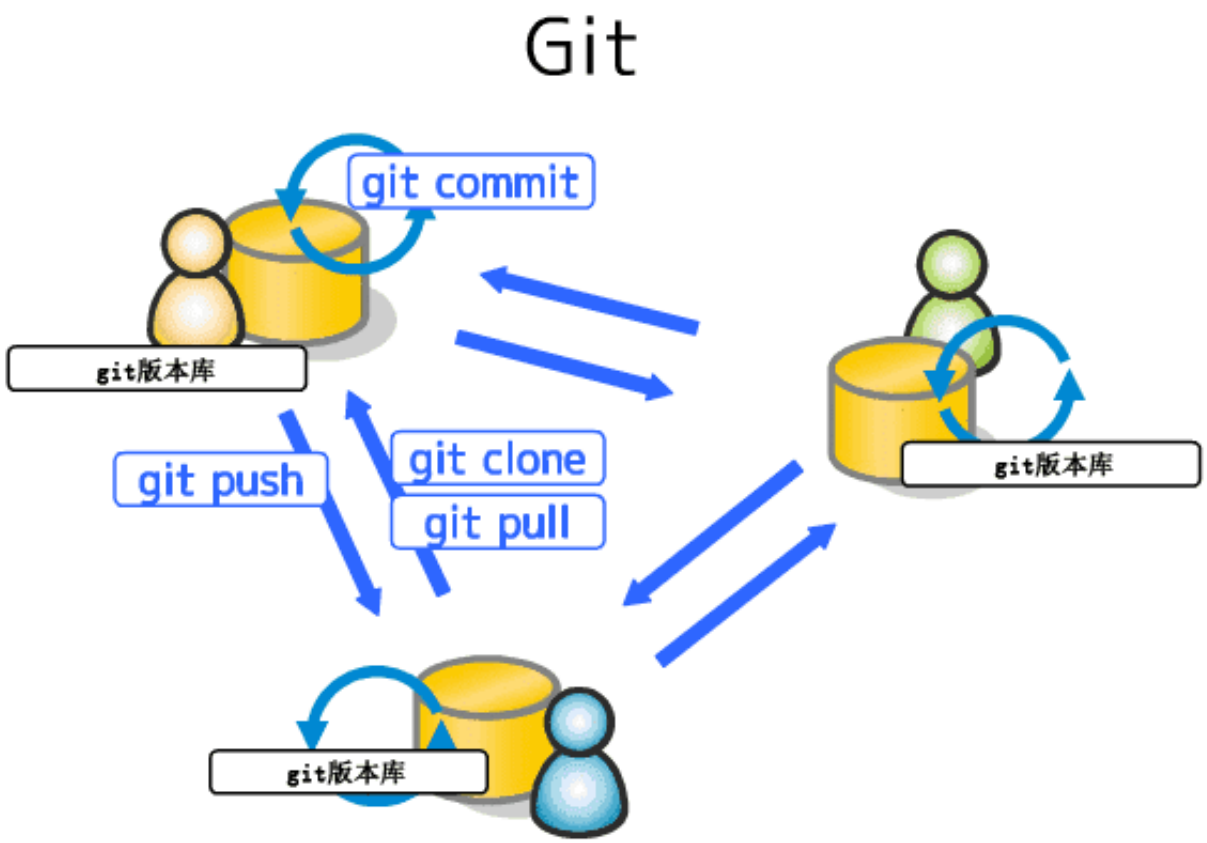
我有明珠一颗，久被尘劳关锁，今朝尘尽光生，照破山河万朵。 柴陵郁禅师（宋）

Git 中级用户的25个提示

PUBLISHED JULY 19TH 2015 BY QINGNIU

我使用 Git 大约已经有18个月时间，自认为能很好地驾驭它了。但是当我们请到 GitHub 的 Scott Chacon 来到 LVS 公司（一个博彩/游戏软件供应商/开发商）做专场培训时，我在第一天就学到了大量的东西。

