Contents

[1. 基本概念与特性 1](#_Toc21670852)

[2. 安装、配置与初始化 1](#_Toc21670853)

[3. Git文件本地提交过程 2](#_Toc21670854)

[4. Git系统状态和日志查看 3](#_Toc21670855)

[5. Git的版本控制 3](#_Toc21670856)

[6. Git分支管理 4](#_Toc21670857)

[7. 远程库GitHub 5](#_Toc21670858)

[8. 标签管理 6](#_Toc21670859)

[9. 忽略特殊文件 7](#_Toc21670860)

[10. GitHub的配置 7](#_Toc21670861)

# 1. 基本概念与特性

Git只能跟踪文本文件的详细修改历史（不包括MS Word），其他的文件只能记录修改时间无法确定修改内容。

Git仓库在文件系统中有如下形式：包括用户选取的存储文件夹（即工作区working directory/working tree）和其中的.git文件夹（即版本库repository）。版本库中包括暂存区（stage/index）、分支（branch，一系列连续版本或连续提交commit组成的单一历史路径；其中在自动创建的第一个分支中，指向其最新版本或最新提交的指针叫master，也作为该分支的名称）和指向当前分支的指针HEAD。

Git指令的完整形式一般复杂多样，而且不仅仅具有单一功能，而是可以通过“附带属性”顺便完成多种操作。换句话说，某个指令的功能可以通过“附带属性”的形式与另一个指令合并为一个以简化操作。初学者无需深究，了解即可。

# 2. 安装、配置与初始化

配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中。别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。而当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中。

①配置用户

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

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

--global // 表示这台机器上所有的Git仓库都会使用这个配置

②初始化Git仓库

git init

初始化一个Git仓库，或将某文件夹选取作为git仓库。

③设置Git显示颜色化的信息文字

git config --global color.ui true

④配置别名

git config –global alias.<name> <keywords sequence>

配置后使用别名name即可替代一串关键字序列，用以简化后续重复的命令输入

git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

以上是一个特殊的日志显示方式模板设定的别名lg

# 3. Git文件本地提交过程

Git中对于工作过程的跟踪不是针对文件而是针对改动（实际上包括insertion和deletion），每一次提交commit都对应一组改动。这个概念在合并过程中起到决定性作用，如合并提交时只是同步了该提交所做的改动到当前区，而不代表两个分支中的文件完全相同。

· 添加文件到Git仓库，分两步：（<>中的内容是非关键字字符）

①使用命令git add <file> (<file>,…)，注意，可反复多次使用，添加多个文件；

该命令即把文件添加至暂存区。

git rm <file> 可以提交删除文件的更新申请update（特别是工作区中还留存该文件时），如果工作区中该文件已被删除，那么rm与add命令效果一样。

git add -f <file>

强制添加文件，用于该文件类型被忽略时

②使用命令git commit -m <message>，完成。

该命令将暂存区内容提交至分支。

· 其他文件操作指令

git mv <path> <path>

移动文件

# 4. Git系统状态和日志查看

①状态查看

git status [<options>…] [- -] [<pathspec>…]

要随时掌握工作区的状态，使用git status命令。

②查看修改与区别

git diff HEAD -- <file>

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。

完整形式：git diff [options] [<commit>] [- -] [<path>…]

git diff可以显示各个不同区或文件之间的差异

③查看日志

git log

版本穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。

--pretty=oneline // 用简单的单行显示代替默认的多行

--graph // 显示分支合并图表

--abbrev-commit // 缩写commit ID

-1 // 显示最后一次提交信息

④查看命令历史

git reflog

回退历史后，较新版本信息在分支中不再可见。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。

# 5. Git的版本控制

可以说，一个版本就是一次提交，也是一组改动的集合。

①版本回退/前进

git reset --hard commit\_id。

在Git中，HEAD指向的版本就是当前版本，因此，HEAD可以用来代替commit\_id。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。

--mixed // 重置提交和暂存区

--soft // 只重置提交

--hard // 重置提交、暂存和工作区

②撤销工作区中的修改

git checkout -- <file>

意思就是，把<file>文件在工作区的修改全部撤销（包括删除文件本身），这里有两种情况：

一种是<file>自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是<file>已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次git commit或git add时的状态。

③撤销暂存区中的修改

git reset HEAD <file>

可以把暂存区的修改申请撤销掉（unstage），重新放回工作区。注意撤销修改申请和删除文件申请完全不同。

④缓存工作和暂存区内容（以便中途切换分支和工作内容）

git stash

缓存当前工作和暂存区状态与内容

get stash list

查看缓存区的内容

git stash apply (stash@{num})

恢复缓存区中第num个缓存

git stash drop (stash@{num})

删除缓存区中第num个缓存

git stash pop (stash@{num})

