Git

- 工作流
  - 集中式工作流
    - 2人以内的项目，1人单独维护的一个项目 — 场景
    - 基于一个master分支开发 — 使用
  - 功能分支工作流
    - 场景
      - 功能分支工作流适用于中小型团队的项目，平台或者工具，5人以内小团队，5人左右，去开发一个比较以功能为单位的工具，服务平台
    - 使用
      - master分支
        - 一般来说，那会是整一个master分支，作为你的稳定分支，这里的代码是随时可以上线的
      - feature分支
        - 有多个feature分支，每个feature分支用来开发一个功能
      - bugfix分支
        - 一般线上发现了bug，就了一个hotfix或者bugfix分支，然后在分支上复现bug
        - 修复bug，然后在测试环境进行验证以及回归测试，最后将分支合并到master上线，修复线上问题
  - 工作流程
    - bugfix
      - 修复bug版本，用产线上bug修复
      - 修复后再合并到master及develop版本
    - master
      - 主要稳定版本
      - 用于预发布和上线
      - 每次发布都要打tag标签，便于代码回溯
    - release
      - 发行版本，用于发布进行QA测试
    - develop
      - 开发版本，用于合并feature
    - feature
      - 开发完成再去合并到develop — 对每个开发人员负责的开发版本
  - 注意点
    - feature分支用完了之后，是要删掉的；release分支测试完了之后，也是要删掉的；bugfix分支用完了之后，也是要删掉的；
    - 唯一长期保存的只有master和develop两个分支，一个用于稳定线上，一个用于稳定上线
  - 稳定迭代
    - 适合的就是版本稳定迭代的项目，一个版本，接一个版本，接一个版本，面向企业内部的系统或者项目
    - 适用场景
      - OA系统、CRM系统、ERP系统，或者面向外部的系统，比如说电商系统，但是系统已经成熟稳定了，版本稳定迭代
      - 互联网公司里面，面向外部的系统，快速迭代，同时5个版本，8个版本，10个版本，多达几十等人频繁的迭出多个版本并行开发
    - 不足之处
      - develop分支会处于一个比较不稳定的状态
  - GitFlow工作流
    - 并行场景
      - 基准代码只有master
      - feature和develop都从master拉取，以master为准
    - 与稳定迭代区别
      - 用于合并release和master代码，然后再讲staging线并master，相当于一个中间缓冲分支
      - 新增staging
- GitLab
  - 使用
    - 创建group组
    - 创建user用户并赋予权限对应组
    - 配置公钥在GitLab中即可 — 使用开发者账户关联本地git
  - 权限控制
    - 将develop分支保护起来之后，就是说，每个研发人员要一次写好feature分支的代码
    - 有对应权限的才能合并develop分支，需要先对merge request提到develop分支进行code review
    - review通过之后，可以合并到develop分支，保持统集成
    - 将develop分支设置为protected
  - 比较分支差异
    - 例
      - git log --abbrev-commit 分支1..分支2
      - git log --abbrev-commit feature/001..master
      - git log --abbrev-commit HEAD..master
      - git log --abbrev-commit HEAD..origin/master
  - 比较代码差异
    - git diff
      - 工作区和暂存区的差异 — 不加
      - 暂存区和仓库的差异 — 加 --cached
      - 工作区和仓库的差异 — 加 HEAD
      - 两个分支之间的差异 — 加 分支1 分支2
      - 最近两次提交之间的差异 — 加 HEAD HEAD^
  - 操作暂存区
    - 场景
      - git add -i
        - 进入交互模式
        - 2 update是放入暂存区，3：revert是从暂存区里撤出来
        - 输入对应文件的数字，回车，对应文件就会变成*开头，表示暂存过的操作成功。再回车就表示当前操作，进入交互模式主界面
        - 7 quit退出交互模式
      - 可以通过git add -i交互模式中的revert操作将暂存区的代码撤出来
      - 然后在分支dev用commit就可以实现将暂存区中的多个功能代码分成多次提交
    - 暂存未提交代码
      - 当前分支存在没有commit的代码时，直接切换分支会报告异常
      - 提示未提交代码会被覆盖，无法切换分支
      - 不加
        - 将代码暂存起来
        - 暂存起来且就可以切换分支了
      - 加 list
        - 查看暂存列表
      - 加 apply stash@0
        - 恢复指定的一次stash
      - 加 apply
        - 将暂存的代码恢复
      - 加 drop stash名称
        - 手动删除掉某个stash
  - 命令 — 高阶实战技巧
    - 修改commit提交历史
      - git commit --amend
        - 说白了就是将上一个commit删掉，重新提交，重新提交代码自然会将暂存区的代码提交
        - git commit --amend是会修改commit的SHA-1 hash值的，所以如果一个commit已经push到了远程仓库，就不要再去修改了，否则的话会造成提交历史的一个混乱
      - git rebase -i
        - 加 HEAD~3
          - 进入编辑模式
          - 编辑前3个版本，将对应版本的pick改成edit
          - 然后就回到了最早的edit版本，使用git commit --amend修改
          - 修改完成后，在使用git rebase --continue继续做下一个版本
          - 最后一次git rebase --continue就说明完成了所有修改
        - 编辑模式也能 删除、合并、切分 commit
    - 撤回
      - 两个分支merge出现了冲突，还原到未合并的状态 — 加 merge --abort
