git config -– global user.name “your name”

git config –- global user.email “[email@example.com](mailto:email@example.com)”

从本地提交一个全新的项目到github:

1.切换到项目根目录，创建本地仓库，git init （此时会对应创建.git目录，里面包含git版本控制信息）

2.将目录下所有文件添加到本地仓库： git add .

3.提交到本地仓库，添加注释：git commit -m ‘first commit’

4.添加远程仓库地址(需要先在github上创建好，复制地址到这里)，命名为origin:

git remote add origin <https://github.com/Xiaobai0419/minexiaobai0419.git>

5.推送到远程仓库： git push -u origin master

创建版本库：

git init: 把一个目录变成git可以管理的仓库

git status: 查看哪些文件修改过，但还没有提交

git add readme.txt 把文件添加到仓库

git commit -m ‘aaa’ 把文件提交到仓库

版本回退：

git log

418e787abc94d63a7af1315bfed0abea8077cf50 git的commit id

回退到上一个版本： git reset –-hard HEAD^

git reset –-hard commitid 回退到某个版本

git reflog

一旦commitid找不到了，无法回退到某个版本号了。可以使用git reflog:它记录了用户的每一次命令(查看历史命令)。

工作区和暂存区：

工作区：就是在电脑中能看到的目录

版本库(repository)：工作区有一个隐藏目录”.git”, 这个不算工作区，而是git的版本库。Git版本库中存放很多东西，其中最重要的就是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。

git add: 把文件添加到暂存区。 git commit: 把暂存区内的所有内容提交到当前分支。每次修改，如果不add到暂存区，那就不会加入到commit中。

撤销修改(git restore):撤销工作区文件和暂存区文件的修改

git status后发现可以使用命令git restore filename来撤销工作区文件的修改。

撤销暂存区的文件修改：根据git status的提示，先git restore –stage filename(从暂存区撤销修改)，再git restore filename(从工作区撤销修改)

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。

git reset HEAD filename

删除文件：

当不小心删除了暂存区中的一个文件时，需要恢复，此时可以用git restore filename

远程仓库:github

添加远程库:在本地创建了一个git仓库后，又在github上创建一个git仓库，并且让这两个仓库进行远程同步。把本地仓库的内容推送到github仓库。

git remote add origin <https://github.com/Xiaobai0419/minexiaobai0419.git>(本地和远程关联)

添加后，远程仓库的名字就是origin,这是git默认 的叫法，也可以改成别的。

下一步，就是把本地库的所有内容推送到远程库上：git push -u origin master(第一次推送master分支时，加-u参数)。Git 不仅把本地的master分支推送到远程新的master分支，还会把本地master分支和远程master分支关联起来，在以后推送或者拉取可以简化命令。

分支管理：

创建与合并分支：

创建分支: git checkout -b dev (创建dev分支) 或者git branch dev git checkout dev

创建分支: git branch dev

查看分支: git branch -a (本地和远程分支)

创建+切换分支: git checkout -b dev

合并某分支到当前分支: git merge dev (先切换到当前分支，在当前分支中git merge dev.把dev分支合并到当前分支)

删除分支: git branch -d dev 最后删除没用的分支

分支改名:

还没推送到远端：git branch -m oldname newname

已经推送到远端：git branch -m oldname newnam ; 删除远程分支: git push –delete origin oldname ;上传新命名的本地分支: git push origin newname

解决冲突：手动解决。当无法自动合并时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

用带参数的git log查看分支的合并情况：git log –-graph –-pretty=oneline (git log -–graph命令可以看到分支合并图)

分支管理策略：

在实际开发中，分支管理的基本原则：

1.master分支应该非常稳定，仅用来发布新版本，平时不能在上面干活。

2. 干活都在dev分支上，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。

3.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。

Bug分支：每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

git stach功能：把现场保存起来，等以后恢复现场后继续工作。 使用git stash，工作区干净，此时可以创建分支来修复bug

bug分支管理：当手头工作没完成时，先把工作现场git stash一下，然后去修复bug,修复后，再git stash pop,回到工作现场。

在master分支上创建分支修复bug,首先需要切换到master分支上(git checkout master)。然后创建修复分支(git branch issue-101),切换到该分支上(git checkout issue-101)

当git stash后，使用git status发现工作区是干净的。此时, git stash list。发现：stash@{0}:WIP on ……。工作现场还在，git把stash内容存放起来了，需要恢复一下。使用git sash pop恢复现场(此时发现git stash list内容为空)。

Feature分支：

每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。 如果要丢弃一个没被合并过的分支，可以通过git branch -D featurename强行删除。

