给 Git 中级用户的 25 个小贴士

Andy Jeffries 给 Git 中级用户总结分享的 25 个小贴士。你不需要去做大量搜索，或许这些小贴士对你就很有帮助的。

我从开始使用git到现在已经差不多18个月了，以为自己已经很懂git了。直到我看到github上 Scott Chacon在 LVS, a supplier/developer of betting/gaming software 上的教学，第一天就受益匪浅。

作为一个很享受git的人，我想要分享从各种社区学到的实用经验，让大家不需要花费过多的功夫就能找到答案。

基本技巧

1.安装后的第一步

安装git后，第一件事你需要设置你的名字和邮箱，因为每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.是基于指针的

git上的所有东西都是储存在文件里的，当你创建一次提交时，它会创建一个包含你的提交信息和相关数据（名字，邮箱，日期/时间、上一次提交等等）的文件并连接一个树文件，而这个树文件包含了对象列表或者其他树。这上面的对象或者blob文件就是这次提交的实际内容（你可以认为这也是一个文件，尽管并没有储存在对象里而是储存在树中）。所有的文件都以经过SHA-1计算后的文件名（译者注：经过SHA-1计算后的数，即git中的版本号）储存在上面。

从这里可以看出，分支和标签都是包含一个指向这次提交的sha-1数（版本号）简单的文件，这样使用引用会变得更快和更灵活，创建一个新的分支是就像创建文件一样简单，SHA – 1数（版本号）也会引用你这个分支的提交。当然，如果你使用GIT命令行工具（或者GUI）你将无法接触这些。但真的很简单。

你可能听说过HEAD引用，这是一个指向你当前提交的内容的SHA-1 数（版本号）的指针。如果你正在解决合并冲突，使用HEAD不会对你的特定分支有任何改动只会指向

你当前的分支。

所有分支的指针都保存在 .git/refs/heads，HEAD指针保存在.git/HEAD，标签则保存在 .git/refs/tags，有时间就去看看吧。

3. 两个母体（Parent），当然!

当我们在日志文件中查看合并提交信息，你会看到两个母体，第一个母体是正在进行的分支，第二个是你要合并的分支。

4.合并冲突

现在，我发现有合并冲突并解决了它，这是一件在我们编辑文件时很正常的事。
将 <<<<, ====, >>>> 这些标记移除后，并保存你想要保存的代码。有些时候在代码被直接替代之前，能看到冲突是件挺不错的事。比如在两个冲突的分支变动之前，可以用这样的命令方式:

```
$ 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
If the file is binary, diffing files isn't so easy… What you'll
normally want to do is to try each version of the binary file and
decide which one to use (or manually copy portions over in the
binary file's editor). To pull a copy of the file from a
particular branch (say you're merging master and feature132):
```

如果是二进制文件（binary），区别这些文件并不容易。通常你会查看每个二进制文件的版本，再决定使用哪个（或者在二进制文件编辑器中手动复制），并将其推送至特定的分支。（比如你要合并master和feature132）

```
$ git checkout master flash/foo.fla
$ git checkout feature132 flash/foo.fla
$
$ git add flash/foo.fla
Another way is to cat the file from git – you can do this to another filename then copy the correct file over (when you've decided which it is) to the normal filename:
```

另一个方法就是在git中cat文件，你可以将其命名为另一个文件名，然后将你决定的那个文件改为正确的文件名：

```
$ git show master:flash/foo.fla > master-foo.fla
$ git show feature132:flash/foo.fla > feature132-foo.fla
$
$
$ rm flash/foo.fla
$ mv feature132-foo.fla flash/foo.fla
$ rm master-foo.fla
$ git add flash/foo.fla
```

更新：感谢carls在原博评论中提醒我，可以使用 “git checkout —ours flash/foo.fla” 和“git checkout —theirs flash/foo.fla” 在不用考虑你需要合并的分支来检查指定版本，就我个人而言，我喜欢更明确的方法，但这也是一个选择...

记住，解决完合并冲突后要添加文件。（我之前就犯过这样的错误）

服务，分支和标注

5. 远程服务

Git有一个非常强大的特性，就是可以有多个远程服务端（以及你运行的一个本地仓

库）。你不需要总是进行访问，你可以有多个服务端并能从其中一个（合并工作）读取再写入另一个。添加一个远程服务端很简单：

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

If you want to see information about your remote servers you can do:

```

如果你想查看远程服务端的信息你可以：

```

# shows URLs of each remote server

$ git remote -v

# gives more details about each

$ git remote show name

You can always see the differences between a local branch and a remote branch:

```

你总是能看到本地分支和远程分支不同的地方：

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

You can also see the changes on HEAD that aren't on that remote branch:

```

你同样也能看到远程分支上没有的HEAD指针的改动：

6. Tagging 标签

在Git中有两种类型的标注：轻量级标注和注释型标注。

记住第二个是Git的指针基础，两者区别很简单，轻量级标注是简单命名提交的指针，你可以将其指向另一个提交。注释型标注是一个有信息和历史并指向标注对象的名字指针，它有着自己的信息，如果需要的话，可以进行GPG标记。

