Git

分布式版本控制系统

Git的三棵树：工作区、暂存区、版本库（代码仓库）

Git 的诞生

林纳斯·托瓦兹在1991年创建了开源的Linux，Linux系统已经发展了十年了，代码库之大让林纳斯很难继续通过手工方式管理了，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！

集中式 vs 分布式

集中式：版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。

分布式：分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。

安装 Git

mac 自带 git，windows 需要安装 git。

安装完成后，还需要最后一步设置，在命令行输入：

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

创建版本库

理解成这个目录里面的所有文件都可以被Git管理起来。

创建一个文件夹，命令行进入到这个文件夹。

```
mkdir repo1 # 创建repo1目录
cd repo1    # 进入到repo1目录
pwd         # 显示当前目录的路径
ls          # 显示当前目录的内容
```

然后把这个目录变为 Git 可以管理的目录

```
git init    # git初始化
```

把文件添加到代码仓库

在 repo1 这个文件夹里，创建一个名字叫做 a.txt 的文件。里面写点内容，比如写 111。

```
touch a.txt # 在当前目录下创建 a.txt 文件，如果装了xcode，可以用 open a.txt -a xcode 来编辑文件
```

第一步，用命令git add告诉Git，把文件添加到仓库

```
git add a.txt
```

第二步，用命令git commit告诉Git，把文件提交到仓库：

```
git commit -m "日志1"
```

继续工作

把 a.txt 的内容改为222

```
git status  # 查看状态
```

> Changes not staged for commit 指有些文件做了改动，但还没有被提交。

git status 执行完后，提示信息中，告诉我们有些文件做了改动，但还没提交，然后建议我们使用 add 提交，或者使用 checkout 撤销修改。

如果我们想看到某个文件具体的改动，可以使用 diff 命令。

```
git diff a.txt # 查看哪些内容发生了变化
```

我们把刚才修改过的文件添加一下。

```
git add a.txt
```

然后再次看看它的状态

```
git status
```

这回提示修改的文件已经添加，但没提交，那我们提交一下

```
git commit -m "日志2"
```

然后再次看看它的状态

```
git status
```

这回提示的是没有改动的文件了，即仓库里和本地的文件都一致了。

> 每当对文件做了一些改动，就要重复的执行上面add和commit。就相当于玩游戏时的存盘，如果哪天想回退，就可以用reset命令了。

版本回退

log 命令显示从最近到最远的提交日志。

```
git log
```

如果嫌弃输出的内容太多，可以加参数，简化输出。

```
git log --pretty=oneline
```

你看到的一大串类似3628164...882e1e0的是commit id（版本号）

在Git中，用HEAD表示当前版本，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^
```

然后查看 a.txt 的内容，发现果然回退到上一版本。

然后继续使用log查看

```
git log
```

发现撤销前的那个版本竟然看不到了，即想回去回不去了，怎么办?

只能想办法找到版本号（如果你命令行窗口未关闭的话）

```
git reset --hard 63300dca594
```

> 版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

如果已经回退到了某个版本，如果想知道回退前的版本号，可以使用reflog找到版本号。

> Git提供了一个命令git reflog用来记录你的每一次命令

```
git reflog
```

工作区和暂存区

工作区（Working Directory）

就是你在电脑里能看到的目录，比如我的repo1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。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

撤销修改

状态1:

继续对 a.txt 的内容进行修改，然后保存文件。不要进行 add 和 commit。

在命令行中，使用 checkout 可以撤销修改。

```
git checkout -- a.txt    # 注意文件名和--之间有空格
```

然后打开文件，就发现内容确实撤销了。

状态2:

如果是对文件内容修改后，已经执行了add，那么撤销就应该用reset了。

```
# 先把暂存区的修改，回退到工作区
git reset HEAD a.txt
# 然后撤销工作区的修改
git checkout -- a.txt
```

状态3:

如果是对文件内容修改后，已经执行了add，并且已经执行了commit，那么撤销应该这么做：

```
git reset --hard HEAD^      # 上文讲过
```

删除文件

把刚才的 a.txt 删除掉，然后在命令行中输入 git status 能够看到已经删除文件的提示。

如果我们想把已经删除的文件还原回来，可以使用 checkout

```
git checkout -- a.txt
```

如果我们确定删除这个文件，那么就要保证工作区和版本库一致

```
git rm a.txt    #版本库中删除 a.txt
```

小结

命令	示例	作用
init	git init	初始化git
status	git status	查看版本库状态
add	git add a.txt	把a.txt文件由工作区添加到暂存区
commit	git commit -m "日期"	把暂存区文件，提交到代码仓库中
diff	git diff a.txt	查看a.txt发生了哪些变化
log	git log --pretty=oneline	从最近到最远的提交日志
reflog	git reflog	用来记录你的每一次命令
reset	git reset --hard HEAD^	回退到某一个版本（代码仓库->工作区）
reset	git reset HEAD a.txt	先把暂存区的修改，回退到工作区（暂存区->工作区）
checkout	git checkout -- a.txt	撤销a.txt文件的操作（工作区->工作区）
rm	git rm a.txt	版本库中删除 a.txt

讲到这里，本地的代码仓库全部搞定了。

远程代码仓库

GitHub

gitHub是一个面向开源及私有软件项目的托管平台，因为只支持git 作为唯一的版本

库格式进行托管，故名gitHub。

2018年6月4日，微软宣布，通过75亿美元的股票交易收购代码托管平台GitHub。

如果我们希望这个项目的版本仓库，本地有一套，网络上也有备份的话，我们需要怎么做?

第1步 生成本地密钥

创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有 id_rsa 和 id_rsa.pub 这两个文件，如果已经有了，可直接跳到下一步。

