【Git学习】

【简单指令】

$ git config --global user.name "username" 创建全局的用户名

$ git config --global user.email "email" 创建全局的用户邮箱

$ mkdir xxxx 创建文件夹

$ cd Y:/WS/Git/GitSkill 打开目录 （注意用 / ）

$ cd .. 返回上一级目录

$ pwd 查看当前目录

$ ls 查勘当前目录下的文件清单

$ rm text.txt 删除文件

Q 当遇到END等字，无法继续输入命令行时，用Q退出。

【简单Git指令】

$ git init 初始化文件夹，创建.git文件

$ git add readme.txt 添加文件到暂存区

$ git commit -m "wrote a readme.txt file" 将暂存区的文件提交到分支

$ git status 查看当前的状态（显示当前分支，暂存区是否有文件可以被提交，工作区是否有文件被修改）

$ git diff readme.txt 查看工作区对最新版本做了哪些修改

$ git log 查看历史提交详细日志（包括id，分支，用户名，时间，log）

$ git log --pretty=oneline 查看历史提交日志的简单信息（包括id，分支，log）

$ git reset --hard HEAD^ 回滚到上一个版本

$ git cat readme.txt 打印出文件的内容

$ git reset --hard 574b4f 回滚到某一个id的版本（id是16进制的提交记录代码）

$ git reflog 查看命令历史，以便确定要回到未来的哪个版本。

$ git diff HEAD -- readme.txt 查看工作区和版本库里面最新版本的区别

【撤销、恢复与删除指令】

$ git reset HEAD readme.txt 把暂存区中对文件的修改撤销掉（unstage），不管修改了几次，就像删除了暂存区的文件。（git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示版本库的最新版本。此时再使用checkout就会把工作区恢复到版本库最新版的状态。）

$ git checkout -- readme.txt 如果修改了工作区但没添加到暂存区，就会恢复到和版本库一样的状态；如果添加到了暂存区后又修改了工作区，就会恢复到和暂存区一样的状态。（--很重要，没有--，就变成了“切换到另一个分支”的命令）

【场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。】

$ git rm text.txt 删除工作区和暂存区的文件。删除之后，如果直接用$ git checkout text.txt无法恢复工作区的text.txt，因为最近的一次$ git rm操作后没有commit，所以工作区恢复的是暂存区的状态，而暂存区已经没有text.txt了。所以要用$ git reset HEAD text.txt撤销暂存区中删除text.txt的操作，再$ git checkout text.txt把工作区恢复成暂存区的状态。

【创建远程仓库】

$ git remote add origin https://github.com/cfjsmhwp/Git-Learning.git

关联本地的仓库（master）和远程仓库（origin）

$ git push -u origin master

第一次推送远程仓库，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

$ git push origin master

把本地master分支的最新修改推送至GitHub。

$ git clone git@github.com:cfjsmhwp/Git.git

克隆一个远程库（origin）到本地（master）

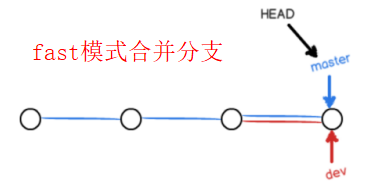
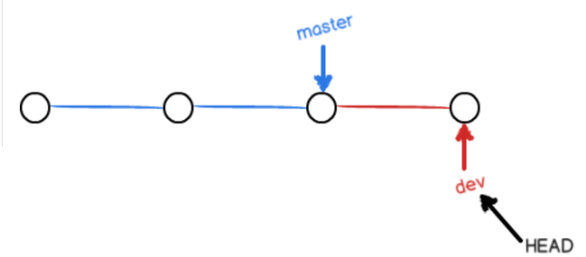
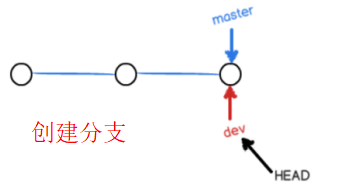
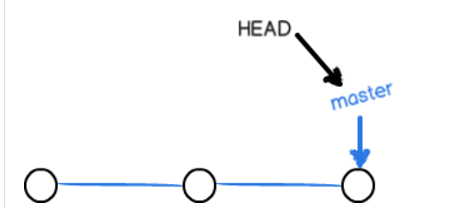
【https://github.com/cfjsmhwp/Git.git

git@github.com:cfjsmhwp/Git.git

Git支持多种协议，默认的git://使用ssh，但也可以使用https等其他协议。Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