创建两种类型的标签很简单（其中一个命令行有改动）

```
$ git tag to-be-tested
```

```
$ git tag -a v1.1.0
```

7. Creating Branches 创建分支

在git中创建分支是件非常简单的事情（非常快并只需要不到100byte的文件大小）。创建新分支并切换到该分支，通常是下面这样的：

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

当然，如果你想切换到该分支，最直接的方式是使用这样一条命令：

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

如果你想要重新命名本地分支，也很简单：

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

更新：或者你（Brian Palmer在原博的评论中指出的）可以使用 -m来切换到“git branch”（就像Mike指出，如果你只需要一个特定的分支，就可以重命名当前分支）

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

8.合并分支

以后你可能回想合并你的变动，有两种方式可以做到这一点：

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

merge和rebase的区别是，merge会尝试解决改动并创建的新的提交来融合他们。rebase则是将从你最后一次从另一个分支分离之后的改动并入，并直接沿用另一个分支的head指针。尽管如此，在你往远端服务器上推送分支之前，不要使用rebase。这会让你混乱。

如果你不能确定哪个分支（哪些需要合并，哪些需要移除）。这里有两个git分支切换方式来帮助你：

```
# 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

如果你想将本地分支放置远程服务端，你可以用这条命令进行推送：

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

如果你想要从服务端删除分支：

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

如果你想要查看远程分支的状态：

这将列出那些曾经存在而现在不存在的远程分支，这将帮助你轻易地删除你本地多余的分支。

最后，如果本地追踪远程分支，常用方式是：

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

尽管这样，Git的新版本将启动自动追踪，如果你使用-b来checkout：

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

Storing Content in Stashes, Index and File System 在stash储存内容、索引和文件系统

10. Stashing

在Git中你可以将当前的工作区的内容保存到Git栈中并从最近的一次提交中读取相关内容。以下是个简单的例子:

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

很多人推荐使用git stash apply来代替pop。这样子恢复后储存的stash内容并不会删除，而‘pop’恢复的同时把储存的stash内容也删了，使用git stash apply 就可以移除任何栈中最新的内容。

```
<code data-language="javascript">$ git stash drop
</code>
```

git可以自动创建基于当前提交信息的指令，如果你更喜欢使用通用的信息（相当于不会对前一次提交做任何改动）

```
<code data-language="javascript">$ git stash save "My stash message"
</code>
```

如果你想使用某个stash（不一定是最后一个），你可以这样将其列表显示出来然后使用:

```
<code data-language="javascript">$ git stash list
  stash@{0}: On master: Changed to German
  stash@{1}: On master: Language is now Italian
$ git stash apply stash@{1}
</code>
```

11.添加交互

在svn中，如果你文件有了改动之后，然后会提交所有改动的文件，在 Git中为了能更好的提交特定的文件或者某个补丁，你需要在交互模式提交选择提交的文件的内容。

```
$ 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&gt;
```

这是基于菜单的交互式提示符。您可以使用命令前的数字或进入高亮字母(如果你有高亮输入)模式。常用形式是，输入你想执行的操作前的数字。(你可以像1或1－4或2、4、7的格式来执行命令)。

如果你想进入补丁模式（在交互模式中输入p或5），同样也可以这样操作:

```
$ 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，你将在选择添加改动的那部分文件的底部获得一些选项。此外，使用“？”会说明这个选项。

12. 文件系统中的储存/检索

有些项目（比如Git自己的项目）需要直接在Git的文件系统中添加额外的并不想被检查的文件。

让我们开始在Git中保存随机文件

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

比如数据库中的对象，如果你不想让一些对象被垃圾回收，最简单的方式是给它加标签:

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

在这里我们设置myfile的标签，当我们需要检索该文件时可以这样:

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

这对开发者可能需要的但是并不想每次都去检查的有用文件（密码，gpg键等等）很管用（特别是在生产过程中）。

Logging and What Changed? 记录日志和什么改变了?

13. 查看日志

在不使用“git log”的情况下，你不能查看你长期的最近提交内容，但是，仍然有一些更便于你使用的方法，比如，你可以这样查看单次提交变动的内容:

或者你只看文件变动的摘要:

这个很赞的别名，可以让你在一行命令下简化提交，并展示不错的图形化分支。

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

14.在日志中查找

如果你想根据指定的作者查找:

更新：感谢 Johannes的评论，解除了我的一些困惑，

或者你可以搜索你提交信息的内容:

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

这些强大的指令被称为pickaxe指令，来检查被移除或添加特定块的内容（比如，当他们第一次出现或者被移除），添加任何一行内容都会告诉你（但是并不包括那行内容刚刚被改动）