由于有些人总是对使用 Git 自我感觉良好，因此，我想分享一些我从社区获取到的 Git 精品，这样就可能会帮助那些人无需浪费大量研究时间而直接找到答案。



基本提示

1. 安装之后第一步

安装 Git 之后，你要做的第一件事情就是去配置你的名字和邮箱，因为每一次提交都需要这些信息：

```
$ git config --global user.name "Some One"
$ git config --global user.email "someone@gmail.com"
```

2. Git 是以指针为基础

存储在 git 中的所有东西都包含在一个文件中。当你提交的时候，git 会创建一个包含提交消息和相关数据的文件（名称、邮件、日期/时间、上一次提交等等），并将其链接到一个树形文件。树形文件包含一个对象列表或者其它树。对象或二进制大数据对象（BLOB）是提交的真正内容（一个文件，如果你愿意，虽然文件名没有存储在

对象中，但会存储在树中）。所有这些文件都以对象的 SHA-1 哈希为文件名进行存储。

分支和标签只是一些文件，这些文件包含（基本上）一个指向提交的 SHA-1 哈希值。使用这些引用在灵活性和速度上均有大幅提升，创建一个新的分支就和创建一个文件一样简单，只是这个文件带有分支名称和一个包含指向提交（你从这个提交建立分支）的SHA-1哈希值。当然，当你使用 Git 命令行工具（或一个图形用户界面）时，你永远也不会这么做，但它就是这么简单。

你可能已经听说过对 HEAD 的引用。它只是一个包含 SHA-1 引用的文件，这个引用指向你当前的提交。如果你正在解决一个合并冲突问题，查看一下 HEAD，你会发现，它与一个特定的分支或分支上的特定点无关，只和你现在的位置有关。

所有的分支指针保存在 .git/refs/heads 目录下，HEAD 在 .git/HEAD 目录下，标签在 .git/refs/tags 目录下 - 你可以随意看看。

3. 两个 **Parents -** 当然！

当在日志文件中查看一个合并提交的消息时，你会看到两个 parents（与正常提交相比）。第一个parent 是你所在的分支，第二个 parents 是你并入的分支。

4. 合并冲突

到目前为止，我确信你一定有一个合并冲突需要解决。通常情况下，通过编辑该文件，删除文件中的**<<<<**, **====**, **>>>>**标记，然后保存你需要保留的代码就可以了。有时候，在任何变更之前查看代码都是一个值得推荐的做法，比如，在你对两个有冲突的分支采取行动之前。这是又一个命令：

```
$ git diff --merge
diff --cc dummy.rb
index 5175dde,0c65895..4a00477
--- a/dummy.rb
+++ b/dummy.rb
@@@ -1,5 -1,5 +1,5 @@@
  class MyFoo
    def say
-     puts "Bonjour"
```

```
 -    puts "Hello world"
++    puts "Annyong Haseyo"
    end
  end
```

如果文件是二进制的，文件比较就不是那么容易了...你通常要做的是尝试每个版本的二进制文件，并决定使用哪一个（或者在二进制文件编辑器手动复制部分内容）。从一个特定分支下 pull 一个文件副本（如果你要合并主分支和分支132的话）：

```
$ git checkout master flash/foo.fla # or...
$ git checkout feature132 flash/foo.fla
$ # Then...
$ git add flash/foo.fla
```

另一种方法是从 git 中查看这个文件 - 你能够以其他文件名的方式进行查看，然后将正确的文件（当你确定它是哪一个时）复制到正常的文件名中：

```
$ git show master:flash/foo.fla > master-foo.fla
$ git show feature132:flash/foo.fla > feature132-foo.fla
$ # Check out master-foo.fla and feature132-foo.fla
$ # Let's say we decide that feature132's is correct
$ rm flash/foo.fla
$ mv feature132-foo.fla flash/foo.fla
$ rm master-foo.fla
$ git add flash/foo.fla
```

更新：感谢 Carl 在早先的的博客评论中给与的提醒，你实际上能使用 “git checkout —ours flash/foo.fla” 和 “git checkout —theirs flash/foo.fla” 检出一个特定的版本而不需要记住你要合并到哪一个分支。我个人更喜欢更明确些，但是你可以随便选择...