多人协作：当从远程仓库克隆时，实际上git自动把本地的master分支和远程的master分支对应起来，并且，远程仓库的默认名称是origin。查看远程仓库的信息：git remote (git remote -v详细信息)

git remote –v

origin <https://github.com/jooney/notes.git(fetch)>

origin https://github.com/jooney/notes.git(push)

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了。

git push origin master 把本地的master分支推送的远地origin仓库的master分支上。

git push origin dev 把本地的dev分支推送到远地仓库的dev分支上。

但是并非一定要把本地分支往远程推送。一般情况：master分支是主分支，因此需要时刻与远程同步。dev分支是开发分支，所有成员需要在上面工作，所以也要与远程同步。Bug分支只用于在本地修复bug，没必要推送到远程。同理feature分支也没必要推送到远程。

抓取分支：

直接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。应该设置dev和origin/dev的链接：

git branch --set-upstream dev origin/dev ，再pull

多人协作的工作模式通常是这样的：

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

如果推送失败，则因为远程分支比本地的分支更新，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 branch-name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，使用命令git branch –set-upstream branch-name origin/branch-name

标签管理：

发布一个版本时，通常在版本库中打一个标签，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的标签其实就是指向某个commit的指针，所以创建和删除标签都是瞬间完成的。

创建标签：

切换到需要打标签的分支上：git checkout master git tag v1.0

使用git tag查看所有的标签。

默认标签是打在最新提交的commit上的。想要在历史commit上打标签：

找到历史提交的commit id,然后打上就可以。

git log –pretty=oneline (找到commit id)

打标签: git tag v0.9 be6a911

创建带有说明的标签，用-a指定标签名，-m指定说明文字:

git tag -a v0.1 -m “version 0.1 released” 3628164

可以用git show tagname 查看标签完整信息

操作标签： 如果标签打错了，可以删除标签

git tag -d v0.1 创建的标签都只会存储在本地，不会自动推送到远程。打错的标签可以在本地安全删除

git merge和 git rebase区别

git merge b 将b分支合并到当前分支

同理 git rebase b 也是将b分支合并到当前分支

假设有2个分支：master分支，从master分支签出来的dev分支。两个分支各自做了一些修改，并且做了提交。两个分支各自前进了，它们之间有了”分叉”。使用 pull命令把master分支上的修改拉下来并且和dev分支修改合并，结果看起来就像一个新的合并的提交。

但是，如果想让dev分支历史看起来像没有经过任何合并一样，可以用git rebase(git checkout dev git rebase master)

git rebase原理: 把dev分支里的每个提交(commit)取消掉，并且把它们临时保存为补丁(patch,这些补丁存放到.git/rebase目录中)，然后把dev分支更新为最新的master分支，最后把保存的这些补丁应用到dev分支上。 当dev分支更新之后，它会指向最新的提交(commit),而那些老的提交会被丢弃。

解决冲突：在rebase的过程中，也许会出现冲突(conflict)。此时，git会停止rebase并会让你去解决冲突。在解决完冲突后，用git add命令去更新这些内容的索引,然后，无需执行git commit,只要执行git rebase --continue。这样git会继续应用(apply)余下的补丁。在任何时候，可以用—abort参数来终止rebase的行动，并且dev分支会回到rebase开始前的状态。(git rebase --abort)

当尝试合并公共分支到当前分支时：

rebase会把当前分支的commit放到公共分支的最后面(把当前分支之前的commit取消掉，作为patch)，把当前分支替换为公共分支，最后把patch(当前分支的几次commit)添加到公共分支(公共分支上的commit)后面。查看git log --graph，不会分叉(因为，patch的commit紧跟在公共分支上的commit之后)。

merge会把公共分支和当前自己分支上的commit合并在一起(两路分叉)，形成一个新的commit。

使用git [clone@github.com:\*\*\*.git方式下载github](mailto:clone@github.com:***.git方式下载github)代码至本地时需要依赖ssh key,遇到权限不足问题一般都是ssh key失效或者ssh key不存在。需要重新创建ssh key.

检查本地ssh key是否存在：ls ~/.ssh

若key不存在，则生成ssh key: ssh-keygen -t rsa -C “自己的邮箱”。之后生成id\_rsa(私钥)，id\_rsa.pub(公钥)

登入github,打开Account settings, SSH Keys页面，点Add SSH Key,填上任意Title,在Key文本框中粘贴id\_rsa.pub文件的内容。

Git cherry-pick

Git merge -I HEAD^n

git fork: 已经有一个中心仓库(master)。开发时每个人需要分别fork一份代码到自己的远程仓库(repository), 然后clone到本地。在自己的本地和自己的远程仓库中进行开发和commit，然后和自己的远程仓库进行push, pull, fetch操作。 需要提交时，发起一个合并请求(pull request或merge request)。