## Git的安装

官网直下 <https://git-scm.com/downloads>

## 自报家门

配置全局用户和邮箱

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

## 创建版本库以及写入文件

1.

$ mkdir learngit

$ cd learngit

$ pwd

Learngit即是文件夹名称

2.

$ git init

将文件夹learngit变成git可以管理的仓库

3.

$ vi readme.txt

创建以及写入文件

4.

ESC shift+zz

Windows系统退出打开文件

5.

$ git add readme.txt

$ git commit -m “\*\*\*\*”

提交更改以及注释

## 4.对文件的的各种更改以及指令

$ git status（查看状态）

$ git log（查看各版本）

$ git reset --hard HEAD//HEAD~n(回退到N版本前)

$git checkout --readme.txt（工作区撤销修改）

$ cat readme.txt(查看内容)

$ git reset HEAD readme.txt（撤销在暂存区的修改）

$ rm test.txt(删除文件)

1{

$ git rm test.txt

rm ‘test.txt’

$ git commit -m “”

}

2{

$git checkout --test.txt

}

## 5.远程仓库的创建即与github的联系

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

创建ssh密钥

1. 打开github setting ssh keys add ssh key

将id\_rsa.pub里内容复制过去可添加密钥

（如果无法使用那么在.ssh文件夹里面创建config文件写入

Host github.com

     HostName ssh.github.com

     User git

     Port 443

）（即将22端口改成443端口）

## 添加以及克隆远程库

### 1.添加

1. 在github上创建Repositories 用learngit来命名（方便后续操作）
2. 在 git bash上写

$ git remote add origin git@github.com:\*\*\*/learngit.git

（\*\*\*是github的用户名）

$ git push -u origin master

Counting objects: 20, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.

Total 20 (delta 5), reused 0 (delta 0)

remote: Resolving deltas: 100% (5/5), done.

To github.com:michaelliao/learngit.git

\* [new branch] master -> master

Branch 'master' **set** up **to** track remote branch 'master' **from** 'origin'.

将本地库内容推送到远程库

或者

1.git remote add origin [https://github.com/](https://github.com/" \t "https://www.liaoxuefeng.com/wiki/896043488029600/_blank)自己的github账户名/learn-git.git

2. git branch -M main

3.git push -u origin main

{删除远程库

$ git remote -v（查看远程库信息）

$ git remote rm origin（删除origin）

}ps：此处删除指删除绑定关系

### 克隆

在创建repository的时候创建个README.md 文件，以便看出克隆

$ git clone [git@github.com:\*\*\*/gitskills.git](mailto:git@github.com:***/gitskills.git’)

Cloning into 'gitskills'...remote: Counting objects: 3, done.remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 3Receiving objects: 100% (3/3), done.

克隆一个本地库

$ cd gitskills

$ ls

README.md

进入gitskills文件查看目录

## 分支管理

### 1.一个分支

1我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

git branch命令会列出所有分支，当前分支前面会标一个\*号。

2.在dev分支上提交，eg:readme.txt做个修改，加上一行：

Creating a **new** branch is quick.

然后提交：

$ git add readme.txt

$ git commit -m "branch test"

[dev b17d20e] branch test

1 file changed, 1 insertion(+)

现在，dev分支的工作完成，我们就可以切换回master分支：

$ git checkout master

即：只在dev分支上改变master上不变

3.

$ git merge dev

合并指定分支到当前分支。合并后，再看readme.txt的内容，此时更改

4删除dev分支

$ git branch -d devDeleted branch dev (was b17d20e).

查看branch，就只剩下master分支：

$ git branch

\* master

Ps:  
创建并切换到新的dev分支，也可以使用：

$ git switch -c dev

直接切换到已有的master分支，可以使用：

$ git switch master

### 多个分支

当多个分支一起修改时产生冲突

直接查看readme.txt的内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.<<<<<<< HEADCreating a new branch is quick & simple.

=======Creating a new branch is quick AND simple.

>>>>>>> feature1

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容

修改如下后保存：

Creating a **new** branch is quick **and** simple.

再提交

（用带参数的git log也可以看到分支的合并情况：

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

最后，删除feature1分支：

$ git branch -d feature1Deleted branch feature1 (was 14096d0).

）

手动解决冲突后再合并

### 3.分支策略

合并分支时git采取模式“Fast forward”不好容易丢失分支信息，需要强制禁止Fast forward

首先，创建并切换dev分支：

$ git **switch** -c 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"

然后，切换回master：

$ git switch masterSwitched 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

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

分支策略基本原则

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；

干活都在不稳定的dev分支上，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；

### Bug分支

通过新的临时分支修复bug

1.

