Git介绍

Git是目前最先进的分布式版本控制系统

那什么是版本控制系统?

> 如果有一个软件，不但能自动帮我记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆类似的文件了，也不需要把文件传来传去。如果想查看某次改动，只需要在软件里瞄一眼就可以

这就是git的初衷

集中式 vs 分布式

集中式版本控制系统是将版本库存放在中央服务器，每次干活之前和提交修改之后都要和中央服务器同步数据，所以必须联网才能工作，受网速限制。而且如果中央服务器坏了，版本库就没了

分布式版本控制系统没有中央服务器，每个人的电脑上都有完整的版本库，不需要联网就能工作（但是会有一台电脑充当“中央服务器”，仅仅是用来方便交换大家的修改），也不用担心某个版本库出问题。分步式系统更强大的地方还在于它的分支管理

安装Git

- 安装homebrew 然后通过homebrew安装Git 文档
- 安装Xcode，然后运行Xcode 选择菜单“Xcode - Preferences - Downloads - Command Line Tools”

创建版本库

任意一个位置执行:

```
$ mkdir learngit
```

learngit文件夹用来存放git仓库

通过`git init`把这个目录变成Git可以管理的仓库

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

此时这个文件夹就是一个空的git仓库 文件夹下会多一个.git 隐藏文件，里面保存的是各种仓库信息，不要随便改动

版本控制系统只能跟踪文本文件的改动，比如TXT，代码等，无法跟踪二进制文件的改动，比如图片、视频、Word文件

对新建文件的操作

将新建的某个文件添加到仓库：

```
$ git add xxx
```

xxx代表添加的文件名

将所有新建的文件添加到仓库

```
$ git add .
```

将文件提交到仓库

```
$ git commit -m "xxxxx"
```

-m后边 双引号中的内容为提交的注释 会展现在历史记录中

`commit`命令可以一次提交很多文件，所以你可以多次add不同的文件之后再执行一次`commit`命令提交所有文件

远程仓库

关联远程仓库

在GitHub（或其他的代码托管平台）创建一个仓库，然后将本地的仓库与GitHub上的仓库关联

```
$ git remote add origin git@github.com:xxx/xxx.git
```

add origin后面的是GitHub上的仓库地址，可以使HTTPS协议或者git协议（需要在代码托管平台添加自己电脑的SSH Key，方法很简单）

然后将本地仓库所有的内容推送到远程仓库上

```
$ git push -u origin master
```

如果出现类似下方这样的提示，表示关联成功了

```
Enumerating objects: 73, done.
Counting objects: 100% (73/73), done.
Delta compression using up to 4 threads
Compressing objects: 100% (62/62), done.
Writing objects: 100% (72/72), 690.35 KiB | 1.98 MiB/s, done.
Total 72 (delta 12), reused 0 (delta 0)
remote: Resolving deltas: 100% (12/12), done.
To github.com:zhenguanqing/DataStructureAndAlgorithm.git
   04a7d8a..81fe63e  master -> master
Branch 'master' set up to track remote branch 'master' from
'origin'.
```

注：开始这一步总是失败，原因是创建的远程仓库必须为一个纯空仓库，任何文件都不能有，否则git会认为这是两个分支，解决方法是先把远程仓库的内容更新到本地：

```
git pull origin master
```

如果报错 `fatal: refusing to merge unrelated histories` 可以使用

```
git pull --allow-unrelated-histories origin  master
```

来达到更新的目的

`--allow-unrelated-histories`会允许关联两个分支的历史分支

从远程仓库克隆

关联远程仓库还可以先创建远程库，然后克隆到本地，这种方式更常使用，因为有可能你是从半路开始开发这个项目的，此时这个项目已经存在了，当然也可以先在代码托管平台上创建远程仓库然后克隆到本地从零开始开发

此时拿到远程仓库地址后使用命令`git clone`克隆一个仓库（在任意一个文件夹目录下执行都可以，克隆成功后本地仓库就存在于这个文件夹下）

```
$ git clone git@github.com:xxx/xxx.git
```