使用https除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。】

【分支】



每次提交，Git都会把它串成一条时间线。第一幅图只有一条分支，就是主分支。HEAD指向master，而master才是指向提交的。HEAD指向的就是当前的分支。因此就能确定当前分支以及分支的提交点。

创建一个新的分支dev的时候，Git创建了一个新的指针dev，指向master相同的提交点，再把HEAD指向dev，就表示当前分支在dev上。因此创建一个新的分支很快，只是增加了一个指针。

接着工作区修改和提交就会在dev分支上增加时间节点。

Dev工作完成之后，要把dev合并到master上，就直接把master指向dev当前提交的时间节点，完成合并。

$ git branch 查看分支

$ git branch dev 创建dev分支

$ git checkout dev 切换到dev分支

$ git checkout -b dev 创建+切换到dev分支

$ git merge dev 合并dev分支到当前分支

$ git branch -d dev 删除dev分支

$ git branch -D dev 如果某个分支还没有合并，要强行删除，要用这个

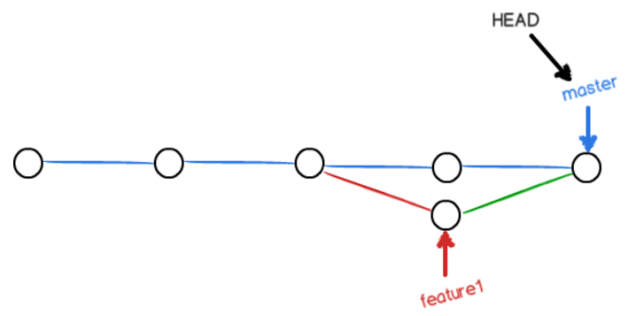
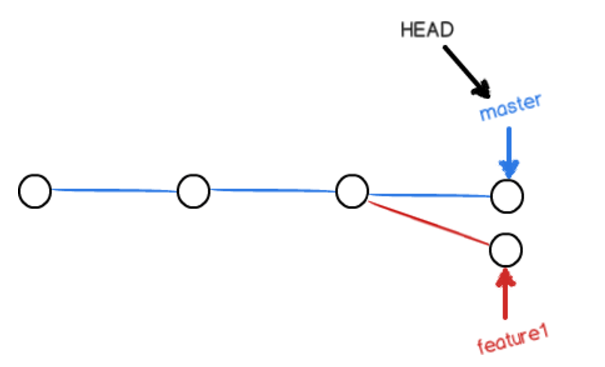
【Your branch is up-to-date with 'origin/master'.

当前分支和远程分支版本的比较信息，表示当前分支是最新的。】

【Your branch is ahead of 'origin/master' by 1 commit.

当前分支和远程分支版本的比较信息，表示当前分支比远程仓库多了一个提交。】

【冲突】



图一表示：Master和featurel分支都各自有了新的提交，这时候Git无法快速合并（修改master指针），合并的话就会有可能有冲突产生。以下是冲突产生后文件的内容。

【<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

Creating a new branch is quick AND simple.