```
# mac 终端中输入命令查看用户主目录下是否有.ssh目录
# wangyangdeMacBook-Pro:~ wangyang$ ls -a

# window 下.ssh的位置
# C:\Users\wangyang\.ssh
```

如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

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

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以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。

第2步 GitHub授权

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，点“Add Key”，你就应该看到已经添加的Key。

添加远程代码仓库

首先，登录 GitHub，然后右上角找到 Create a new repo，创建一个新的代码仓库。

> 注意：创建代码仓库时，不要添加 readme 和 .gitignore 和 Licenses，如果添加了这些，首次就不能上传，需要先下载后上传。

1. readme.md 这个文件的内容是这个项目的介绍等等相关信息。
2. .gitignore 指忽略规则，比如 node_modules 这个文件夹我们根本就不需要 git 去管理。
3. licenses 版权许可证，比如说他人修改代码后，是否可以闭源等。

> 我的账户名是wangyang1025，邮箱是550759049@qq.com，密码保密。

创建成功之后，根据 GitHub 的提示，在本地的 repo1 下执行

```
# 把本地仓库与网络仓库相关联
git remote add origin git@github.com:wangyang1025/repo_test.git
```

注意要把 wangyang1025 换成你自己的 GitHub 账户名

下一步，就可以把本地库的所有内容推送到远程库上：

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

如果推送失败的化，应先检查是不是账户写错了，然后检查是不是远程库里面有文件，如果远程库里面有文件，我们需要先下载，然后上传。

```
git pull --rebase origin master
```

> 传输速度的快慢受限于网络和文件大小

推送完毕后，回到 GitHub 中，就能够看到相应的文件了。

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

只要本地作了提交，就可以通过命令：

```
git push origin master
```

把本地master分支的最新修改推送至GitHub

SSH警告

当你第一次使用Git的clone或者push命令连接GitHub时，会得到一个警告：

```
The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.
RSA key fingerprint is xx.xx.xx.xx.xx.
Are you sure you want to continue connecting (yes/no)?
```

这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

```
Warning: Permanently added 'github.com' (RSA) to the list of known hosts.
```

这个警告只会出现一次，后面的操作就不会有任何警告了。

把远程代码仓库克隆到本地

命令行进入到本地的 test 文件夹中。

```
git clone git@github.com:wangyang1025/repo_test.git
```

就会在 test 目录下，把 github 的 repo_test 拿下来了。

然后就可以直接使用 git 了。（ 无需 git init ）

分支

创建与合并分支

我们创建dev分支，然后切换到dev分支

```
git checkout -b 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  # 创建dev分支
git checkout dev # 切换到dev分支
```

用git branch命令查看当前分支

```
git branch
```

git branch命令会列出所有分支，当前分支前面会标一个*号。

然后，我们就可以在dev分支上正常提交。

对某文件进行更改后，提交。

```
git add 1.txt
git commit -m "分支测试"
```

dev分支的工作完成，我们就可以切换回master分支。

```
git checkout master
```

命令行切换到不同的分支时，我们的文件也会有对应的变化。

现在，我们把dev分支的工作成果合并到master分支上：

```
git merge dev
```

git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

合并完成后，就可以放心地删除dev分支了：

```
git branch -d dev
```

删除后，查看branch，就只剩下master分支了

```
git branch
```

小结

命令	示例	作用
branch	git branch	查看分支
branch	git branch dev	创建 dev 分支
checkout	git checkout dev	切换到 dev 分支
checkout	git checkout -b dev	创建 dev 分支，并切换到 dev 分支
merge	git merge dev	合并某分支到当前分支
branch	git branch -d dev	删除 dev 分支

解决冲突

创建 dev2 分支

```
git checkout -b dev2
```

修改 1.txt 的内容

在 dev2 分支中，提交 1.txt

```
git add 1.txt
```

```
git commit -m 'dev2'
```

切换到 master 分支

```
git checkout master
```

修改 1.txt 的内容

在 master 分支中，提交 1.txt

```
git add 1.txt
git commit -m 'master'
```

现在，master 分支和 dev2 分支，分别有新的提交。

这种情况下，Git无法执行快速合并。

```
git merge dev2 # 将 dev2 合并到 master 分支中
```

命令行提示冲突了，告诉我们需要手动解决冲突，然后提交。

也可以使用 status 查看哪些文件有冲突。

```
git status
```

也可以直接查看 1.txt 的内容

```
<<<<<<< HEAD
master
=======
dev2
>>>>>>> dev2
```

需要我们手动对 1.txt 文件进行更改后，重新提交

```
git add 1.txt
git commit -m '手动更新'
```

最后，删除 dev2 分支

```
git branch -d dev2
```

工作完成

git 当然还有其他一些功能，如果感兴趣可以去官网自行查阅。