例如：

```
$ git clone
git@github.com:zhenguanqing/DataStructureAndAlgorithm.git
Cloning into 'DataStructureAndAlgorithm'...
remote: Enumerating objects: 79, done.
remote: Counting objects: 100% (79/79), done.
remote: Compressing objects: 100% (56/56), done.
remote: Total 79 (delta 15), reused 72 (delta 12), pack-reused 0
Receiving objects: 100% (79/79), 691.97 KiB | 117.00 KiB/s, done.
Resolving deltas: 100% (15/15), done.
```

版本与修改

版本回退

随着在工作中对文件不停的修改，然后提交修改到版本库中，版本库会存在很多个`commit`，每次提交生成一个，可以把每次阶段性的修改放到一个`commit`中，如果出现错误，就可以从最近的一个`commit`恢复

通过`git log`可以查看我们提交修改的历史记录（从近到远），加上`--pretty=oneline`可以展示精简信息

历时记录信息中类似 `1094adb...` 的是`commit id` （版本号）

如果回退到上一个版本?

在Git中 `HEAD`表示当前版本，上一个版本就是`HEAD^`，上上个版本就是`HEAD^^`，往回的版本比较多的时候写成`HEAD~100`

使用`git reset`命令 可以退回某一个版本

退回上一个版本：

```
$ git reset --hard HEAD^
```

退回上个之后还想重新回到最新的版本怎么办？找到刚才的最后一次commit记录中的commitid，然后执行

```
$ git reset --hard 1094a
```

`1094a` 代表的是最后一个版本的 commitid （不需要写全，写出前几位就可以）

如果已经找不到最后一个版本的commitid怎么办？比如重启了电脑

可以使用`git reflog`命令

Git的版本回退速度很快，是因为Git内部有一个当前版本的HEAD指针

区别：

- git log - 查看提交历史 以便确定回退到哪个版本
- git reflog - 查看命令历史 以便确定要回到未来的哪个版本

工作区和暂存区

工作区（Working Directory）： 电脑本地存放Git仓库的文件夹就是一个工作区

版本库（Repository）： 工作区内有一个隐藏文件夹`.git`这个文件夹不算工作区，而是Git的版本库
版本库中存放了很多东西，最重要的就是暂存区（stage，或者叫index），还有Git为我们自动创建的第一个分支`master`，以及`master`的第一个指针`HEAD`

当我们把文件添加到版本库中时，先是通过`git add`将文件添加到暂存区，然后用`git commit`将当前暂存区的所有内容提交到当前分支，一旦提交后，如果对工作区没有做任何修改，那么工作区就是干净的

- git status 查看当时工作区的状态

管理修改

每次修改，如果不使用`git add`将修改添加到暂存区，那么`commit`的时候就不会将这部分修改提交到版本库，所以Git跟踪管理的是修改，而不是文件，这也正是它优秀的地方

> git diff到底比较的是那个两个文件之间的差异。经过在网上搜网，终于找到乐答案。 这里分为两种情况，一种是当暂存区中有文件时，另一种是暂存区中没有文件。 （1）当暂存区中没有文件时，git diff比较的是，工作区中的文件与上次提交到版本库中的文件。 （2）当暂存区中有文件时，git diff则比较的是，当前工作区中的文件与暂存区中的文件。 而 git diff HEAD -- file，比较的是工作区中的文件与版本库中文件的差异。HEAD指向的是版本库中的当前版本，而file指的是当前工作区中的文件。 补充：git diff命令比较的是工作目录中当前文件与暂存区快照之间的差异，也就是修改之后还没有暂存起来的变化内容。 注意：git diff本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。所以，有时候你一下子暂存了所有更新过的文件后，运行git diff后却什么也没有，就是这个原因。 如果要查看已暂存的将要添加到下次提交里的内容，可以使用git diff --cached或者git diff --staged。

撤销修改

如果对文件的修改还没有`git add`到暂存区，可以使用`git checkout --xxx`将修改撤销，让这个文件回到最近一次`git commit`或者`git add`时的状态，`--`很重要 如果没有的话就变成了“切换到另一个分支”的命令

撤销未加入暂存区的所有修改使用`git checkout .`

如果对某些修改已经`git add`到暂存区了，就需要使用`git reset HEAD<file>`可以把暂存区的修改撤销掉，使用`git reset HEAD -- .`可以撤销所有暂存区的修改