在解决了合并冲突问题之后（就像我上面所做的那样），请记得将这个文件添加给索引。

服务器、分支和标签

5. 远程服务器

Git 最强大的功能之一是可以有一个以上的远程服务器（另一个事实，你总是可以运行一个本地仓库）。你不一定总是需要写访问权限，你可以从多个服务器中读取（用于合并），然后写到另一个服务器中。添加一个远程服务器很简单：

```
$ git remote add john git@github.com:johnsomeone/someproject.git
```

如果你想查看远程服务器的相关信息，你可以这样做：

```
# shows URLs of each remote server
$ git remote -v

# gives more details about each
$ git remote show name
```

你可以查看本地分支和远程分支之间的差别：

```
$ git diff master..john/master
```

你也能查看不在远程分支上的 HEAD 的变化：

```
$ git log remote/branch..
# Note: no final refspec after ..
```

6. 标签

在 Git 中存在两种类型的标签 - 一个轻量级标签和一个注解标签。记着第二个提示中说过 Git 是基于指针的，二者的区别很简单。一个轻量级标签无非是一个指向提交的具名指针。你可以改变它并指向另一个提交。一个注解标签是一个指向标签对象的具名指针，这个标签对象拥有自己的消息和历史。如果有需要，标签对象的消息可以采用 GPG 加密签名。

创建两种类型的标签其实很容易（只是一个命令行选项的差异）

```
$ git tag to-be-tested
$ git tag -a v1.1.0 # Prompts for a tag message
```

7. 创建分支

在 Git 中创建分支非常容易（闪电般的速度，因为它仅仅需要创建一个不到100字节的文件）。创建一个新分支并切换过去的通用写法是：

```
$ git branch feature132
$ git checkout feature132
```

当然，如果你知道你要马上切换过去，你可以使用一条命令就能做到：

```
$ git checkout -b feature132
```

如果你要重命名一个本地分支，同样是件容易的事（长命令方式用来显示具体执行过程）：

```
$ git checkout -b twitter-experiment feature132
$ git branch -d feature132
```

更新：或者你（就像 Brian Palmer 在博客文章评论中指出的那样）只使用 “git branch” 和 -m 选项就可以一步到位：

```
$ git branch -m twitter-experiment
$ git branch -m feature132 twitter-experiment
```

8. 合并分支

在将来某个时候，你想要合并你的变更。有两种方式可以实现：

```
$ git checkout master
$ git merge feature83 # Or...
$ git rebase feature83
```

merge 和 rebase 的区别在于，merge 试图解决变更而且创建一个融合后的新提交，而 rebase 则试图把自你上次在其他分支上的变化，在另一个分支的 HAED 上重现。但是，在你向远程服务器推送一个分支之后，不要进行 rebase 操作 - 这会引发混淆/问题。

如果你不能确定哪些分支仍然有独立的工作在进行 - 以便你能知道你需要合并哪一个分支以及删除哪些分支，git branch 命令有两个选项可以帮助实现这一点：

```
# Shows branches that are all merged in to your current branch
$ git branch --merged

# Shows branches that are not merged in to your current branch
$ git branch --no-merged
```

9. 远程分支

如果你有一个本地分支，你想让它出现在远程服务器上，你可以使用一个推送命令：

```
$ git push origin twitter-experiment:refs/heads/twitter-experiment
# Where origin is our server name and twitter-experiment is the branch
```

更新：感谢 Erlend 在博客文章评论中提到的 - 这实际上和 git push origin twitter-experiment 达到的效果的一样，但是通过使用全部语法，你能看到你实际上在两端使用了不同的名字（你的本地名字可能是 add-ssl-support，而远程名字可能是

issue-1723）。

如果你想删除一个远程服务器上的分支（请注意分支名称之前的冒号）：

```
$ git push origin :twitter-experiment
```

如果你想显示所有远程分支的状态，你能像这样查看它们：

```
$ git remote show origin
```

这可能会列出一些服务器上曾经有过但现在已不存在的分支。如果碰到这种情况，你可以很轻松地使用如下命令从本地检出并将其删除：

```
$ git remote prune
```

最后，如果你有一个远程分支，你想在本地进行跟踪它，通常的做法是：

```
$ git branch --track myfeature origin/myfeature
$ git checkout myfeature
```