      - 远程仓库撤回commit
        - 1
          - 回到了上一个commit的状态 — 加 git revert HEAD
        - 2
          - 同步撤回远程仓库的commit操作 — 加 git push origin master
      - 撤回远程分支merge
        - 1
          - 加 git revert -m 1 HEAD
        - 2
          - 加 git push origin master
        - 注意点
          - 撤回merge会导致下次合并无法合并merge内容
          - 需进行特殊处理，再撤回到merge的版本，才能找回merge内容
          - 不建议使用
  - 二分查找解决提交引入的bug
    - 流程概述
      - 开始二分查找 — 加 start
      - 会继续往前面二分查找 — 加 bad
      - 会继续往后面二分查找 — 加 good
      - 从二分查找状态中恢复到之前的状态 — 加 reset
    - 当前分支就会合并其他分支的某个commit的hash — 合并其他分支的部分commit
- 命令
  - 内幕原理/底层命令
    - 存储命令
      - hooks
      - info
      - description
      - config
      - objects
      - refs
      - HEAD
      - index
      - .git目录
        - 目录中包含了我们的client-side或者是server-side的钩子脚本
        - 目录中包含了我们在.gitignore文件中定义的不需要commit的那些文件模式匹配规则
        - 文件仅仅是由git lsess模块生成的，我们一般不用去管这个文件
        - 文件包含了这个git项目的所有配置项
        - 目录存储了git中所有的数据
        - 目录中存储了所有的指针
        - 包括了branch、tag、remote、等等，指向了objects目录中的数据
        - 指向了当前我们所处的分支指针
        - 文件存储了暂存区中的内容
      - 底层命令
        - 加 -w
          - -w之后，git就会将value写入数据库之中
          - 同时用hash值作为key来引用那个value对应的文件
        - 加 --stdin
          - 指的是从命令行接收value
        - 组合 -w --stdin
          - echo 'hello world' | git hash-object -w --stdin
        - 组合 -w 文件名
          - 第不使用--stdin，那么要紧跟在hash-object之后跟一个文件名
          - 它就会将一个文件作为value存储到数据库中去，然后返回一个hash值
          - git hash-object -w l23.txt
        - blob object
          - 存储的文件为blob类型
          - git hash-object
      - 底层命令
        - 加 -p hash值
          - 查看对应文件内容
        - 加 -t hash值
          - 查看git中存储的文件的类型
          - 一般和-blob、tree、commit等
        - 其他
          - 将输出的文件内容保存到指定文件
          - git cat-file -p 830aae4804e65cc73a720da7252750c7606e a30 > test.txt
          - 检查于通过git恢复被删除的文件
          - 查看master分支指针指向的那个tree object — git cat-file -p master^{tree}
      - cat-file
        - 查看存储在git中的文件内容
    - 提交代码的内幕原理
      - 概述
        - tree object
          - tree object通常会包含指向其他多个blob的指针，或者是其他tree object的指针
      - 底层命令
        - tree object
          - update-index
            - 加 --add
              - 因为暂存区中还没有这个文件，你可以认为这个文件是第一次进入暂存区，所以需要这个选项
            - 加 --cacheinfo
              - 不是从工作区中的文件，而是从暂存区（或者是Git仓库（git数据库））中加入文件
            - 加 100644
              - 100644 mode：这种模式的意思是，这个是一个普通的文件
            - 组合
              - 将数据库中存储的某个blob数据的引用放到暂存区中
              - git update-index --add --cacheinfo 100644 830aae4804e65cc73a720da7252750c7606e a30 test.txt
            - 将暂存区中的内容创建为一个tree object
            - 主要是包含了好一个或多个blob object的引用
          - git write-tree
        - commit object
          - 底层命令
            - echo 'first commit' | git commit-tree d8329f
          - 概述
            - commit object的含义很简单，就是指向了一个tree object
            - 而那个tree object指向了多个blob，其实这个tree object就代表了某一时刻项目中所有代码文件的快照版本
            - 同时那个commit object包含了作者、提交时间、提交备注，此外还包含了指向上一次commit object的指针
            - 高级命令对应底层命令
              - git hash-object -w
              - git update-index --add --cacheinfo 100644
              - git write-tree