>>>>>>> featurel】

Git用<<<<<<<，=======，>>>>>>> 标记出不同分支的内容。

修改文件的内容后，再次add、commit文件。就变成了图二。

【用带参数的git log命令行，可以看到分支以及合并的情况。

$ git log --graph --pretty=oneline --abbrev-commit

\* eaed092 (HEAD -> master) conflict fixed

|\

| \* 8343538 (featurel) AND simple

\* | 8971f5e & simple

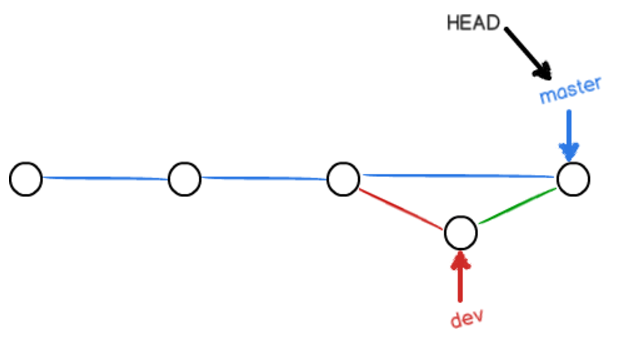
|/

\* 7d93a92 branch test

\* 2834b5e (origin/master, origin/HEAD) Initial commit

】

【分支管理策略】



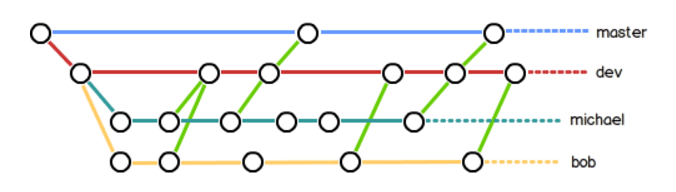
用普通的方式合并的时候，如果条件可能（不发生冲突），Git就会使用Fast Forward模式。

这种模式下会看不到分支合并的历史，不产生新的合并的节点，而是把master的指针直接指到了dev的节点上。删除分支后，就会丢失分支信息。

如果禁止Fast Forward模式，Git在merge的时候就会产生一个commit的节点，这样从分支历史上就可以看出分支信息。如图一。

$ git merge --no-ff -m "merge with no-ff" dev

合并dev分支，--no-ff参数，表示禁止Fast Forward模式。-m参数，因为有一个新的commit所以要加上把描述添加进去。



在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

所以，团队合作的分支看起来就像这样：如图二

【Bug分支，stash暂存区】

当你接到一个代号为101的bug需要立即修复的时候，你想创建一个分支issus-101来修复它，但是当前正在dev上开发的工作还没有完成，不能提交。Git提供了一个Stash功能，可以把当前的工作现场“储藏”起来（stash翻译为隐藏，贮藏）方便以后恢复。

$ git stash

现在，用git status查看dev工作区，就是干净的（因为恢复到了某个提交后的节点）（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

【Saved working directory and index state WIP on dev: 6224937 add merge（描述）

HEAD is now at 6224937 add merge（描述）

提示表示为，将dev到一半的工作区保存，并且将现在的工作区指向dev的某个节点（id：6224937；描述：add merge）。这个时候你可以切换到别的分支，然后在这个分支上创建issus-101分支解决bug。】

$ git stash list 可以查看多个被stash储藏起来的工作区

$ git stash pop 恢复被stash储藏起来的工作区，并且删除stash中的内容

$ git stash apply 恢复被stash储藏起来的工作区，但不删除stash中的内容

$ git stash drop 删除stash中的内容

$ git stash apply stash@{0} 指定恢复被stash储藏起来的工作区

现在，bug已经修复。你需要恢复之前工作一半的工作现场。首先切换回dev分支，然后用git stash list命令查看

【$ git stash list

stash@{0}: WIP on dev: 6224937 add merge（描述）】

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；另一种方式是用git stash pop，恢复的同时把stash内容也删了。

【远程分支拉取，推送】

$ git remote 查看远程库信息

$ git remote -v 查看远程库抓取（fetch）和推送（push）地址的信息

$ git push origin master 推送master分支

$ git push origin dev 推送dev分支

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

master分支是主分支，因此要时刻与远程同步；

dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；

bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；

feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

$ git clone <https://github.com/cfjsmhwp/GitSkill.git> 默认克隆后本地只能看到master分支

$ git checkout -b dev origin/dev 创建远程origin的dev分支到本地

【$ git checkout -b dev origin/dev

Switched to a new branch 'dev'

Branch dev set up to track remote branch dev from origin.】

多人协作时，大家都会往master和dev分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的其他目录下克隆。当小伙伴从远程库clone时，默认情况下，小伙伴只能看到本地的master分支。不信可以用git branch命令看看。现在，小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

$ git checkout -b dev origin/dev。接着，他就可以在dev上继续修改，然后时不时通过$ git push origin dev把dev分支push到远程。

