本地：

设置全局

git config --global user.name "XiaoqiGuo"

git config --global user.email "Kallyqi@outlook.com"

修改后用

生成一个git仓库

git init

git add

git commit -m '评论'

状态查询

git status

查询提交历史：

git log --pretty=oneline

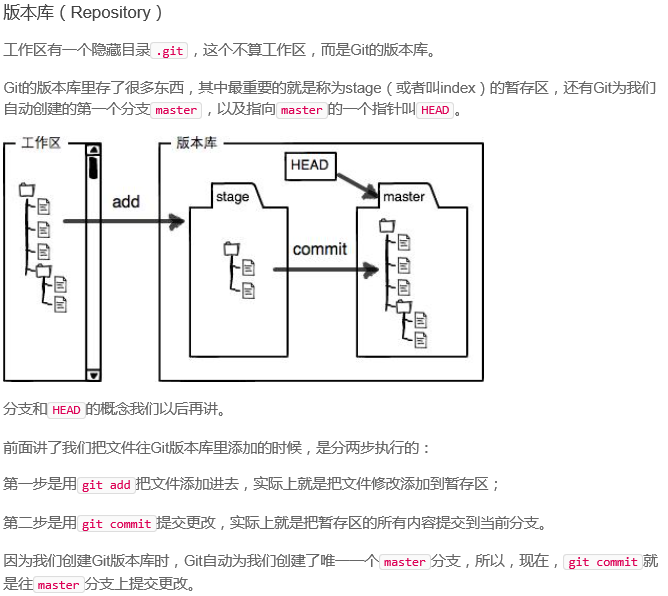
查询历史操作

git reflog

版本回退：HEAD为当前版本 HEAD^为上一个版本 HEAD^n n个版本

git reset --hard HEAD^

git reset --hard commit\_id



第一次修改 -> git add -> 第二次修改 -> git commit

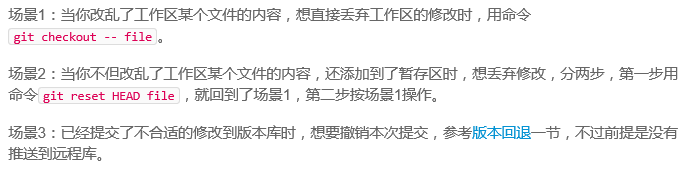
Git管理的是修改，当git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

工作区内容撤回暂存区内容：回退到最后一次 git commit或者git add状态（本地多加的修改都撤回.git种暂存区的内容）

git checkout – filename

本地回退：





本地工作区删除文件后：

删错了：用以下恢复：

git checkout – file

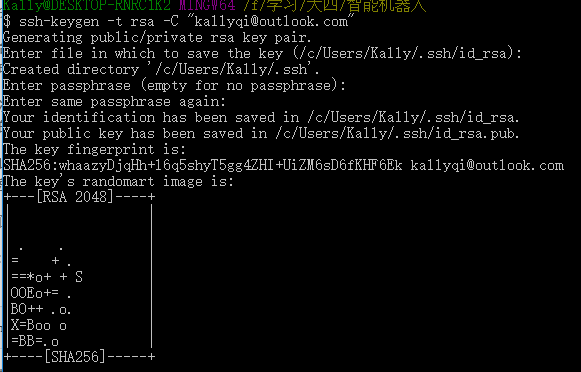
确实要在版本库中删除：

git rm file

远程仓库：

1.ssh密钥生成配置：

ssh-keygen -t rsa -C “Kallyqi@outlook.com”



在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，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。

2.登陆GitHub，打开“Account settings”，“SSH Keys”页面：

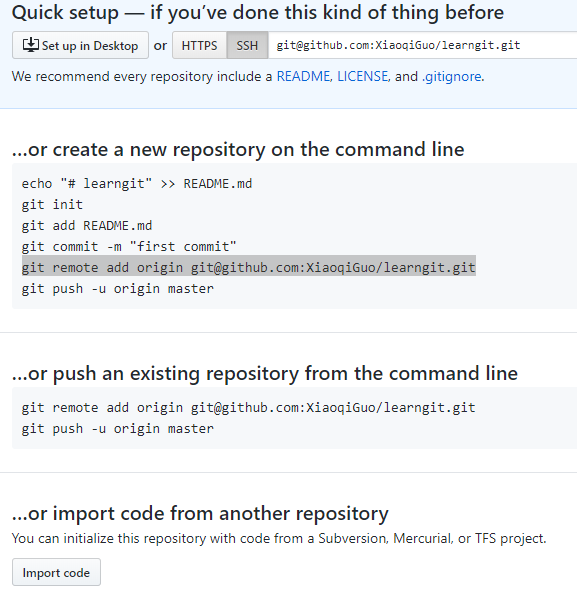
然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容

用国内的coding码云之类同样粘贴id\_rsa.pub文件内容

本地的已有的仓库，再网站上创建一个最好同名的仓库，然后通过git remote add把它和远程库关联，如下：

连接远程已创建的仓库

git remote add origin git@github.com:XiaoqiGuo/learngit.git



把本地的master推到远程仓库用命令：

$ git push -u origin master

如果使用git remote add时报错：

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

fatal: remote origin already exists.