              - git commit-tree d8329f
              - 加 add
              - 加 commit
      - 扩展
        - blob、tree、commit，三种object都是存为一个独立的文件在objects目录中的blob，并根据SHA-1目录命名和文件名，都是基于hash值来的
        - SHA-1 hash值如何计算 查询git blob的内容分为两块
          - 一个是header，一个是content
        - SHA-1
          - header就是类型+空格+长度+\u0000，然后合成header+content来计算出一个SHA-1校验和，作为hash值
    - 引用指针的幕原理
      - HEAD文件
        - 保存了对某个分支指针的引用，其实就是某个分支对应的文件
      - FETCH_HEAD文件
        - 就是最近一次fetch下来的每个分支对应的commit hash值
      - 扩展
        - 分支就是一个文件，保存了指向的commit的40位hash值
        - git/ref
          - 分支对应目录文件
            - heads
          - 标签对应文件
            - tags
              - heads、tags、remotes等子目录
          - 远程分支指针文件
            - remotes
          - 标签
            - tag object
              - tag object生成指向创建时指定的那个commit object，不能修改
            - remote
              - 如果我们添加了一个remote，同时push了一些东西到那个remote
              - git会在refs/remotes中保留对每个分支，最近一次push到远程服务器对应的commit
    - git文件合并内幕原理
      - 底层命令
        - packfile打包
          - 手动packfile打包命令 — 加 gc
          - 合并的时候，采取的是 rebase feature/001来合并
          - 保持项目的提交历史就是一条直线
          - git rebase --onto a22e8f8 9c4bf8c — 基于对应commit重新构建提交历史
          - git commit-tree 9c4bf8c^{tree} — 将原来的一个仓库拆分成两个仓库
          - 让多个项目共享一个子项目的代码修改权利
            - submodule等组合命令
            - Subtree等组合命令
              - git以前是推荐使用submodule来管理子项目，但是从1.5.2开始就推荐使用subtree了
              - subtree的优点，就是子项目的维护权限只来说几乎就是透明的，不需要引入太多的维护成本
      - 扩展
        - 手动 git gc，会执行packfile的压缩优化
        - 将 .git/objects下的各种object对应的文件，压缩成一个packfile
        - 每个packfile都对应一个pack文件和一个.idx文件，pack文件包含了多个object的内容，.idx文件包含了每个object在那个pack里面的offset偏移量
        - 在打packfile的意义：多个文件合并一个，节省空间；对大文件进行增量处理，仅仅增量新的部分，不用每次重传的，之前的版本保留增量内容
        - 用git verify-pack -v命令，可以查看 pack文件里包含的具体内容
        - git gc：garbage collector，jvm gc差不一样的，git将未使用过的文件的
    - git远程分支内幕原理
      - 分支对应配置文件概述
        - 每次执行git remote add命令之后，实际上就会在 .git/config文件中加入一段配置
          - [remote "origin"]
            - url = git@172.168.31.101:YoungStar/123.git
            - fetch = +refs/heads/*:refs/remotes/origin/*
          - refspec
            - 就是fetch后面的那串东西，+refs/heads/*:refs/remotes/origin/*
            - 代表的是远程仓库被追踪的分支，远程仓库的master — refs/heads/*
            - 代表的是本地仓库用来追踪远程仓库的分支，本地仓库的origin/master分支 — refs/remotes/origin/*
            - 代表的是在执行git pull的时候，将远程分支和本地分支进行merge — +
            - 不常用 — 可以多动指定多个refspec**
          - 别名
            - git checkout -b feature/001 origin/feature/001。
            - 这段命令执行之后，就会在 .git/config中加入branch配置，标注了本地分支和origin/*远程分支之间的关联关系
            - [branch "master"]
              - remote = origin
              - merge = refs/heads/master
            - [branch "test"]
              - remote = origin
              - merge = refs/heads/test
      - 远程merge相关
        - fast forward — 这
        - Three-way merge(3-way merge) — 这
- 版本控制原理解析
  - 概述
    - 分布式版本控制系统
      - 服务器所有代码在本地都有副本拷贝
      - 服务器挂掉如果，本地也可以使用，不需要联网即进行版本控制
      - 服务器磁盘损坏，代码全都丢失，也可以基于本地代码进行恢复
  - 配置
    - 远程仓库配置
      - 本地生成公钥 — ssh-keygen -t rsa -C "your_email@youremail.com"
        - id_rsa