【远程推送冲突】

当你的推送和小伙伴的推送产生了冲突的时候，直接用$ git push origin dev就会产生错误：

【$ git push origin dev

To https://github.com/cfjsmhwp/GitSkill.git

! [rejected] dev -> dev (fetch first)

error: failed to push some refs to 'https://github.com/cfjsmhwp/GitSkill.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

推送失败，因为小伙伴的最新提交和你试图推送的提交有冲突。

解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送】

$ git pull

但是直接使用 $ git pull也失败了，原因是没有设置本地dev分支和远程origin/dev分支的链接，错误如下：

【$ git pull

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

From https://github.com/cfjsmhwp/GitSkill

dc23c69..857f4e6 dev -> origin/dev

There is no tracking information for the current branch.

Please specify which branch you want to merge with.

See git-pull(1) for details.

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream-to=origin/<branch> dev

】

根据提示，设置dev和origin/dev的链接：

$ git branch --set-upstream dev origin/dev

【$ git branch --set-upstream dev origin/dev

The --set-upstream flag is deprecated and will be removed. Consider using --track or --set-upstream-to

Branch dev set up to track remote branch dev from origin.】

设置成功，这回再次 $ git pull就会成功拉取，但是合并有冲突，需要打开文件，手动解决。解决冲突之后，使用 $ git add 和 $ git commit 提交。最后就可以使用 $ git push origin dev推送到远程仓库了。

总结：

首先，可以试图用git push origin dev 推送自己的修改；

如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；

如果合并有冲突，则解决冲突，并在本地提交；

没有冲突或者解决掉冲突后，再用git push origin dev 推送就能成功！

如果git pull提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream dev origin/dev

【标签】

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像，但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。标签就是一个让人容易记住的有意义的名字，跟某一个commit绑定在一起。例如tag v1.2。

$ git tag v1.0 给当前分支（默认为HEAD）打上标签

$ git tag v0.9 857f4e6 给指定的commit id打上标签

$ git tag -a v0.8 -m “version v0.8 released message” 857f4e6 给指定id创建带有说明的标签

$ git tag -s v0.8 -m “version v0.8 released message” 857f4e6 采用PGP私钥签名的标签

$ git tag 查看当前分支的所有标签

$ git show v0.9 查看标签信息（说明，PGP签名）

$ git push origin v1.0 推送某个标签到远程

$ git push origin --tags 一次性推送全部尚未推送到远程的本地标签

$ git tag -d v0.1 删除本地的标签

$ git push origin :refs/tags/v0.9 要删除远程标签，需要先删除本地标签，然后在执行这一行

【在Github上fork开源项目】

在GitHub上，可以任意Fork开源仓库；

自己拥有Fork后的仓库的读写权限；但是没有作者的仓库的权限。

你自己修改了代码以后，可以在github上推送pull request给官方仓库来贡献代码。需要官方库接受。

【同步多个远程库，码云】

我们在本地库上使用命令git remote add把它和码云的远程库关联：

git remote add origin git@gitee.com:liaoxuefeng/learngit.git

之后，就可以正常地用git push和git pull推送了！

如果在使用命令git remote add时报错：

git remote add origin git@gitee.com:liaoxuefeng/learngit.gitfatal: remote origin already exists.

这说明本地库已经关联了一个名叫origin的远程库，此时，可以先用git remote -v查看远程库信息：

$ git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

可以看到，本地库已经关联了origin的远程库，并且，该远程库指向GitHub。

我们可以删除已有的GitHub远程库：

$ git remote rm origin

再关联码云的远程库（注意路径中需要填写正确的用户名）：

git remote add origin git@gitee.com:liaoxuefeng/learngit.git

此时，我们再查看远程库信息：

git remote -v

origin git@gitee.com:liaoxuefeng/learngit.git (fetch)

origin git@gitee.com:liaoxuefeng/learngit.git (push)