说明本地仓库已经关联了一个名为origin的远程库，可以用

git remote rm origin删除原来关联的库 或者用不同的命名将同一个仓库关联到不同的远程仓库

关联后可用git remote –v 查看远程库信息

例如：

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:michaelliao/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:liaoxuefeng/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

分支管理

创建dev

$ git checkout -b dev

Switched to a new branch 'dev'

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

将当前分支dev转换到master分支：

$ git merge dev

Git鼓励大量使用分支：

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>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

用git log --graph命令可以看到分支合并图。

冲突后：

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下--no-ff方式的git merge：

首先，仍然创建并切换dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

修改readme.txt文件，并提交一个新的commit：

$ git add readme.txt

$ git **commit** -m "add merge"

[dev 6224937] **add** merge

1 file changed, 1 insertion(+)

现在，我们切换回master：

$ git checkout master

Switched to branch 'master'

准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：

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

Merge made by the 'recursive' strategy.

readme.txt | 1 +

1 file changed, 1 insertion(+)

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并后，我们用git log看看分支历史

Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

git stash

git stash list 显示储藏现场

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：

git stash pop 将堆栈现场推出

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：

$ git branch

\* master

现在，你的小伙伴要在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

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

$ git **commit** -m "add /usr/bin/env"

[dev 291bea8] **add** /usr/bin/env

1 file changed, 1 insertion(+)

$ git push origin dev

Counting objects: 5, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 349 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

**To** git@github.com:michaelliao/learngit.git

fc38031..291bea8 dev -> dev

多人协作

Reads: 499205

当你从远程仓库克隆时，实际上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。

要查看远程库的信息，用git remote：

$ git remote

origin

或者，用git remote -v显示更详细的信息：

$ git remote -v

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

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

上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

推送分支

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

$ git push origin master

如果要推送其他分支，比如dev，就改成：

$ git push origin dev

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？

1.master分支是主分支，因此要时刻与远程同步；

2.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；

3.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；

4.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！

抓取分支

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）或者同一台电脑的另一个目录下克隆：

$ git clone git@github.com:michaelliao/learngit.git

Cloning into 'learngit'...

remote: Counting objects: 46, done.

remote: Compressing objects: 100% (26/26), done.

remote: Total 46 (delta 16), reused 45 (delta 15)

Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.

Resolving deltas: 100% (16/16), done.

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：

$ git branch

\* master

现在，你的小伙伴要在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

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

$ git **commit** -m "add /usr/bin/env"

[dev 291bea8] **add** /usr/bin/env

1 file changed, 1 insertion(+)

$ git push origin dev

Counting objects: 5, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 349 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

**To** git@github.com:michaelliao/learngit.git

fc38031..291bea8 dev -> dev

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

$ git add hello.py

$ git **commit** -m "add coding: utf-8"

[dev bd6ae48] **add** coding: utf-8

1 file changed, 1 insertion(+)

$ git push origin dev

**To** git@github.com:michaelliao/learngit.git

! [rejected] dev -> dev (non-fast-forward)

error: failed **to** push **some** refs **to** 'git@github.com:michaelliao/learngit.git'

hint: Updates were rejected because the tip **of** your **current** branch **is** behind

hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

hint: before pushing again.

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

推送失败，因为你的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

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

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

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

From github.com:michaelliao/learngit

fc38031..291bea8 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 dev origin/<branch>

git pull也失败了，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dev和origin/dev的链接：

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

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

再pull：

$ git pull

Auto-merging hello.py

CONFLICT (content): Merge conflict **in** hello.py

Automatic merge failed; fix conflicts **and** **then** commit the result.

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的[解决冲突](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001375840202368c74be33fbd884e71b570f2cc3c0d1dcf000)完全一样。解决后，提交，再push：

$ git **commit** -m "merge & fix hello.py"

[dev adca45d] merge & fix hello.py

$ git push origin dev

Counting objects: 10, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (6/6), 747 bytes, done.

Total 6 (delta 0), reused 0 (delta 0)

**To** git@github.com:michaelliao/learngit.git

291bea8..adca45d dev -> dev

因此，多人协作的工作模式通常是这样：

1首先，可以试图用git push origin branch-name推送自己的修改；

2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；

3没有冲突或者解决掉冲突后，再用git push origin branch-name推送就能成功！

4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 branch-name origin/branch-name。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

标签管理

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

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（跟分支很像对不对？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。

Git有commit，为什么还要引入tag？因为commitid不好记 、找

命令git tag <name>用于新建一个标签，默认为HEAD，也可以指定一个commit id；如用git log 查看历史的commit id 用

git tag <name> <commit\_id>

git tag -a <tagname> -m "blablabla..."可以指定标签信息；

git tag -s <tagname> -m "blablabla..."可以用私钥PGP签名标签；

命令git tag可以查看所有标签。

git show <tagname>可以看标签对应的版本信息（打标签者、日期、说明文字、commitid、作者、日期）

命令git push origin <tagname>可以推送一个本地标签；

命令git push origin --tags可以推送全部未推送过的本地标签；

命令git tag -d <tagname>可以删除一个本地标签；

删除远程仓库的标签先删除本地再用push查看是否针从远程删除

命令git push origin :refs/tags/<tagname>可以删除一个远程标签。

使用GitHub：

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；fork然后git clone 自己账号下的仓库才能修改

可以推送pull request给官方仓库来贡献代码。