# git基础

所有的版本控制系统, 其实只能跟踪文本文件的改动, 比如TXT文件, 网页, 所有的程序代码等等, Git也不例外.

初始化一个Git仓库：

git init

添加文件到Git仓库，分两步：

git add <file>，注意，可反复多次使用，添加多个文件；

git add –-all命令全部加入缓存区

git commit -m <message> 提交文件到版本库

要随时掌握工作区的状态：git status

如果git status告诉你有文件被修改过，用git diff <file>可以查看修改内容。工作区与缓冲区比较。

命令显示从最近到最远的提交日志

git log

简化log的输出信息

git log --pretty=oneline

回退到上一个版本

git reset --hard HEAD^

回退到指定版本(id可以只写前几位)

git reset --hard <commit\_id>

查看版本变更历史

git reflog

工作区: 在电脑里能看到的目录.

版本库(repository): 就是.git文件夹. Git的版本库里存了很多东西, 其中最重要的就是称为stage(或者叫index)的暂存区, 还有Git为我们自动创建的第一个分支master, 以及指向master的一个指针叫HEAD.



工作区, 版本库和暂存区的关系.

untracked状态是指该文件还没有被添加过.

查看工作区和版本库里面最新版本的区别：

git diff HEAD -- <file>

把<file>文件在工作区的修改全部撤销

git checkout -- <file>

一种是readme.txt自修改后还没有被放到暂存区, 现在, 撤销修改就回到和版本库一模一样的状态.

一种是readme.txt已经添加到暂存区后, 又作了修改, 现在, 撤销修改就回到添加到暂存区后的状态.

总之, 就是让这个文件回到最近一次git commit或git add时的状态.

已经add, 还没有commit, 取消缓存区的修改:

(1) git reset HEAD <file>

(2) git checkout -- <file>

如果已经commit了, 那就回退版本(看前面的内容).

git rm –cached <file>，可以从缓存区移除文件，使该文件变为未跟踪的状态，同时下次提交时从本地库中删除。

从版本库中删除文件:

(1) rm <file> //删除工作区的文件

(2) git rm <file> //删除版本库中的文件

(3) git commit -m "remove <file>" //提交删除

如果误删了, 要从版本库中回复最新的版本:

git checkout -- test.txt

# git与远程仓库github

本地git仓库关联远程的github服务器

先看看/c/Users/<user\_name>/.ssh/文件夹受否存在, 如果存在是否有id\_rsa和id\_rsa.pub这两个文件.

如果已经有了, 可以直接跳到下一步.

第一步:

创建SSH Key, 一路回车

ssh-keygen -t rsa -C "youremail@example.com"

id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

第二步:

登录github, 打开Account settings, SSH Keys, 然后, 点Add SSH Key,填上任意Title, 在Key文本框里粘贴id\_rsa.pub文件的内容.

git本地已经有一个仓库了, 可以在github上创建一个空的仓库<repository\_name>, 然后将本地仓库与远程仓库进行关联.

关联远程仓库：

git remote add origin [git@github.com:<github\_user\_name>/<repository\_name>.git](mailto:git@github.com:%3cgithub_user_name%3e/%3crepository_name%3e.git)

如git remote add origin [git@github.com:cMH2017/learngit.git](mailto:git@github.com:cMH2017/learngit.git)

远程仓库的名字就是origin.

把本地库的所有内容推送到远程库上：

git push -u origin master

-u参数, 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.

关联以后，推送最新修改:

git push origin master

查看远程仓库的信息:

git remote –v



显示出可以抓取和推送的origin地址

解除本地仓库与远程仓库的关联关系:

git remote rm origin

克隆一个本地库：（一定要用ssh地址， 如果用http地址， 每次push都要弹窗登录）

git clone git@github.com:<github\_user\_name>/<repository\_name>.git

Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

# 分支, 合并与冲突

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

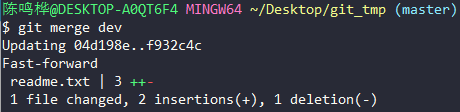
创建+切换分支：

git checkout -b <name>

推荐使用

git switch -c <name>

合并某分支到当前分支：git merge <name>

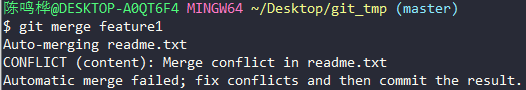


Fast-forward表示这次合并是快进模式, 也就是直接把master指向dev分支.

