用原型方案的一些内容。

我去可以通过把内容拿到你的分支里，来解决这个问题:

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

这会把这个文件内容从分支 `solution` 拿到分支 `develop` 里来:

```
# 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
```

然后，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.

我把几个提交(commit)提交到了同一个分支，而这些提交应该分布在不同的分支里

假设你有一个 `main` 分支， 执行 `git log`，你看到你做过两次提交:

```
(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
```

让我们用提交hash(commit hash)标记bug（`e3851e8` for #21, `5ea5173` for #14).

首先，我们把 `main` 分支重置到正确的提交(`a13b85e`):

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

现在，我们对 bug #21 创建一个新的分支:

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

接着，我们用 *cherry-pick* 把对bug #21的提交放入当前分支。这意味着我们将应用(apply)这个提交(commit)，仅仅这一个提交(commit)，直接在HEAD上面。

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

这时候，这里可能会产生冲突， 参见交互式 rebasing 章 **冲突节** 解决冲突.

再者， 我们为bug #14 创建一个新的分支，也基于 `main` 分支

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

最后，为 bug #14 执行 `cherry-pick`：

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

我想删除上游(upstream)分支被删除了的本地分支

一旦你在github 上面合并(merge)了一个pull request，你就可以删除你fork里被合并的分支。如果你不准备继续在这个分支里工作，删除这个分支的本地拷贝会更干净，使你不会陷入工作分支和一堆陈旧分支的混乱之中（IDEA 中玩转 Git）。

```
$ git fetch -p
```

我不小心删除了我的分支

如果你定期推送到远程，多数情况下应该是安全的，但有些时候还是可能删除了还没有推到远程的分支。让我们先创建一个分支和一个新的文件:

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

添加文件并做一次提交

```
(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
```

现在我们切回到主(main)分支， ‘不小心的’ 删除 `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!
```

在这时候你应该想起了 `reflog`，一个升级版的日志，它存储了仓库(repo)里面所有动作的历史。

```
(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
```

正如你所见，我们有一个来自删除分支的提交hash(commit hash)，接下来看看是否能恢复删除了的分支。

```
(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！我们把删除的文件找回来了。Git的 `reflog` 在rebasing出错的时候也是同样有用的。

我想删除一个分支

删除一个远程分支:

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

你也可以:

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

删除一个本地分支:

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

我想从别人正在工作的远程分支签出(checkout)一个分支

首先，从远程拉取(fetch) 所有分支:

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

假设你想要从远程的 `daves` 分支签出到本地的 `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]` 的简写)

这样就得到了一个 `daves` 分支的本地拷贝，任何推过(pushed)的更新，远程都能看到.

Rebasing 和合并(Merging)

我想撤销rebase/merge

你可以合并(merge)或rebase了一个错误的分支，或者完成不了一个进行中的rebase/merge。Git 在进行危险操作的时候会把原始的HEAD保存在一个叫ORIG_HEAD的变量里，所以要把分支恢复到rebase/merge前的状态是很容易的。

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

我已经rebase过，但是我不想强推(force push)

不幸的是，如果你想把这些变化(changes)反应到远程分支上，你就必须得强推(force push)。是因你快进(Fast forward)了提交，改变了Git历史，远程分支不会接受变化(changes)，除非强推(force push)。这就是许多人使用 merge 工作流，而不是 rebasing 工作流的主要原因之一， 开发者的强推(force push)会使大的团队陷入麻烦。使用时需要注意，一种安全使用 rebase 的方法是，不要把你的变化(changes)反映到远程分支上，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
```

我需要组合(combine)几个提交(commit)

假设你的工作分支将会做对于 `main` 的pull-request。一般情况下你不关心提交(commit)的时间戳，只想组合 **所有** 提交(commit) 到一个单独的里面，然后重置(reset)重提交(recommit)。确保主(main)分支是最新的和你的变化都已经提交了，然后:

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

如果你想要更多的控制，想要保留时间戳，你需要做交互式rebase (interactive rebase):

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

如果没有相对的其它分支， 你将不得不相对自己的 `HEAD` 进行 rebase。例如：你想组合最近的两次提交(commit)，你将相对于 `HEAD~2` 进行rebase， 组合最近3次提交(commit)，相对于 `HEAD~3` ，等等。

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

在你执行了交互式 rebase的命令(interactive rebase command)后，你将在你的编辑器里看到类似下面的内容:

```
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 `#` 开头的行都是注释，不会影响 rebase.

然后，你可以用任何上面命令列表的命令替换 `pick`，你也可以通过删除对应的行来删除一个提交(commit)。

例如，如果你想 **单独保留最旧(first)的提交(commit),组合所有剩下的到第二个里面**，你就应该编辑第二个提交(commit)后面的每个提交(commit) 前的单词为 `f`：

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

如果你想组合这些提交(commit) **并重命名这个提交(commit)**，你应该在第二个提交(commit)旁边添加一个 `r`，或者更简单的用 `s` 替代 `f`：

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

你可以在接下来弹出的文本提示框里重命名提交(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，你应该看到类似下面的内容:

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

安全合并(merging)策略

`--no-commit` 执行合并(merge)但不自动提交，给用户在做提交前检查和修改的机会。 `no-ff` 会为特性分支(feature branch)的存在过留下证据，保持项目历史一致（更多Git资料，参见IDEA 中如何完成 Git 版本回退？）。

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

我需要将一个分支合并成一个提交(commit)

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

我只想组合(combine)未推的提交(unpushed commit)

有时候，在将数据推向上游之前，你有几个正在进行的工作提交(commit)。这时候不希望把已经推(push)过的组合进来，因为其他人可能已经有提交(commit)引用它们了。

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

这会产生一次交互式的rebase(interactive rebase)，只会列出没有推(push)的提交(commit)， 在这个列表时进行reorder/fix/squash 都是安全的。

检查是否分支上的所有提交(commit)都合并(merge)过了

检查一个分支上的所有提交(commit)是否都已经合并(merge)到了其它分支，你应该在这些分支的head(或任何 commits)之间做一次diff:

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

这会告诉你在一个分支里有而另一个分支没有的所有提交(commit)，和分支之间不共享的提交(commit)的列表。另一个做法可以是:

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

交互式rebase(interactive rebase)可能出现的问题

这个rebase 编辑屏幕出现'noop'

如果你看到的是这样:

```
noop
```

这意味着你rebase的分支和当前分支在同一个提交(commit)上，或者 *领先（ahead）* 当前分支。你可以尝试:

- 检查确保主(main)分支没有问题
- rebase `HEAD~2` 或者更早

有冲突的情况

如果你不能成功的完成rebase，你可能必须要解决冲突。

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
```

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。

- EOF -

推荐阅读 — 点击标题可跳转

1、百度工程师浅谈分布式日志

2、ELF 文件、镜像（Image）文件、可执行文件、对象文件详解

3、三星被曝因 ChatGPT 泄露芯片机密！韩媒惊呼数据“原封不动”直传美国，软银已禁止员工使用