如果修改已经commit，再想退回的话就需要使用上一章的版本退回方案来撤销修改了，如果修改不紧commit而且已经push到远程仓库 那就没办法了

分支管理

分支可以将工作中的任务进行更好地拆分，让我们更好地兼顾不同的工作任务，避免不同任务之间互相影响进度。

创建新分支（例如dev）并切换到这个分支

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

`git checkout` 加上 `-b` 表示创建新分支并切换，相当于两条命令

```
$ git branch dev
$ git checkout dev
Switched to branch 'dev'
```

使用 `git branch` 命令查看当前分支

```
$ git branch
* dev
  master
```

当我们在 `dev` 上完成了某一阶段的工作后，就可以切换回 `master` 分支，并把 `dev` 上修改的内容合并进 `master` 分支

```
$ git merge dev
```

此时就可以删除 `dev` 分支了

```
$ git branch -d dev
```

解决冲突

当两个不同的分支对同一个文件作出修改后，再合并其中一个分支到另一个分支，就有可能出现冲突，比如

```
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

此时需要根据提示找到有冲突的文件（可以使用`git status`查看），冲突解决后再次提交就彻底完成了合并

使用带参数的`git log`可以看到分支的合并情况

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

使用`git log --graph`可以看到分支合并图

分支策略

合并分支时，加上`--no-ff`参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而`fast forward`合并就看不出来曾经做过合并。

使用`--no-ff`合并分支时候会生成一个新的commit，所以要加上`-m`添加commit说明

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

> 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。

Bug分支与储存功能

如果在开发过程中需要切换到另一个分支去做一些任务（比如一个软件的线上版本出现了问题，需要新建一个fix分支去修复这些问题，优先级比较高），但是当前的分支任务还未完成，没办法提交，就可以提供Git提供的`stash`功能，可以把当前工作“储藏”起来，等fix分支的工作完成后再切换回来继续工作

```
$ git stash
Saved working directory and index state WIP on master: 8b63070
merge
```

此时用`git status`查看工作区，是干净的

此时我们可以切换到其他分支去完成修复工作，完成后再切换回刚才正在开发的分支，利用`git stash list`命令查看

```
$ git status
stash@{0}: WIP on master: 8b63070 merge
```

说明刚才存储的工作内容还在，需要恢复一下：

一是用`git stash apply`恢复，但是恢复后刚才存储的内容不删除，需要用`git stash drop`来删除

另一种是用`git stash pop`可以在恢复的同时把刚才储存的内容删除

```
$ git stash pop
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (31fe0b99cb0672f72bd995c56806ce464b37491c)
```

如果在开发过程中产生了多个“储存”，在恢复的时候，可以先利用`git stash list`查看储存列表，然后利用`git stash apply stash@{0}`恢复某一个储存，恢复后利用`git stash drop stash@{0}`来将刚才恢复这个储存删除

```
$ git stash list
stash@{0}: WIP on master: 2f7013a 0
stash@{1}: WIP on master: 2f7013a 0
```

```
$ git stash apply stash@{0}
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git stash drop stash@{0}
Dropped stash@{0} (a18304ccf7ba4530441db39ab448d9494ad89d64)
```

由于现在的分支是从master分支上分出来的，那master分支上的bug在当前的分支也会存在，那我们怎样快速地在当前的分支也将这个bug也修复呢?

我们只需要将修复分支的那次提交复制到当前的分支就可以（不需要将整个master分支合并过来）

```
$ git cherry-pick 53f63c9
[dev1 dbce2f8] fix
 Date: Fri Aug 16 11:38:40 2019 +0800
 1 file changed, 2 insertions(+)
```

此时Git会自动给dev1做一次提交，虽然跟 `53f63c9` 的改动相同，但他们属于两个不同的commit

Feature分支

为软件添加一个新功能可以新建一个feature分支，来实现一些实验性的代码，开发完成后合并到主分支并删除feature分支

```
git checkout -b branch_name
```

正常情况下，功能性分支开发完成后，就可以合并进开发分支了，合并完成没有后就可以删除刚才的分支

但如果我们需求突然有变动，这个功能不要了，而且要永久性删除