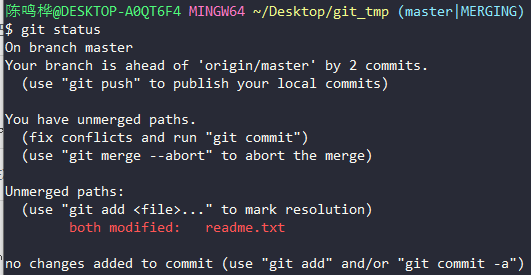
删除分支：git branch -d <name>

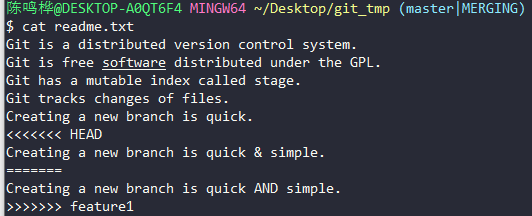
强行删除分支：git branch -D <name>

合并冲突, 无法执行Fast-forward



使用git status查看冲突情况



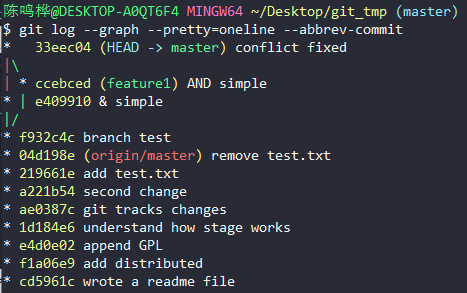


Git用<<<<<<<, =======, >>>>>>>标记出不同分支的内容

然后add和commit

看到分支的合并情况

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

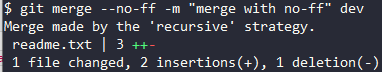


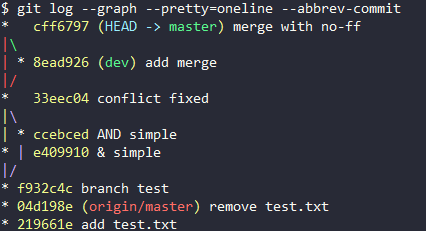
通常, 合并分支时, git会尽可能使用Fast forward方式, 但是这种方式会使删除分支后, 分支信息丢失.

如果要强制禁用Fast forward, git就会在merge时生成一个新的commit, 这样, 从历史分支中就可以看出分支信息.

--no-ff参数表示禁用Fast forward

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





删除dev分支后, 再看一下log



可以看到8ead9926后面没有(dev)了, 但是可以看出来曾经做过合并的痕迹.

下面看看开启Fast forward的情况

git switch -c dev

在readme.txt增加一行:

used Fast forward again.

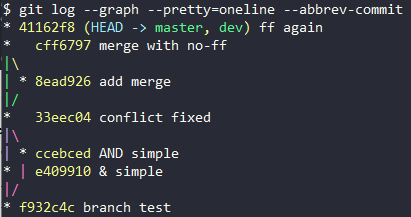
git add readme.txt

git commit -m "ff again"

git switch master

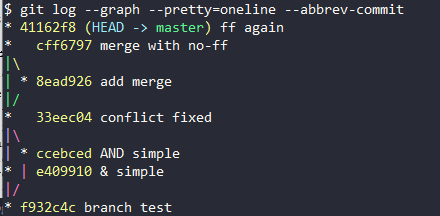
git merge dev

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



git branch -d dev

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



可以看到并没有dev的独立分支信息, 看不出来曾经做过合并.

# BUG分支

BUG分支的应用场景:

在开发dev分支时, 突然发现软件出bug, 但是当前分支的工作还没有完成, 可以使用stash功能隐藏当前的工作限现场.

工作区的内容是共享的, 比如在先在master分支创建一个dev分支, 然后再master分支下的readme.txt文件增加一行aaaaa.

