Contents

[1. 基本概念与特性 1](#_Toc47375217)

[2. 安装、配置与初始化 2](#_Toc47375218)

[3. Git文件本地提交过程 3](#_Toc47375219)

[4. Git系统状态和日志查看 4](#_Toc47375220)

[5. Git的版本控制 5](#_Toc47375221)

[6. Git分支管理 6](#_Toc47375222)

[7. 远程库GitHub 8](#_Toc47375223)

[8. 标签管理 10](#_Toc47375224)

[9. 忽略特殊文件 11](#_Toc47375225)

[10. 冲突解决 13](#_Toc47375226)

[11. Git Submodules使用 13](#_Toc47375227)

[12. GitHub的配置 14](#_Toc47375228)

# 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。

· Git指令的“附带属性”具有统一性，如-v或--verbose，-r或--remote等在多个不同指令中均充当相似的配置选项，可以统一记忆。且单个字母的属性前加单横线，单词属性前加双横线。但是单字母缩写和单词形式不一定首字母相同，存在少数个别例外。

· Git指令的“附带属性”也具有结合性，如一个指令的多个附带属性-n和-d，可以简写为-nd，等效于-n -d。

· 不同于cmd等命令形式，Git指令的附带属性是区分大小写的，即-x和-X代表不一样的设置。

# 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。在git文档中，经常会看见xxx.yyy形式的量，这些参数都是git config的可配置属性。

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

⑤配置换行符号

在Unix系统中，行末标志end of line由line feed(LF)标识。在windows中，则用carriage return(CR)和LF标识，是为CRLF。因此在多人多系统合作项目中，文档的换行格式存在配置需求。

git config --global core.autocrlf true/input/false

true开启autocrlf功能，每次签出文件会自动将LF转换为CRLF，提交时则相反，适合windows系统工作区。

input签出时不转换为CRLF，但提交时转换每个CRLF至LF，从而保持repository中使用LF格式。适合Max或Linux等Unix系统工作区。

false关闭autocrlf，不做任何转换。

⑥配置编辑器

当需要用户输入和编辑文本时，Git Bash会启动文本编辑器editor，编辑器的种类繁多且用法各异。在windows系统平台上，最易用方便的就是设为notepad。

git config core.editor notepad

另外一提，Git是基于Unix系统命令格式而发明的，Git Bash一般会默认使用Unix Vi/Vim编辑器。键入i进入输入insert模式，esc退出输入模式，键入:x!或:x或:wq!或:wq再enter以保存并退出，键入:q!不保存放弃编辑。

⑦其他配置属性与选项

· merge.conflictstyle

diff3 //将原本的冲突标记（两方）改以三方冲突标记显示（增加了共同基版本）

merge //默认显式方式，即两方冲突标记形式

# 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 add .

可以添加所有当前目录下的合格文件。这种写法适合于所有类似情况。

git rm [--cached] <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。

--cached

只unstage并删除暂存区中的文件，工作区中的文件无论是否修改都不会受影响。

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。

③其他文件操作指令

git mv <path> <path> 移动文件

git clean [-n] [-d] [-f] [-X] 清除未跟踪文件

-n或--dry-run

不执行清除，仅显示会被清除的内容

-d

对未跟踪目录进行深入递归以删除内容，否则只会删除根目录下的未跟踪文件

-f或--force

强制执行，若clean.requireForce设置为true

-X

设置只删除被gitignore忽略的文件

# 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

[options]

-b或--ignore-space-change

--check //检查引入的修改是否还留存冲突标记（>>>>或<<<<等）和空格错误（core.whitespace）

--ours //从本地分支的角度查看合并结果的差异（合并属性）

--theirs //从外来分支的角度查看合并结果的差异（合并属性）

--base //从共同基的角度查看两方引入的差异（合并属性）

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。Git Bash中输出完日志内容后会出现“冒号：”而不再显示命令提示行，此时输入“q”可以退出日志显示即可。

--graph // 显示分支合并图表

--pretty=oneline // 用简单的单行显示代替默认的多行

--abbrev-commit // 缩写commit ID

--oneline // 等效于同时应用--pretty=oneline和--abbrev-commit

--merge // 只显示出现冲突的提交（合并时）

-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 [options] 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。

[options]

--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

git reset [options] origin/rbranch

另外还可以用branch名称代表其HEAD版本，此时执行的内容不再是回退，而是将目标分支的HEAD引入作为当前分支的最新提交，且不执行任何合并与修改检测。

**注意：**这个操作并非一般的常见操作，只用于解决一些特殊问题的情况，且执行前应做好相关的确认与缓存准备。

git revert [-m 1|2] <commit>

回滚提交，但是不重置refs或缩短分支，而是以一个新提交延续分支。在分支历史中，该合并仍然存在，只是实质没有合并，因此无法直接使用git merge继续合并。除非再次用git revert回滚新提交。

