Git V2.16.2

- **init**
  - `git init` — 主要用于客户端
    - 初始化一个版本库
  - `git init --bare` — 主要用于服务端
    - 创建一个裸的git仓库，主要用于服务端
- **config**
  - 初始化
    - `git config --global user.name "username"`
    - `git config --global user.email email@email.cc` — 初始化用户名称和邮箱
  - 设置别名
    - `git config --global alias.br branch` — 设定命令别名
  - 查看配置
    - `git config --list` — 检查当前配置
    - `git config <key>` — 检查 Git 的某一项配置
  - `git config --global core.editor vim` — 设置默认的文本编辑器
  - `git config credential.helper store` — 保存凭证，避免每次pull或者push的时候询问除密，在制作种子的时候十分有用
- **add**
  - 将文件添加到暂存区
    - `git add *.txt`
    - `git add -A` — 添加全部文件到暂存区
    - `git add -p` — 会轮流询问每一个文件是否需要添加
    - `git add -i` — 交互式暂存
- **commit**
  - `git commit` — 会调到默认的文本编辑器，填写更新信息
    - `git commit [(--message | -m ) "update" ]` — 提交更新到版本库
  - `git commit -a -m 'update'` — 跳过使用暂存区，将所有已经版本跟踪目修改从工作区提交到版本库
  - `git commit --amend` — 将暂存区的文件重新提交，将目前的修改追加到上一次提交中
    - 例如，你提交后发现忘记了暂存某些需要的修改，可以add到暂存区，然后git commit --amend
    - 这是一个变基操作，请谨慎！
- **push**
  - `git push origin master` — 推送到远程仓库
  - 推送标签到远程仓库
    - `git push origin v1.4` — 推送标签
    - `git push origin --tags` — 推送多个标签到远程仓库
  - 移除远程分支
    - `git push origin :dev`
    - `git push origin --delete dev` — 移除远程分支（只是移除指针，可恢复）
- **status**
  - 查看工作目录状态
    - `git status` — 检查当前文件状态
    - `git status [ -s | --short ]` — 获得简短的状态输出
- **reset**
  - `git reset HEAD <file>` — 将指定文件还原到HEAD的状态
    - 在没有带参数执行reset的情况下，默认执行 git reset --mixed 也就是索引中已经保存的是之前版本的文件
  - 取消本地工作区的合并
    - `git reset --merge` — 取消工作区的合并！
  - 这三个参数分别对应reset中的三棵树
    - `--soft` — 对应的是仅仅移动HEAD的指针，并未对索引和工作目录做任何操作，它实际上只是撤销了最近一次的git commit可以使用git commit --amend来继续提交
    - `--mixed` — mixed会改变索引的操作，这个时候你只需要 git add 和 git commit 就可以回到之前的状态
    - `--hard` — hard 参数会修改HEAD，索引，工作目录，也就是reset重写三棵树的最后一步
    - `git reset --hard 39ea21a` — 重置分支指针
  - 你可以使用reset来进行压缩提交，比如对现在这个二进制的git.xmind文件
    - `git reset --soft b123456 git.xmind` — 先 将文件在HEAD的指向上退回一个较旧的状态，这个时候git status或者 git diff可以看到效果
    - 然后git commit 即可 — 将分支的子分支重新指向最新的文件，即可压缩提交
    - 这样就不需要进行交基操作
- **stash**
  - 查看储存栈
    - `git stash list` — 查看储存栈中的储存
  - 保存到储存栈
    - `git stash [ save "message" ]` — 将现在工作区和暂存区的更改保存到储存栈
    - `git stash -u` — 储藏未跟踪的文件
    - `git stash --keep-index` — 储存工作区的更改而忽略暂存区的修改
  - 从储存栈恢复
    - `git stash pop` — 从栈顶恢复，并删除（出栈）
    - `git stash pop stash@{2}` — 从栈的任意位置恢复内容，并删除（出栈）
    - `git stash apply` — 从栈顶恢复储存，而不删除
    - `git stash apply stash@{2}` — 从栈的任意位置恢复储存，而不删除
  - 从储存栈删除
    - `git stash drop` — 从栈顶移除记录