```
git branch -d branch_name
```

此时，Git会提示我们销毁失败，因为这个分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用大写的`-D`参数。。

```
it branch -D branch_name
```

多人协作

查看远程仓库的信息可以使用

```
git remote
```

或使用`git remote -v`显示更详细的分支

推送分支，就是把该分支上所有的的本地提交推送到远程仓库，比如：

```
git push origin dev
```

理论上，`master`是主分支，要时刻与远程同步，`dev`是主开发分支，所有成员都要在上面工作，所以也需要与远程同步； bug分支只需要在本地修复bug，就没必要推送到远端了，而feature分支要看你的伙伴是否需要在这个分支上开发，否则可以不用推送

如果你的同伴对某个分支做了修改并推送到了远端，此时你也做了修改，是无法直接推送的，需要先`git pull`将远端的内容更新到本地并合并，然后再推送。

如果本地的分支与远程的分支没有链接的话，需要用 `git branch --set-upstream-to=origin/branch_name branch_name` 建立分支的关联

变基

不同的人开发同一个分支会导致提交历史分叉，这时候rebase就派上了用场

```
git rebase
```

原本分叉的分支会变成一条直线（但是本地的分叉提交会被修改）

标签管理

在发布一个版本时，我们通常会在版本库中打一个标签（tag），这样就可以确定这个版本的时刻，标签就是这个版本的快照。但本质上他是指向某个commit的指针，相对于commitid，它的可读性更好

使用 `git tag tag_name` 就可以打一个标签。

使用 `git tag` 可以查看所有标签

如果要给版本库以前的某个时刻打标签可以使用 `git tag tag_name commitid`

注意：标签不是按时间顺序列出的，而是按字母排列的，使用 `git show tag_name` 可以查看标签信息

还可以使用 `git tag -a tag_name -m'message' commitid` 创建带有说明的标签，-a 用来指定标签名 -m用来添加说明

删除本地标签： `git tag -d tag_name` 推送某个标签到远端： `git push origin tag_name` 或使用 `git push origin --tags` 推送所有本地标签 删除远端标签： `git push origin:refs/tags/tag_name`

使用gitee和github

如果本地已有一个git仓库，可以在码云或者github上新建一个远程仓库，然后将本地的仓库和远程的那个仓库关联起来

```
git remote add origin git@gitee.com:zhenguanqing/LearnGit.git
```

还可以使同一个本地仓库关联两个不同的远程仓库

```
git remote add origin git@github.com:zhenguanqing/LearnGit.git
```

此时这个仓库就关联了两个仓库，使用`git remote -v`查看远程仓库信息：

```
gitee   git@gitee.com:zhenguanqing/LearnGit.git (fetch)
gitee   git@gitee.com:zhenguanqing/LearnGit.git (push)
github  git@github.com:zhenguanqing/LearnGit.git (fetch)
github  git@github.com:zhenguanqing/LearnGit.git (push)
```

此时如果需要推送改动需要指定往哪个远程仓库推送，比如

```
git push origin master
```

```
git push github master
```

解决failed to push some refs to git

原因可能是远程仓库中的内容没有更新到本地

可以使用`git pull --rebase origin master`

然后再`git push origin maseter`

自定义Git

忽略特殊文件

在工作过程中有一些文件和内容时没有必要或者不能推送到远端的，比如机密文件、操作系统自动生成的文件、编译产生的文件等等，我们只需要在Git工作区的根目录下创建一个`.gitignore`文件，然后把需要忽略的文件名称填进去，Git就会自动忽略这些文件

如果有时候我们想添加一个文件却发现添加不了，可能是这个文件被`.gitignore`忽略了，此时我们可以用`-f`强制添加到Git:

```
git add -f README.md
```

如果你不想要这个规则了，可以使用`git check-ignore`检查到底是哪一行规则忽略了这个文件

```
git check-ignore -v README.md
```

配置别名

可以使用`git config --global alias.x xxx`为Git命令配置别名

例如

```
git config --global alias.st status
```

之后查看状态的命令就可以用

```
git st
```

等同于

```
git status
```

配置文件

配置Git的时候`--global`是针对当前的用户起作用，如果不加，就是只针对当前的仓库

Git的配置文件放在`.git/config`文件中