-m 1|2 // 若该提交是一个合并，此参数指定保留哪个分支内容，1为本分支，2为外来分支

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。

git restore -- <file>

这个指令也有类似的功能。

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。

◆Git常用的分支合并算法/策略有fastforward（将新版本和旧版本合并），three-way merge（将两个新版本合并）和recursive three-way merge（前者的扩展版本，但功能相同）。对于除Git外的其他版本控制系统，有各自不同的版本管理和合并算法。这些算法只是让计算机能够完成人类直觉上的合并任务，并非加入了特别或神奇的操作，面对冲突多数情况下还是只能依赖手动解决。

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

**注意：**git checkout切换分支后虽会签出并且切换为另一个分支的内容，但是不会移除当前分支中、切换分支前进行的修改（以便再次在签出分支中提交）。因此切换分支之后，最好使用git status检查当前分支的内容。git switch则会按照用户预期的完全切换分支内容，不会留下“残留”内容。

②查看分支情况

git branch

-r //查看远程库分支情况

③分支合并操作

git merge [options] [-m <message>] [<branch>]

合并目标分支（可以多于1个，是为octopus merge）至当前分支，若失败或出现冲突需要手动resolve

[options]

--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。

--autostash

--no-autostash // 在合并前自动创建一个临时stash（并不直接存入stash list），在合并后自动重新应用，适用于在编辑中的工作区上直接进行合并操作。这条指令也会产生MERGE\_AUTOSTASH项。

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

-s <strategy> // 指定合并策略或算法，默认是recursive

-X <strategy-option> // 指定某算法的某个配置选项  
-s recursive：  
-X ours //用于自动优选本地版本的内容解决冲突  
-X theirs //和ours相反

-X ignore-all-space //比较修改时忽略所有空格

-X ignore-space-change //比较修改时忽略1个以上空格数量上的变化

--continue // 手动解决完冲突后，使用该指令继续合并并自动commit。也可以直接使用git commit

--quit // 放弃当前的合并，但是不恢复修改的内容。

--abort // 如果在合并中途（正在解决冲突）想要取消并恢复被修改内容，使用该选项。git merge --abort和后面的git reset --merge在一般合并的情况下是等效的。除非设置了MERGE\_AUTOSTASH，则git merge --abort将在取消合并修改后再将stash内容应用，而git reset --merge会取消合并后将stash内容存于stash list，即不立即将其应用。

git reset --merge

用于合并过程中遇到冲突时重置合并过程，详见git merge。

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

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。

git rebase <branch>

首先，git 会把当前分支里面的每个 commit 取消掉；  
其次，把上述所有提交临时保存成 patch 文件，存在 .git/rebase 目录下；  
然后，把当前分支更新到最新的<branch>分支内容；  
最后，把上面保存的 patch 文件应用到当前分支上。

git rebase -i HEAD~n

重新整理最近n次的提交记录

-i或--interactive

会使Git会弹出编辑窗口，并提供对每个提交的单独设置选项。可以实现对多个提交的合并、删除、重排序等。

# 7. 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>

用于更改或设置关联的远程库路径

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，或--set-upstream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，即git push就自动将当前分支与远程对应分支合并。

git push origin <lbranch>:<rbranch>

将本地分支推送合并给远程的非对应分支。origin只是关联远程库的引用名，也可以是其他库名称。

git push origin --delete <rbranch>

删除远程库中分支

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）。若要关联其他分支，使用下一组的指令。

url 本命令后面的地址可以是远程库的http网址

④本地分支关联远程库中的分支（分支关联/跟踪）

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，-b用于指定lbranch的名称（可以不同于rbranch）。

git checkout --track origin/<rbranch>

类似上一指令，只是默认将本地分支取相同名称。

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）的上游分支（也就是其关联分支或正在跟踪的分支），如果不设置会显示有关tracking的错误，即为缺少分支关联。也可以注意到，本地分支和远程分支取名可以不一样，因而不会自动设置关联。

<upstream>可以是origin/<rbranch>

⑤查看和同步远程库的信息

git remote -v

显示远程库的详细信息，其实就只是fetch和push命令分别对应的远程库名和其地址。

git remote [-v|--verbose] update [-p|--prune] [(<group>|<remote>)...]