    - `git stash drop stash@{2}` — 从任意位置移除记录
    - `git stash clear` — 清空储存栈
  - 储存未跟踪的文件
    - `git stash branch testChange` — 基于暂存创建一个分支
    - `git stash [-u|--include-untracked]`
    - `git stash --patch` — 交互式的询问每一个文件是否需要储存
- **clean**
  - `git clean -d -n` — 会通知清理工作目录会清理那些文件（~多数的作用），但不会执行删除
  - `git clean -d -f` — -f 参数有 “强制执行” 和 “确认执行” 的意思
  - `git clean -i -d -n -f` — -i 交互式的询问，每一个删除的文件
  - `git clean -x -d` — 删除构建时的.o 文件
  - 如果需要清理全部的未跟踪的文件，强烈推荐使用该指令
- **show**
  - `git show v1.4` — 可以看到标签信息与对应的提交信息
  - `git show 1c002d` — 显示在这次提交内的更改
  - `git show 1c002d --name-only` — 显示这次提交修改了什么文件
- **grep**
  - `git grep -n TODO` — 从提交历史中寻找某段代码，-n 参数输出匹配位置的行号
  - `git grep --count TODO` — 输出总结信息
  - `git grep --break --heading -n -e '#define' --and \(-w LINE -e BUF_MAX \) v1.8.0`
- **rm**
  - `git rm *.txt`
    - `git rm *.txt -f （强制）` — 将文件从暂存区移除，并不继续跟踪（会在文件系统删除文件）
  - `git rm --cached README` — 把文件从版本库中和暂存区移除！
- **mv**
  - `git mv a.txt b.txt` — 将文件重命名/移动
- **gc**
  - `git gc` — 清除不必要的文件并优化本地储存库
- **submodule(undone)**
  - 初始化
    - `git submodule init`
    - `git submodule update`
  - 循环操作
    - `git submodule foreach git pull origin master`
- **rerere(undone)**
- **blame(undone)**
- **bundle(undone)**
- **fsck (undone)**
- **reflog**
  - git在你工作的同时，会在后台保存一份reflog，这是一份最近几个月你的HEAD和分支引用的日志
- **filter-branch**
  - 以校本化的方式重写大量提交
    - 强烈推荐
  - 删除历史中的全部.env 和passwords.txt 文件
    - `git filter-branch --tree-filter 'rm -f .env passwords.txt' HEAD`
  - 修改 trunk 为根文件夹
    - `git filter-branch --subdirectory-filter trunk HEAD`
  - 全面修改电子邮件地址
    - `git filter-branch --commit-filter ' if ["$GIT_AUTHOR_EMAIL"="schacon@localhost"]; then GIT_AUTHOR_NAME="Scott Chacon"; GIT_AUTHOR_EMAIL="schacon@example.com"; git commit-tree "$@"; else git commit-tree "$@"; fi HEAD'`
- **cherry-pick (undone)**
- **gitk**
  - 打开默认的图形化工具
- **remote**
  - 列出远程仓库
    - `git remote` — 当前配置的每一个远程仓库服务器的简写
    - `git remote -v` — 显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL
    - `git remote show origin` — 查看远程仓库的更多信息
  - `git remote add myClone https://github.com/Kuri-su/Portflow-Monitor.git` — 添加远程仓库，并给远程库起名
  - `git remote rename origin ori` — 重命名远程仓库
  - `git remote rm origin` — 移除远程仓库
- **clone**
  - 可以使用各种专用协议 [ssh:// | git://| file:// | http://]
  - 克隆仓库
    - `git clone https://url.com/Kuri-su.git` — 克隆现有仓库，在当前目录新建的 Portflow-Monitor 文件夹下
    - `git clone https://url.com/Kuri-su.git test` — 克隆现有仓库，在当前目录新建的 test 目录下
- **pull**