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

如果你改动一个特定的文件会怎么样？如：lib/foo.rb

如果你有feature/132 和ferature/145这两个分支，并想查看这些不在master上的分支内容。（ ^ 符号是意味着非）

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

你同样可以使用ActiveSupport风格的日期来缩短时间范围:

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

默认会使用OR来合并查询，但你也可改用AND（如果你有不止一个条件）

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

15.选择试图/改动的之前的版本。

根据你知道的信息，可以按照以下方式来找到之前的版本:

```
$ 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
```

注意：不像前一部分所说，在最后的插入符号意味着提交的父类，在前面的插入符号意味着不在这个分支上。

16. 选择一个方式

最简单的方式：

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

你也可以省略[new]，这样将默认使用当前的HEAD指针。

Rewinding Time & Fixing Mistakes 回滚和修复错误

17.重置更改

如果你没有提交你可以简单的撤销改动：

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

通常我们使用”unstage“这样的别名来代替：

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

如果你已经提交了，有两种情况：如果是最后一次提交你仅仅需要amend：

这将不执行最后一次提交，恢复你原来的内容，提交信息将默认为你下次提交的信息。

如果你已经提交过不止一次了并且想完全回到之前那个记录，你可以重置分支回到指定的时间。

```
```

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

如果你想将分支回滚但想要SHA1数（版本号）不一样（也许你可以将分支的HEAD指向另一个分支，或者之后的提交），你可以通过如下方式：

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

实际上还有个更快的方式（这样并不会改变你的文件复制内容，并回归到第一次FOO的状态并指向SHA）

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

18. 提交至错误的分支

好吧，假定你提交到master上了，但是你想提交的是名为experimental的主题分支上，如果想移除这个改动，你可以在当前创建一个分支并将head指针回滚再检查新的分支

```
$ git branch experimental   # Creates a pointer to the current master state

$ git reset --hard master~3 # Moves the master branch pointer back to 3 revisions ago

$ git checkout experimental
```

如果你在分支的分支的分支进行了改动将会很麻烦，那么你需要做的就是在其他处进行分支rebase改动

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

19. rebase的交互

这是个很不错的功能，我曾看过演示但一直以来并没有真正搞懂，现在我知道了，非常简单。假如你进行了三次提交，但是你想重新编辑它们（或者结合它们）。

然后你让你的编辑器打开一些指令，你需要做的就是修改指令来选择/squash/编辑(或

删除)/提交和保存/退出，编辑完使用`git rebase –continue`来通过你的每一个指令。

如果你选择编辑一个，它将离开你的提交状态，所以你需要使用git commit -amend来编辑它。

注意：不要在rebase的时候提交——只能添加了之后再使用—continue, —skip 或—abort.

20. 清除

如果你在分支中提交了一些内容（也许是一些SVN上老的资源文件）并想从历史记录中完全移除，可以这样：

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

如果你已经将其推送至origin，并提交了一些垃圾内容，你同样可以推送之前在本地系统这样做：

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

Miscellaneous Tips 各种各样的技巧

21.你看过的前面的引用

如果你知道你之前看到的SHA-1数（版本号），并需要进行一些重置/回滚，可以使用reflog命令查询最近查看的sha – 1数（版本号）：

```
$ git reflog

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

22. 分支命名

一个有趣的小技巧，不要忘记分支名不仅仅限于a-z和0-9，在名字中使用/和.用于命名伪命名空间和版本控制，也是个不错的主意，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. 找到Dunnit

找出谁在一个文件中改变了一行代码，简单的命令是:

有时候是上一个文件发生了变动(如果你合并两个文件，或者你已经转移到一个函数)，这样你就可以使用:

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

有时候需要通过点击来追踪来回的变动，这里有一个不错的内置gui:

24. 数据库维护

通常Git并不需要过多的维护，它几乎可以自己搞定，尽管如此你也可以查看数据库使用的统计:

如果数值过高你可以选择将你的克隆垃圾回收。这不会影响你推送内容或其他人，但它可以让你的命令运行的更快，并使用更少的空间:

它也可以在运行时进行一致性检验:

你可以在后面添加-auto 参数（如果你在服务器跑定时任务时），这在统计数据时是必须的。

当检查的结果是“dangling”或“unreachable”这样的是正常的，这通常是回滚和rebase的结果。 得到“missing” 或 “sha1 mismatch” 这样的结果是不好的...你需要得到专业的帮助!

25. 恢复失去的分支

如果你意外的删除一个分支，可以重新创建它:

```
$ git branch experimental SHA1_OF_HASH
```

你可以使用git reflog查看你最近访问过的SHA1数（版本号）

另一个方式就是使用 `git fsck —lost-found` ，悬空对象（dangling commit ）是就是失去HEAD指针的提交，（删除的分支只是失去了HEAD指针成为悬空对象）

Done!完成!

这篇是我写过最长的博文，希望大家能从此文中获益，如果你有所收益或是有任何问题都可以在评论中告诉我！