弹出缓存区中第num个缓存（等于恢复+删除）

⑤合并提交

不是合并整个分支，而是合并某一次commit所做的改动

git cherry-pick <commit ID>

合并后，Git自动将当前区提交为一个新的commit

# 6. Git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；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；开发工作都在dev分支上进行，每个feature都应该有自己的分支，每个人都应有自己的分支，时不时地往dev分支上合并就可以了。可以说，但凡可以细化和分工的内容都应该分支处理。

①创建与切换分支

git checkout -b <branch> 创建并切换分支

或

git branch <branch> 创建分支

git checkout <branch> 切换分支

新版本Git命令：

git switch -c <branch> 创建并切换分支

git switch <branch> 切换分支

②查看分支情况

git branch

③合并分支

git merge (-m <message>) <branch>

合并目标分支至当前分支，若失败或出现冲突需要手动resolve

--no-ff // 禁用FastForward模式，被合并的分支会产生两个独立但相同的副本，而非一个，且也会留下历史记录；一般都会禁用FF。实际上，FF只是改变了分支的指针指向，因此不留痕迹。

-m <message> // 加入消息，尤其用于产生新commit的情况，如no-ff

④删除分支

git branch -d <branch>

一般的删除指令

-D // 强行删除，用于丢弃一个没有被合并过的分支，此时一般删除操作会报错

⑤重命名分支

git branch -m <branch>

⑥分支变基

git rebase

蕴含一系列分支操作（可能会合并失败），目的是把本地未push的分叉提交历史整理成直线，以达到清晰目的，否则会出现不必要的分叉回路。尤其是两方同时编辑同一条分支时，不希望出现分叉回路而是一条直线时。

# git7. 远程库GitHub

Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。配置远程库关联和分支关联有多种命令和方式，但一般来说，二者的概念是独立的。

①从本地关联远程库（远程库已建立后）

这种方法一般都需要继续进行分支关联来定义本地库和远程库的实际内容关系。

git remote add origin <remote-url>

origin是Git对远程库的默认指代，虽然可以更改但是惯例上不必要

<remote-url>是远程库的路径名，举例git@github.com:michaelliao/learngit.git

git remote set-url origin <remote-url>

用于更改或设置远程库路径

②向远程库推送（本地与远程关联后）

git push -u origin <rbranch>

将整个分支推送给远程库

-u参数用于远程库原本不存在该分支的对应分支的情况，亦即远程没有关联至本地分支的对应分支，初次使用后无需再加该参数

③从远程库克隆至本地库（另一种库关联）

git clone git@server-name:path/repo-name.git

使用这种库关联方式后，初始时一般本地只有master分支（若远程库有其他分支时，不会在初始时被关联至本地）

④本地分支关联远程库中的分支（分支关联）

git checkout -b <lbranch> origin/<rbranch>

实际上是在本地创建一个关联到远程库中某分支的分支

git branch --set-upstream-to=<upstream> [<lbranch>]或

git branch -u <upstream> [<lbranch>]

将一个分支（如远程分支rbranch）设置为lbranch（若省略则为当前分支cbranch）的上游（准确的说是当前提交commit的上游），如果不设置会显示有关tracking的错误，即为缺少分支关联。

<upstream>可以是origin/<rbranch>

⑤查看远程库的信息

git remote -v

显示远程库的详细信息

⑥从远程库取回对应分支

git pull [options] [<repository> [<refspec>…​]]

本指令也可以顺便关联本地与远程分支，从远程库取回对应分支并与本地合并

[options] --allow-unrelated-histories 本次允许合并不同祖先或源内容的分支，

<repository> 可以是远程库的名称

<refspec> 可以是远程库分支名称

git fetch [<options>] [<repository> [<refspec>…​]]

取回分支和标签，但是不与本地合并

# 8. 标签管理

①创建标签

git tag <tagname> (<commit ID>)

不带commit ID时默认对当前最新版本加标签

完整形式：git tag -a <tagname> -m <message> (<commit ID>)

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

②查看所有标签列表

git tag

③查看标签详细信息

git show <tagname>

④删除标签

git tag -d <tagname>

本地删除标签

git push origin :refs/tags/<tagname>

本地删除标签后，推送远程同步删除（若已经推送过）

⑤推送标签至远程

git push origin <tagname>

单独标签

git push origin --tags

全部标签

# 9. 忽略特殊文件

通过添加文件信息至.gitignore文件来在Git库中隐藏这些文件。之后commit该.gitignore文件即可。

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。

①查看忽略文件信息

git check-ignore -v <file>

查看.gitignore中关于file的具体设定

# 10. GitHub的配置

SSH秘钥

由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置。SSH是公开密钥的加密方式，需要先创建密钥对。为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。