  - `git pull origin master` — 拉取分支最新内容 pull = fetch + merge
  - `git pull --rebase` — 会以线性进行，用于变基场景
- **fetch**
  - `git fetch origin master` — 从远程仓库中拉取(不会自动合并)
- **checkout**
  - `git checkout master` — 切换分支
  - `git checkout -b test2` — 创建并切换到分支
  - `git checkout -- readme.md` — 撤销对文件的修改
  - `git checkout --track origin/dev` — 跟踪远程分支，并在本地创建同名分支（git checkout -b serverfix origin/serverfix）的简写 不过git checkout -b sf origin/serverfix 可以使本地分支和远程分支不同名
- **branch**
  - 创建分支
    - `git branch test` — 创建分支
    - `git branch still-a-branch 38bb7da5e` — 从任意提交创建分支
    - `git branch still-a-branch older-branch` — 从现有分支创建分支
  - 查看分支
    - `git branch` — 列出全部分支
    - `git branch -v` — 列出全部分支，并查看每个分支的最后一次提交
    - `git branch -vv` — 将所有的本地分支列出来并且包含更多的信息
    - `git branch --merge` — 查看哪些分支已经合并到当前分支
    - `git branch --no-merged` — 查看所有包含未合并工作的分支
    - `git branch -a` — 查看全部的本地分支和远程分支
    - `git branch -r` — 查看全部的远程分支
  - 切换跟踪的远程分支 — `git branch -u origin/serverfix`
  - 删除分支
    - `git branch -d test` — 删除分支
    - `git branch -D test` — 强制删除分支
- **merge**
  - `git merge dev` — 合并分支
- **tag**
  - 列出标签
    - `git tag` — 列出现有标签
    - `git tag -l 'v1.8.5*'` — 查找标签
  - 附上标签
    - `git tag -a v1.4` — 创建轻量标签
      - 一个轻量标签很像一个不会改变的分支 - 它只是一个特定提交的引用。
    - `git tag 1.4.7 a7b8b4 -m "alpha beta"` — 给 a7b8b4 的提交创建一个名为1.4.7标签，注释为“alpha beta”
    - `git tag -a v1.4 -m 'my version 1.4'` — 创建附注标签
      - 附注标签是存储在 Git 数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间，还有一个标签信息；并且可以使用 GNU Privacy Guard （GPG）签名与验证
  - 删除标签 — `git tag -d v1.4`
- **diff**
  - `git diff foo.txt` — 工作目录中当前文件和暂存区域快照之间的差异
  - 查看暂存区文件和HEAD的区别 — `git diff --staged`
  - 指定范围内比较
    - `git diff 77d233f HEAD` — 对比两次提交
    - `git diff 77d233f^` — 和上一次提交进行比较
    - `git diff 77d233f 05bcfd1 - book/bisection/` — 限制比较的范围
- **log**
  - 显示提交差异和统计信息
    - `git log -2` — 显示最近的两条提交记录
    - `git log -p -2` — -p用来显示每次提交的内容差异
    - `git log --stat` — 每次提交的简略的统计信息
  - 简短 | 完整 的显示提交
    - `git log --pretty=short` — 简短显示log
    - `git log` — 查看提交历史
    - `git log --pretty=full` — 完整显示log
    - `git log --pretty=fuller` — 更加完整的显示log
    - `git log --pretty=format:"%h - %an, %ar : %s"` — 更改显示的格式
  - 图形化显示 合并 | 提交 | 分支 历史
    - `git log --stat` — 图形化显示合并和分支提交历史
    - `git log --graph` — 图形化展示提交历史
  - 根据日期筛选提交
    - `git log --after=2.weeks` — 此日期之后的提交
    - `git log --before="2017-10-15"` — 此日期之前的提交
  - 根据 作者 | 提交者 | 文件 筛选提交
    - `git log --author kurisu` — 仅显示包含作者的提交
    - `git log --committer kurisu` — 仅显示指定提交者相关的提交
    - `git log git.png` — 仅仅显示文件相关的提交