从master切换到dev分支后, 还是能看到aaaaa.

git stash 把当前工作现场“储藏”起来，等以后恢复现场后继续工作

git stash list 查看被stash的工作现场

git stash apply 恢复后，stash内容并不删除

git stash apply stash@{0} 恢复指定的stash

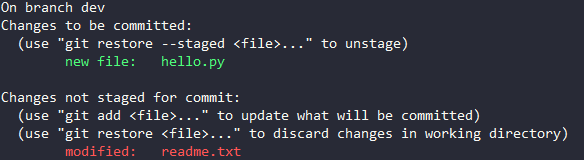
git stash drop 删除stash中的内容

git stash pop，恢复的同时把stash内容也删了

与stash相关的一系列操作

假如当前再dev分支, 缓冲区和工作区都有内容

git status



隐藏当前工作区的内容: git stash



git checkout master

git checkout -b issue-101

修改readme.txt的内容

git add readme.txt

git commit -m "fix bug 101"

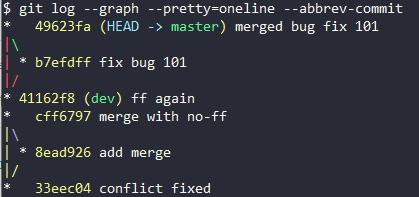
git switch master

git merge --no-ff -m "merged bug fix 101" issue-101

git branch –d issue-101

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

看到commit-id为49623fa



回去dev: git switch dev

git status

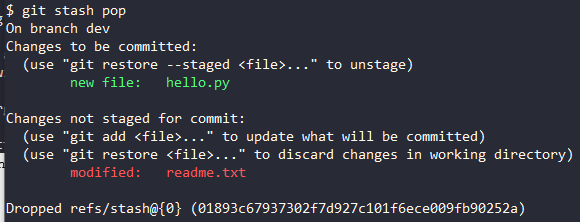
输出:

On branch dev

nothing to commit, working tree clean

git stash list

git stash pop



dev是从master分出来的, 再master中存在的bug在dev中也存在, 我们只需要把49623fa bug fix 101这个提交所作的修改复制到dev分支.

复制一个特定的提交到当前分支

git cherry-pick <commit-id>

但是在这之前应该保证dev的工作区是干净的, 即应该提交一次dev

但是

|  |
| --- |
| $ git cherry-pick 49623fa  error: commit 49623fab5ebec8d99b52a47cbb231ea09aab5c95 is a merge but no -m option was given.  fatal: cherry-pick failed |

要使用这条命令:

git cherry-pick -m 1 49623fa

-m 1告诉git要采用master分支的变动.

然后现在dev分支的readme.txt文件就有改好bug以后的内容了.

# feature分支

每添加一个新的功能, 最好新建一个feature分支, 在上面开发, 完成后, 合并, 最后删除feature分支.

要删除还没有合并的分支, 要使用-D选项, 即强行删除.

git branch -D <branch-name>

# 多人合作

查看远程库的信息

git remote -v

推送分支到远程仓库

git push origin <name>

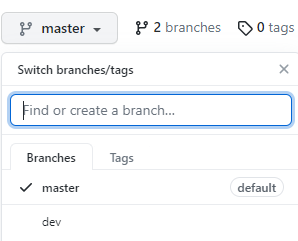
推动本地的某分支到远程仓库

git push origin <branch-name>

git push origin master

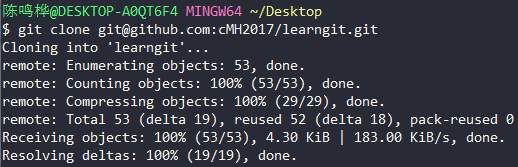
git push origin dev

可以在github上看到有两个分支



克隆远程仓库, 默认是克隆master分支

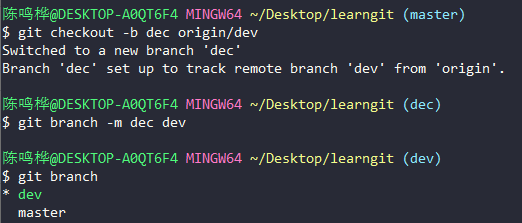
git clone [git@github.com:<user-name>/<repository-name>.git](mailto:git@github.com:%3cuser-name%3e/%3crepository-name%3e.git)