$ git stash

Saved working directory **and** index state WIP on dev: f52c633 add merge

当前工作储藏，即：用git status 看很干净，然后再创建分支修复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：

$ git checkout master

$ git checkout -b issue-101

$ git add readme.txt

$ git commit -m "fix bug 101"

修复完成后，切换到master分支，并完成合并，最后删除issue-101分支：

$ git switch master

现在，回到dev分支

$ git switch dev

$ git status

用git stash list命令看看工作现场

$ 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，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

再用git stash list查看，就看不到任何stash内容了：

$ git stash list

可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：

$ git stash apply stash@{0}

同样的bug，要在dev上修复，我们只需要把4c805e2 fix bug 101这个提交所做的修改“复制”到dev分支。注意：我们只想复制4c805e2 fix bug 101这个提交所做的修改，并不是把整个master分支merge过来。

为了方便操作，Git专门提供了一个cherry-pick命令，让我们能复制一个特定的提交到当前分支：

$ git branch

\* dev

Master

$ git cherry-pick 4c805e2

Git自动给dev分支做了一次提交，注意这次提交的commit是1d4b803，它并不同于master的4c805e2，因为这两个commit只是改动相同，但确实是两个不同的commit。用git cherry-pick，我们就不需要在dev分支上手动再把修bug的过程重复一遍。

### 5.Feature分支

强行删除一个未被合并的分支：

$ git branch -D feature-vulcan

### 6.多人协作

从远程仓库克隆时，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

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

当多人推送冲突

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

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dev和origin/dev的链接：

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

再pull：

$ git pull

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/896043488029600/900004111093344" \t "https://www.liaoxuefeng.com/wiki/896043488029600/_blank)完全一样。解决后，提交，再push：

$ git **commit** -m "fix env conflict"

$ git push origin dev

### 7.Rebase

$ git rebase

原本分叉的提交现在变成一条直线了！

原理：Git把本地的提交“挪动”了位置，放到了f005ed4 (origin/master) set exit=1之后，这样，整个提交历史就成了一条直线。rebase操作前后，最终的提交内容是一致的，但是，本地的commit修改内容已经变化了，它们的修改不再基于d1be385 init hello，而是基于f005ed4 (origin/master) set exit=1，但最后的提交7e61ed4内容是一致的。

这就是rebase操作的特点：把分叉的提交历史“整理”成一条直线，看上去更直观。缺点是本地的分叉提交已经被修改过了。

最后，通过push操作把本地分支推送到远程：

## 标签管理

1. 创建标签

在Git中打标签非常简单，首先，切换到需要打标签的分支上：

$ git branch

$ git checkout master

然后，敲命令git tag <name>就可以打一个新标签：

$ git tag v1.0

可以用命令git tag查看所有标签：

$ git tag

v1.0

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

找到历史提交的commit id，然后打上忘打的标签

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

比方说要对add merge这次提交打标签，它对应的commit id是f52c633，敲入命令：

$ git tag v0.9 f52c633

再用命令git tag查看标签：

$ git tag

v0.9

v1.0

标签是按字母排序的。可以用git show <tagname>查看标签信息：

$ git **show** v0.9

可以看到，v0.9确实打在add merge这次提交上。

还可以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" 1094adb

用命令git show <tagname>可以看到说明文字：

$ git show v0.1

version 0.1 released

### 操作标签

删除：

$ git tag -d v0.1

推送某个标签到远程，使用命令git push origin <tagname>：

$ git push origin v1.0

或者，一次性推送全部尚未推送到远程的本地标签：

$ git push origin --tags

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

$ git tag -d v0.9

然后，从远程删除。删除命令也是push，但是格式如下：

$ git push origin :refs/tags/v0.9

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

# 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；

可以推送pull request给官方仓库来贡献代码。

# 自定义git

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

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**

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**

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。

还有些时候，当我们编写了规则排除了部分文件时：

*# 排除所有.开头的隐藏文件:*

.\**# 排除所有.class文件:*

\*.**class**

但是我们发现.\*这个规则把.gitignore也排除了，并且App.class需要被添加到版本库，但是被\*.class规则排除了。

虽然可以用git add -f强制添加进去，但有强迫症的童鞋还是希望不要破坏.gitignore规则，这个时候，可以添加两条例外规则：

*# 排除所有.开头的隐藏文件:*

.\**# 排除所有.class文件:*

\*.**class***# 不排除.gitignore和App.class:*

!.gitignore

!App.**class**

把指定文件排除在.gitignore规则外的写法就是!+文件名，所以，只需把例外文件添加进去即可。

可以通过[GitIgnore Online Generator](https://michaelliao.github.io/gitignore-online-generator/" \t "https://www.liaoxuefeng.com/wiki/896043488029600/_blank)在线生成.gitignore文件并直接下载。

.gitignore文件放Git仓库根目录下，但其实一个Git仓库也可以有多个.gitignore文件，.gitignore文件放在哪个目录下，就对哪个目录（包括子目录）起作用。

配置别名

$ git config --global **alias**.st status

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。