基础命令

用户设置

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

初始化仓库

```
$ git init
Initialized empty Git repository in /Users/learngit/.git/
```

把文件添加到仓库

```
git add filename
```

把文件提交到仓库

```
git commit -m "comment message"
```

初始化一个Git仓库，使用git init命令。

添加文件到Git仓库，分两步：

- 第一步，使用命令git add ，注意，可反复多次使用，添加多个文件；
- 第二步，使用命令git commit，完成。

运行*git status*命令看看结果

```
git status
```

但如果能看看具体修改了什么内容

```
git diff filename
```

- 要随时掌握工作区的状态，使用git status命令。
- 如果git status告诉你有文件被修改过，用git diff可以查看修改内容

版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用git log命令查看

```
git log

git log --pretty=oneline
```

首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交,上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100.

```
git reset --hard HEAD^

git reset --hard 3628164
```

Git提供了一个命令*git reflog*用来查看历史

```
git reflog
```

- HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令*git reset --hard commit_id*。
- 穿梭前，用*git log*可以查看提交历史，以便确定要回退到哪个版本。

- 要重返未来，用*git reflog*查看命令历史，以便确定要回到未来的哪个版本。

工作区（Working Directory）：就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区。

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。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用“git add”把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用“git commit”提交更改，实际上就是把暂存区的所有内容提交到当前分支。

*git checkout -- file*可以丢弃工作区的修改

```
git checkout -- filename
```

*git checkout -- file*命令中的“--”很重要，没有“--”，就变成了“创建一个新分支”的命令.

用命令*git reset HEAD file*可以把暂存区的修改撤销掉（unstage），重新放回工作区

```
git reset HEAD  filename
```

*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，表示最新的版本。

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。

确实要从版本库中删除该文件，那就用命令*git rm*删掉，并且commit

```
git rm test.txt

git commit -m "remove test.txt"
```

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本

```
git checkout -- test.txt
```

*git checkout*其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

命令*git rm*用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失**最近一次提交后你修改的内容**。

远程仓库

要关联一个远程库，使用命令*git remote add origin git@server-name:path/repo-name.git*；

关联后，使用命令*git push -u origin master*第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令*git push origin master*推送最新修改；

用命令*git clone*克隆一个本地库

```
//Git本身的源代码你既可以用 git:// 协议来访问:
git clone  git://git.kernel.org/pub/scm/git/git.git
//也可以通过http 协议来访问:
git clone http://www.kernel.org/pub/scm/git/git.git
```

git 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*

在 当前分支下，要把dev分支的内容合并

```
git merge dev
```

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

用*git log --graph*命令可以看到分支合并图。

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。

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
```

现在，用*git status*查看工作区，就是干净的（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。

首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：

```
//转换到master分支
git checkout master
//创建并转换到的名字为 issue-101 的分支
git checkout -b issue-101
```

查看存储的工作 用git stash list

```
git stash list
```

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：

修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；

当手头工作没有完成时，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.

如果要丢弃一个没有被合并过的分支，可以通过*git branch -D name*强行删除。

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*或者用git remote -v显示更详细的信息：

```
git remote

git remote -v
```

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？

- 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！

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://git.kernel.org/pub/scm/git/git.git
```

指定本地dev分支与远程origin/dev分支的链接

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

git pull
```

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；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 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，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。

- 查看远程库信息，使用*git remote -v*；
- 本地新建的分支如果不推送到远程，对其他人就是不可见的；
- 从本地推送分支，使用*git push origin branch-name*，如果推送失败，先用*git pull*抓取远程的新提交；
- 在本地创建和远程分支对应的分支，使用*git checkout -b branch-name origin/branch-name*，本地和远程分支的名称最好一致；
- 建立本地分支和远程分支的关联，使用*git branch --set-upstream branch-name origin/branch-name*；
- 从远程抓取分支，使用*git pull*，如果有冲突，要先处理冲突。

tag 标签

命令*git tag name*就可以打一个新标签，可以用命令*git tag*查看所有标签

```
//新建标签
git tag v1.0

//给commit id 为25656e2的历史版本打标签
git tag v1.0  25656e2

//查看标签
git tag
```

用*git show tagname*查看标签信息

```
git show v1.0
```

- 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；
- -a tagname -m "blablabla..."可以指定标签信息；
- -s tagname -m "blablabla..."可以用PGP签名标签；
- 命令*git tag*可以查看所有标签；

推送某个标签到远程，使用命令git push origin tagname，或者，一次性推送全部尚未推送到远程的本地标签

```
git push origin v1.0


git push origin --tags
```

删除标签

分两步，1、删除本地；2、删除远程。

```
//删除本地
git tag -d v0.9
//删除远程
git push origin :refs/tags/v0.9
```

- 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；
- 命令*git push origin :refs/tags/tagname*可以删除一个远程标签。

ignore 文件

不需要从头写*.gitignore*文件，GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：https://github.com/github/gitignore

忽略文件的原则是：

1. 忽略操作系统自动生成的文件，比如缩略图等；
2. 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；
3. 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。

配置别名

如果敲*git st*就表示*git status*

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.br branch
```

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