基于远程分支origin/dev创建本地分支dev:

git checkout -b dev origin/dev



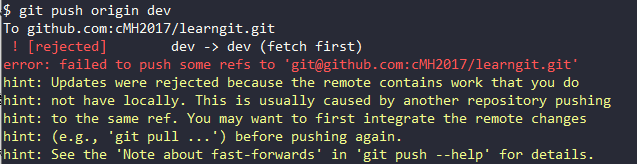
git的分支可以重命名

git branch -m <old-name> <new-name>

将本地的dev分支推动到远程

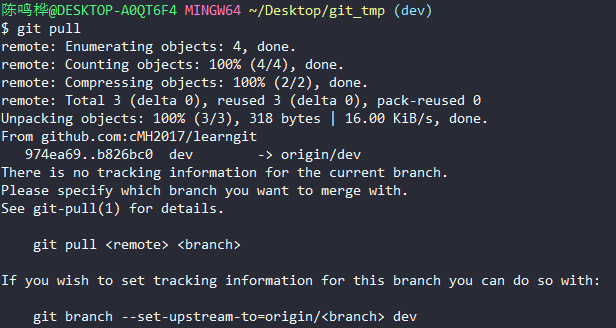
git push origin dev

两个人同时对dev进行开发, A先推送到远程了, B也要推送, 但是有冲突



解决办法: 先用git pull把最新的提交从origin/dev抓取下来, 然后在本地合并, 解决冲突再提交.

执行git pull (失败)



原因是没有指定本地dev分支与远程origin/dev分支的链接, 根据提示设置链接.

指定本地dev分支与远程origin/dev分支的链接:

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

然后

git pull 把最新的提交从origin/dev抓下来

按照处理冲突的流程处理完后再次push到远程仓库

git push origin dev

# 标签管理

标签是与commit-id绑定在一起的, 是一串方便人阅读的字符.

用于新建一个标签

git tag <tagname> [which]

which可以填commit-id号, 如果不填, 默认对HEAD打标签.

新建一个标签, 指定标签名和标签信息

git tag -a <tagname> -m <tag-info> [commit-id]

查看有哪些标签(按字典序列出)

git tag

查看标签信息:

git show <tag-name>

删除一个本地标签

git tag -d <tagname>

推送一个本地标签到远程

git push origin <tagname>

推送所有标签到远程

git push origin --tags

删除一个已经推送到远程仓库的标签

(1) 先从本地删除

git tag -d <tag-name>

(2) 从远程删除

git push origin :refs/tags/<tag-name>

让Git显示颜色，会让命令输出看起来更醒目：

git config --global color.ui true

# 忽略特殊文件

.gitignore文件, 把要忽略的文件名填进去，Git就会自动忽略这些文件。

所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>

一份C语言的gitignore文件示例

|  |
| --- |
| # Prerequisites  \*.d  # Object files  \*.o  \*.ko  \*.obj  \*.elf  # Linker output  \*.ilk  \*.map  \*.exp  # Precompiled Headers  \*.gch  \*.pch  # Libraries  \*.lib  \*.a  \*.la  \*.lo  # Shared objects (inc. Windows DLLs)  \*.dll  \*.so  \*.so.\*  \*.dylib  # Executables  \*.exe  \*.out  \*.app  \*.i\*86  \*.x86\_64  \*.hex  # Debug files  \*.dSYM/  \*.su  \*.idb  \*.pdb  # Kernel Module Compile Results  \*.mod\*  \*.cmd  .tmp\_versions/  modules.order  Module.symvers  Mkfile.old  dkms.conf  # 不排除.gitignore  !.gitignore |

无视.gitignore, 强制讲文件添加到版本库

git add -f App.class

检查.gitignore规则是否写错:

git check-ignore -v App.class

git config --global alias.st status 以后st就表示status

--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

# 配置别名

将命令git <command>配置为别名git <cmd>

git config --global alias.<cmd> <command>

有空格要加单引号

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

-- global表示针对当前用户起作用, 否则只针对当前仓库起作用.

git的配置文件在每个仓库的.git/config