然而，如果你使用 -b 标识符去检出的话，新版的 Git 会自动建立跟踪：

```
$ git checkout -b myfeature origin/myfeature
```

在临时存放区、索引和文件系统中保存内容

10. 临时存放（**Stashing**）

在Git中，你可以把当前的工作状态储存在一个临时的存储区域堆栈，然后重新加以利用。简单的案例如下：

```
$ git stash # Do something...
$ git stash pop
```

很多人推荐使用 git stash apply 来代替 “pop”，然而如果你真这么做的话，你最终得到一个长长的毫无用处的储藏清单。如果对它进行清理，“pop” 只会把它从堆栈中删除。如果你已经使用了 git stash apply ，你可以使用如下命令从堆栈中删除最后一项：

```
$ git stash drop
```

Git 会基于当前的提交消息自动创建一个注释信息。如果你更喜欢使用一个自定义的消息（因为它可能和之前的提交无关）：

```
$ git stash save "My stash message"
```

如果你想从你的列表中（不必是最后一个）对一个特定的 stash 加以利用，你可以列出它们并像这样来使用它：

```
$ git stash list
  stash@{0}: On master: Changed to German
  stash@{1}: On master: Language is now Italian
$ git stash apply stash@{1}
```

11. 交互式添加

在 Subversion 的世界里，你修改文件然后只是提交有变化的文件。而在 Git 的世界里，你在提交某些文件甚至某些补丁上有更多的控制权。为了提交某些文件或者文件的某些部分，你必须进入交互模式。

```
$ git add -i
           staged     unstaged path


*** Commands ***
  1: status      2: update   3: revert   4: add untracked
  5: patch      6: diff     7: quit     8: help
What now>
```

这会让你进入一个基于交互式命令的菜单模式。你可以使用命令的数字符号或者加亮字符（如果你开启颜色高亮显示功能的话）进入对应模式，然后就是正常输入文件数的问题了（你可以使用像1或1-4或2,4,7这样的格式）。

如果你想进入修补模式（交互模式下输入‘p’或‘5’），你也可以直接进入那个模式：

```
$ git add -p
diff --git a/dummy.rb b/dummy.rb
index 4a00477..f856fb0 100644
--- a/dummy.rb
+++ b/dummy.rb
@@ -1,5 +1,5 @@
 class MyFoo
   def say
-    puts "Annyong Haseyo"
+    puts "Guten Tag"
   end
 end
Stage this hunk [y,n,q,a,d,/,e,?]?
```

如你所见，在底部你得到一系列选项为选择去添加文件改变的部分，这个文件的所有变化等等。使用‘?’命令可以了解选不同选项的解释。

12. 存储**/**从文件系统检索

一些项目（例如 Git 项目自身）直接在 Git 文件系统中存储额外的文件而不必是检入文件。

让我们开始在 Git 中存储一个任意文件：

```
$ echo "Foo" | git hash-object -w --stdin
51fc03a9bb365fae74fd2bf66517b30bf48020cb
```

此时，该文件对象已在数据库中，但是如果你不设置（一些东西）指向那个文件对象，它将被作为垃圾而回收。最简单的方法是标记它：

```
$ git tag myfile 51fc03a9bb365fae74fd2bf66517b30bf48020cb
```

既然在这里我们已经标记了 myfile。当我们需要获取该文件时，我们可以这样做：

```
$ git cat-file blob myfile
```

程序员可能经常用到的工具文件（密码、GPG 密钥、等等），不需要每次都检出到磁盘上（特别是在生产环境下），这种方法非常有效。

日志记录

13. 查看日志

如果你不使用‘git log’查看最近提交历史的话，你就不能长时间顺利地使用 Git。但是，也存在一些如何更好使用它的建议。例如，你可以查看每次提交中改变的一个补丁：

```
$ git log -p
```

或者你可以只是查看一个哪些文件有所更改的概述：

```
$ git log --stat
```

你可以在一行中设置一个不错的别名，用于显示简短的提交和漂亮的带有消息的分支图（像 gitk，但在命令行上）：

```
$ git config --global alias.lol "log --pretty=oneline --abbrev-commit --
$ git lol
* 4d2409a (master) Oops, meant that to be in Korean
* 169b845 Hello world
```