更新本地存储的远程库信息，这不是同步或合并内容。一般情况下，GitBash不会实时更新本地的远程库信息（如果修改来自于他人），如果想要看远程库是否有变化又不想使用fetch或pull等内容性操作，可以使用本指令。再使用git status即可看到最新状态。

git branch -r

显示远程库中的分支信息，-r可以放在很多其他命令中代表和远程库相关的含义。

git branch -a

显示本地和远程库中的所有分支信息。

git branch -vv

将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。-vv一般比-v多输出一些相关信息。

git ls-remote origin

显示远程库中的ref/heads信息，和其相关的commit ID。git branch会显示所有存在的分支历史，对于已经合并融入至其他分支的分支，仍然有branch记录，但是其ref/heads信息则不会存在。

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] 可以添加git merge指令的一些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，实际上git pull就是git fetch和git merge两个命令的结合（具体参数被自动确定）。

# 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配置语法：

除了注释，只是单纯的文件名匹配列表。

以井号”#”开头表示注释；

以斜杠“/”开头表示目录；

以星号“\*”通配多个字符，但不包括/，即可以表示文件夹或文件，但不是路径；

以问号“?”通配单个字符；

以方括号“[]”包含单个字符的匹配列表，可以用[0-9]等类似表示法表示范围

以叹号“!”表示不忽略(跟踪)匹配到的文件或目录；

此外，git 对于 .ignore 配置文件是按行从上到下进行规则匹配的，意味着如果前面的规则匹配的范围更大，则后面的规则将不会生效。

举例：

*# Python:*

\*.py[cod]

\*.so

\*.egg

\*.egg-info

dist

build

*#忽略fd1目录；不管是根目录的 /fd1/ ，还是某个子目录 /child/fd1/ ，都会被忽略*

*#注意不能后加\*写成fd1/\*，这无法被识别*

fd1/

*#忽略根目录下的 /fd1/ 目录*

/fd1/

*#忽略全部内容，但是不忽略 .gitignore 文件、根目录下的 /fw/bin/ 和 /fw/sf/ 目录*

/  
!.gitignore  
!/fw/bin/  
!/fw/sf/

*#忽略doc目录下的所有.txt文件，包括子目录下的，若不加\*\*表示仅忽略doc根目录下的*

doc/\*\*/\*.txt

②查看忽略文件信息

git check-ignore -v <file>

查看.gitignore中关于某文件的具体设定，file是一个可以随意查找的名称，如文件名的一部分或目录名的一部分。

③忽略已跟踪或上传文件的相关问题

· 已经上传或跟踪的文件无法被gitignore忽略，至少无法被单独修改gitignore的行为忽略。而单独修改该文件并无视已经跟踪的文件可能会造成以下问题：

a. 如果忽略的是一个文件类型，那么在添加gitignore条目之后创建的该类型文件均不会被跟踪。但如果另一个分支上由于应用gitignore时间节点的不同，跟踪了被原分支忽略的文件。那么原分支和新分支将无法合并，且会报错“untracked files will be overwritten”，但在原分支视角下，这些未跟踪文件由于被忽略而根本不会显示。

· 如何删除或同步忽略已跟踪或上传文件？

使用git rm --cached <file>命令删除该文件，再进行提交和推送，但本地文件不受影响或被删除。

# 10. 冲突解决

1）手动解决冲突

①从已标记的综合文件选择冲突解决结果

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容。在所有可文本化的文件中，如下：

<<<<<<< HEAD

本分支的副本

=======

外来分支的副本，是为MERGE\_HEAD

>>>>>>> xxxxxx

直接将上述内容改成想要留存的内容，再保存即可（不要留有<=>等符号）。

②检出各分支版本的冲突文件进行比较和冲突解决

· 有些时候，文件的分歧较大时，只查看综合标记文件可能不足以完成冲突解决。此时可以用最根本的方法——调出冲突文件各版本进行全篇审阅。

git show :n:filename > newFileName

将某个版本的filename文件复制到newFileName文件，以查看

n //1代表共同祖先版本，2代表本地分支版本，3代表外来分支版本

git merge-file -p ourName commonName theirName > fileName

单独针对一个文件的合并指令，前三个名称为三路合并的三个文件名，最后一个文件名是存储最终结果的文件名称。

2）自动冲突解决

①同时对全部文件生效（合并前设定）

· 见git merge部分指令和配置属性

②单独合并个别文件并自动解决冲突（合并时设定）

git checkout [--ours|--theirs] <filename/path>

针对有时需要针对个别文件进行自动冲突解决（即完全选取一方修改作为冲突解决）。

**注意**：在rebase情况下，己方和对方的概念可能和直观相反。

# 11. Git Submodules使用

submodules简单理解就是git项目中的子git项目。

# 12. 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。