现在可以看到，origin已经被关联到码云的远程库了。通过git push命令就可以把本地库推送到Gitee上。

有的小伙伴又要问了，一个本地库能不能既关联GitHub，又关联码云呢？

答案是肯定的，因为git本身是分布式版本控制系统，可以同步到另外一个远程库，当然也可以同步到另外两个远程库。

使用多个远程库时，我们要注意，git给远程库起的默认名称是origin，如果有多个远程库，我们需要用不同的名称来标识不同的远程库。

仍然以learngit本地库为例，我们先删除已关联的名为origin的远程库：

git remote rm origin

然后，先关联GitHub的远程库：

$ git remote add github git@github.com:xxx/learngit.git

注意，远程库的名称叫github，不叫origin了。

接着，再关联码云的远程库：

$ git remote add gitee git@gitee.com:xxx/learngit.git

同样注意，远程库的名称叫gitee，不叫origin。

现在，我们用git remote -v查看远程库信息，可以看到两个远程库：

git remote -v

gitee git@gitee.com:liaoxuefeng/learngit.git (fetch)

gitee git@gitee.com:liaoxuefeng/learngit.git (push)

github git@github.com:michaelliao/learngit.git (fetch)

github git@github.com:michaelliao/learngit.git (push)

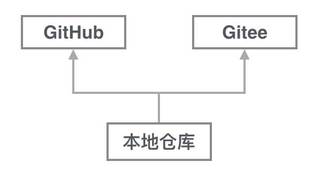
如果要推送到GitHub，使用命令：

$ git push github master

如果要推送到码云，使用命令：

$ git push gitee master

这样一来，我们的本地库就可以同时与多个远程库互相同步：



码云也同样提供了Pull request功能，可以让其他小伙伴参与到开源项目中来。你可以通过Fork我的仓库：[https://gitee.com/liaoxuefeng/learngit](https://gitee.com/liaoxuefeng/learngit" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)，创建一个your-gitee-id.txt的文本文件， 写点自己学习Git的心得，然后推送一个pull request给我，这个仓库会在码云和GitHub做双向同步。

【自定义Git】

$ git config --global color.ui true

【忽略特殊文件】

忽略文件的原则是：

忽略操作系统自动生成的文件，比如缩略图等；

忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；

忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

把.gitignore也提交到Git，就完成了！

检验.gitignore的标准是使用git status命令后提示working directory clean。

使用Windows的童鞋注意了，如果你在资源管理器里新建一个.gitignore文件，它会非常弱智地提示你必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。

有些时候，你想添加一个文件到Git，但发现添加不了，原因是这个文件被.gitignore忽略了：

$ git add App.class

The following paths are ignored by one of your .gitignore files:

App.class

Use -f if you really want to add them.

如果你确实想添加该文件，可以用-f强制添加到Git：

$ git add -f App.class

或者你发现，可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用git check-ignore命令检查：

$ git check-ignore -v App.class

.gitignore:3:\*.class App.class

Git会告诉我们，.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

.gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理！

【配置Git简写命令】

我们只需要敲一行命令，告诉Git，以后st就表示status：

$ git config --global alias.st status

好了，现在敲git st看看效果。

当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch：

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

以后提交就可以简写成：

$ git ci -m "bala bala bala..."

--global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

在[撤销修改](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001374831943254ee90db11b13d4ba9a73b9047f4fb968d000" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)一节中，我们知道，命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：

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

当你敲入命令：

$ git unstage test.py

实际上Git执行的是：

$ git reset HEAD test.py

配置一个git last，让其显示最后一次提交信息：

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

这样，用git last就能显示最近一次的提交：

$ git last

commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2

Merge: bd6ae48 291bea8

Author: Michael Liao <askxuefeng@gmail.com>

Date: Thu Aug 22 22:49:22 2013 +0800

merge & fix hello.py

甚至还有人丧心病狂地把lg配置成了：

git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

来看看git lg的效果：



为什么不早点告诉我？别激动，咱不是为了多记几个英文单词嘛！

配置文件

配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中：

$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

ignorecase = true

precomposeunicode = true

[remote "origin"]

url = git@github.com:michaelliao/learngit.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

[alias]

last = log -1

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中：

$ cat .gitconfig

[alias]

co = checkout

ci = commit

br = branch

st = status

[user]

name = Your Name

email = your@email.com

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。