  - 根据 关键字 筛选提交
    - `git log --grep kurisu` — 仅显示包含关键字的提交
    - `git log -S function_name` — 列出那些添加或移除了某些字符串的提交
  - 查看从dev分支上还没有合并到master的那些提交(双点号)
    - `git log master..dev` — dev分支还没有被合并到master分支上的提交
    - `git log "branchA branchB"`
    - `git log branchA..branchB`
    - `git log branchA ^not branchB` — branchA还没有被合并到branchA的提交
    - 这三种操作等价
    - `git log branchA branchB "branchC`
    - `git log branchA branchB ^not branchC` — 所有包含在branchA和branchB中而不包含在branchC中的提交
  - 查看两个提交的非共有提交(三点号) — `git log master...experiment`
  - 日志搜索
    - `git log -S ZLIB_BUF_MAX o-oneline` — 使用-S选项让Git只显示出添加过或删除过该字符串的那些提交（只会显示提交）
    - `git log -G Gitflow --stat --oneline` — 使用正则表达式进行搜索（和度只到提交）
    - `git log -L :git_deflate_bound:zlib.c` — 将zlib.c文件中git_deflate_bound 函数的改动全部列举出来
  - 显示提交修改了什么文件 — `git log --name-only`
  - note
    - `git log --abbrev-commit --oneline`
    - `git log --graph --oneline`
- **rebase**
  - 推送前使用变基操作！
    - 不要将任何已经推送到中央服务器中的提交包含在内，这样做会出现相同变更的不同版本，造成混乱
    - “如果你和别人过不去，那就万事大吉。否则，同事会埋怨你，朋友和家人也都恨你。”
  - `git rebase master` — 对分支变基，消除钻石线
    - 变基实际上是将特别分支的提交在当前分支上重放
  - 改变历史 — `git rebase -i HEAD~3` — 变基，在 HEAD~3 ..HEAD 范围内，的每一次提交都会被重写，无论是否修改信息
    - pick
    - reword
    - edit
    - squash
    - fixup
    - exec
- **底层指令 (Plumbing) (undone)**
  - cat-file
    - 查看资源库对象的内容或类型和大小信息
    - `git cat-file -p HEAD` — 对象的基本内容，包括SHA-1，父提交的SHA-1，提交者的信息和提交时间，提交备注信息
    - `git cat-file -t HEAD` — 对象的类型
    - `git cat-file -s HEAD` — 对象的大小
  - ls-tree
    - 列出树对象的内容
    - `git ls-tree -r HEAD` — 递归列出树对象的内容
  - ls-files
    - 查看索引（也称为暂存区）的文件列表
    - `git ls-files -c` — cached
    - `git ls-files -s` — 输出更详细的信息，包括SHA-1等
    - `git ls-files -d` — delete files
    - `git ls-files -m` — modified files
  - hash-object (undone)
  - write-tree (undone)
  - update-index
    - `git update-index --assume-unchanged foo.txt` — 忽略临时性的本地修改
    - `git update-index --no-assume-unchanged foo.txt` — 停止忽略 单个文件
    - `git update-index --really-refresh` — 停止全部忽略
  - read-tree (undone)
  - update-ref (undone)
  - symbolic-ref (undone)
  - verify-pack (undone)
  - rev-parse
    - `git rev-parse dev` — 查看分支指向的某个特定的SHA-1值
- **Other**
  - 祖先引用
    - `^` — 在SHA-1值或者HEAD后面加上 "^" ，git会将其解释为此次提交的父提交
      - `git show HEAD^` — 查看HEAD的父提交，也就是倒数第二次提交
      - `git show d12345^` — 查看d12345的上一次提交
      - `git show HEAD^2` — 查看HEAD的第二个父提交，也就是倒数第三个提交（^2和^^等价）
      - `git show HEAD^^`
    - `~N` — `git show HEAD~3` — HEAD的父提交的父提交，等价于git show HEAD^^^

XMind:ZEN