14. 检索日志

如果你想在日志中查询一个特定作者，你可以这样指定：

```
$ git log --author=Andy
```

更新：感谢 Johannes 的评论，我终于化解了一部分困惑。

或者如果你有一个搜索词出现在提交消息中：

```
$ git log --grep="Something in the message"
```

还有一个功能更强大的叫 pickaxe 的命令，它可以查找条目用来添加或删除一个特定的内容（也就是，当它第一次出现或被删除的时候）。这样你就可以知道何时增加了一行（但是如果那一行中的字符随后被改变，你将无从得知）：

```
$ git log -S "TODO: Check for admin status"
```

如果你改变一个特定的文件会怎么样呢，例如 lib/foo.rb

```
$ git log lib/foo.rb
```

比如说你有一个 feature/132 分支和一个 feature/145 分支，你想查看在这些分支但却不在主分支上的提交（备注：^ 代表非）：

```
$ git log feature/132 feature/145 ^master
```

你也可以使用 ActiveSupport 风格的日期缩小日期范围：

```
$ git log --since=2.months.ago --until=1.day.ago
```

它默认使用 OR 模式来组合查询，但是你也可以很轻松地改为 AND 模式（如果你的查询项不止一个的话）

```
$ git log --since=2.months.ago --until=1.day.ago --author=andy -S "some
```

15. 选择查看/修改的版本

当引用一个修订版本时，你有许多选项可以选择，当然，这取决于你对此功能的了解程度：

```
$ git show 12a86bc38 # By revision
$ git show v1.0.1 # By tag
$ git show feature132 # By branch name
$ git show 12a86bc38^ # Parent of a commit
$ git show 12a86bc38~2 # Grandparent of a commit
$ git show feature132@{yesterday} # Time relative
$ git show feature132@{2.hours.ago} # Time relative
```

请注意，和上一节有所不同，在行尾的脱字符表示提交的 parent - 行首的脱字符则表示不在这个分支上。

16. 选择一个范围

最简单的方法是这样来用：

```
$ git log origin/master..new
# [old]..[new] - everything you haven't pushed yet
```

你也可以删除 [new]，这将使用当前的 HEAD。

时间回退和错误修复

17. 重置更改

如果你还没有提交一个更改，你可以很容易地重置它：

```
$ git reset HEAD lib/foo.rb
```

通常使用'unstage'作为别名比较好，因为它不是那么显而易见。

```
$ git config --global alias.unstage "reset HEAD"
$ git unstage lib/foo.rb
```

如果你已经提交了文件，你可以做两件事情 - 如果是最后一次提交，你可以这样来修改：

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

这将回滚到最后一次提交，让你的工作副本回到变化存储在暂存区的状态，你可以编辑提交消息准备下一次提交。

如果你的提交不止一次，并且只想完全回滚它们，你可以重置分支回到之前的时间点。

```
$ git checkout feature132
$ git reset --hard HEAD~2
```

如果你真的想把分支指向一个完全不同的 SHA-1（也许你把一个分支的 HEAD 指向另一个分支，或者进一步提交），你可以按照以下方式去做：

```
$ git checkout FOO
$ git reset --hard SHA
```

实际上还有一种更便捷的方式（因为它不会先将你的工作副本变回最初 FOO 状态，然后再指向 SHA）：

```
$ git update-ref refs/heads/FOO SHA
```

18. 提交到错误的分支

好吧，让我们假设你提交到主分支，但应该已经创建了一个叫做 experimental 的主题分支。为了移除这些变化，你可以在当前点创建一个分支，回退 HEAD，然后检出新的分支：

```
$ git branch experimental   # Creates a pointer to the current master st
$ git reset --hard master~3 # Moves the master branch pointer back to 3
$ git checkout experimental
```

如果你已经在一个分支的一个分支的一个分支等上面做了些变更，这将会更复杂。然后你需要做的就是在这个分支上将其变更 rebase 到另一个的地方：

```
$ git branch newtopic STARTPOINT
$ git rebase oldtopic --onto newtopic
```

19. 交互式 **rebasing**

这是一个很酷的特性，我之前已看过演示，但当时没有真正搞明白，现在来看其实很简单。比方说，你已做了3次提交，但是你想对它们进行重新排序或者编辑（或者合并它们）：

```
$ git rebase -i master~3
```

然后你将编辑器打开。你所要做的就是修改 “pick/squash/edit 的指令来进行如何提交，然后保存/退出。在编辑之后，你可以使用 git rebase —continue 让你的每一个指令一个一个进行。

如果你选择编辑一个文件，这会让你停留在你提交时的状态，因此你需要使用 git commit —amend 对它进行编辑。

备注：在 REBASE 过程中不要进行提交工作 - 只能添加然后使用 —continue, —skip or —abort 选项。

20. 清理

如果你已经提交了一些内容到你的分支中（也许你是从SVN中的旧代码库导入的），你想从历史中删除掉所有的已提交内容：

```
$ git filter-branch --tree-filter 'rm -f *.class' HEAD
```

如果你已经向远程服务器推送过代码，但自那之后提交的都是一些垃圾，在推送之前你可以在本地系统上执行这样的操作：

```
$ git filter-branch --tree-filter 'rm -f *.class' origin/master..HEAD
```

各种各样的提示

21. 之前你看过的引用

如果你知道你之前已经查看过一个 SHA-1，但是你已经做了一些重置/回退工作，你可以使用 reflog 命令去查看你最近看过的 SHA-1：

```
$ git reflog
$ git log -g # Same as above, but shows in 'log' format
```

22. 分支命名

一个可爱的小提示 - 请记住，分支的名字并不局限于 a-z 和 0-9 这些字符。名字中可以使用 / 和 . 来伪装命名空间或者版本号，例如：

```
$ # Generate a changelog of Release 132
$ git shortlog release/132 ^release/131
$ # Tag this as v1.0.1
$ git tag v1.0.1 release/132
```

23. 寻找谁是始作俑者

寻找谁更改了一个文件中的一行代码经常会用到。简单命令如下：

```
$ git blame FILE
```

有时更改来自于前一个文件（如果你已经合并了两个文件，或者你已经移动了一个函数），因此你可以这样用：

```
$ # shows which file names the content came from
$ git blame -C FILE
```

有时通过向前或向后点击来进行变化跟踪，这是很好的方法。有一个内置的 GUI 程序专门为此设计：

```
$ git gui blame FILE
```

24. 数据库维护

Git 通常不需要大量维护，它基本上可以自我维护。然而，你可以使用如下命令查看数据库统计信息：

```
$ git count-objects -v
```

如果数值很高，你可以选择使用垃圾回收你的重复内容。这不会影响推送或者其它用户，但却可以让你的命令运行更快且占用更少空间：

```
$ git gc
```

经常运行一致性检查也是值得推荐的做法：

```
$ git fsck --full
```

你也可以在行尾添加一个 —auto 参数（如果你频繁运行它，或者在你的服务器上每日从 crontab 中运行它），如果统计数据表明必须进行要进行一致性检查，只要 fsck 命令就行。

如果检查 “dangling” 或 “unreachable” 的结果一切正常，这经常是由于回退 HEAD 或 rebasing的结果。如果检查“missing” 或 “sha1 mismatch” 出了问题...寻求专业帮助吧！

25. 恢复一个丢失的分支

如果你使用 -D 选项删除了一个分支 experimental，你可以重新创建它：

```
$ git branch experimental SHA1_OF_HASH
```

你可以使用 git reflog 来发现一个 SHA-1 哈希值，如果你近期访问过它的话。

另一种方法是使用 `git fsck —lost-found`。一个悬空的提交就是一个 lost HEAD（它只会是一个已删除分支的 HEAD，因为当一个 HEAD^ 被 HEAD 引用时，它就没有悬空）

作者： *Andy Jeffries*，*Ruby on Rails* 开发者 & 跆拳道教练，生活在英国伦敦。

原文： *25 Tips for Intermediate Git Users*（*2009*）

感谢： *Jodoo* 帮助审阅并完成校对。