        - id_rsa.pub
      - 远程仓库gitlab/github中 — id_rsa.pub公钥关联
    - 初始化 — git init
    - 提交文件至本地仓库
      - git add
      - git commit -m "first commit"
    - 关联远程仓库 — git remote add origin git@github.com:wanglaxi9/practice.git
    - 本地仓库推送到远程仓库并绑定 — git push -u origin master
  - 本地仓库结构与文件状态
    - 仓库结构
      - git项目有3个主要的部分
        - working directory / working tree：工作区 — 保存的是一个项目当前的一个版本对应的所有文件
        - staging area：暂存区
          - 就是一个文件，包含在git版本库中，主要是保存了一次要提交到那部分文件信息
          - 对暂存区有另外一个名称，叫做index，也就是索引
        - git directory / repository：git版本库
          - 其实就是git用于存储自己的元数据，以及文件数据库的地方
          - 默认就是在项目的.git隐藏目录中
    - 相关机制
      - 快照机制
        - 每次提交文件，都是保存一份这个文件当前这个状态的一个完整快照
        - 同时对这次提交维护一个指针，指向这个文件快照
      - 本地化操作
        - 大多数的git版本控制操作，只要在本地执行即可，所有的版本文件都在本地，因此操作是非常快速的
        - 比如说通过git查看提交历史，比如历史文件的差异，都可以在本地完成，不需要通过服务器做任何事情
      - 完整性保证
        - git在存储任何文件之前，都会对其执行一个校验和，然后用校验和指向那个文件
        - git用的是SHA-1 hash算法来计算校验和，这是一个40位的字符串
        - 例：24b9da655225987aa493b52f86f6cd6d3b00373
    - 文件状态
      - 代码文件分成两种
        - tracked — tracked文件就是已经提交到git版本库中的文件，后面可以进行modified或者staged状态
        - untracked — untracked文件，就是从来没有提交到git版本库的代码文件（也从来没有放入暂存区）
      - git管理的文件有三种状态
        - 提交状态（committed） — 我们的文件已经安全的保存在git的本地数据库中了
        - 修改状态（modified） — 我们修改了文件，但是还没有提交到git的数据库中去
        - 暂存状态（staged） — 将修改后的文件标记为即将通过下一次提交，保存到git数据库中去
      - 注意点
        - 在对文件修改没有add的情况对文件来说就是modified状态容回到工作区，add后才是modified状态暂存区
        - 文件的修改必须git add到暂存区里，才会被git commit时提交到仓库里，否则就是停留在工作区中
  - branch/分支
    - commit object/提交版本
      - 概述
        - 每个commit object就代表了这个项目的所有代码在那次提交的时候一个完整的快照版本
      - 工作原理
        - tree/blob
          - 将一个文件版本加入暂存区的时候，就会计算一个hash校验和加入版本库，并在对应版本库中blob文件类型中保存(使用)
          - 2 执行提交的操命形成一个tree放入版本库，这个tree会包含对本次提交的所有文件的blob的指针
        - commit object
          - 1 每执行一次commit，git都会存储一个commit object
          - 2 这个commit object中会包含一个指针，指向这次提交文件的快照
          - 3 这个commit object同时也包含作者的姓名和邮箱、提交说明，以及对上一次commit object的指针
    - HEAD/当前指针
      - 概述
        - HEAD就是指向一个对应的commit指针
        - git维护了一个特殊的指针HEAD，默认指向master分支指针
        - 当本地切换到其他分支时，HEAD指针就会指向那个分支
        - HEAD指针就是当前工作区域的指针
        - 当前指针，并非最新指针，当前指针HEAD可以指向之前的commit object
    - branch/分支
      - 概述
        - 每个分支，其实就是一个指针而已，分支就指向了那次的提交commit object
        - 切换分支，就是切换HEAD指向的分支指针
      - 分支版本控制
        - 如果在testing分支上提交代码，那么会commit所在分支来一个新的commit object
        - 而testing分支指正会指向最新的commit object
        - HEAD就指向testing分支指针，而master指针还是指向之前的那个commit object
      - 分支切换 — git checkout master
        - 会切换回master分支，此时HEAD指针会指向master指针
        - 同时working目录里的文件的commit object，对应的tree和blob中的blob，也就是对应的文件快照恢复到工作区中来
      - 分支合并
        - 其实就是将两个分支当前指向的两个commit的代码，合并到一起，形成那一个新的commit
      - 新建分支
        - 新建分支只不过是多一个指针而已，指向当前的这个commit
      - origin/远程仓库
        - 概述
          - 远程版本库的分支，在本地都有远程分支（remote-tracking 分支）
          - 本地的master分支，对应的远程分支就是origin/master
          - 比如本地的feature/iss53分支，对应的远程分支就是origin/feature/iss53
          - 本地仓库和远程仓库之间推送和拉取代码，实际上就是在同步看看commit提交分支，包括进行一些分支指针的合并
        - 远程与本地同步
          - git fetch origin — 更新origin/master，origin/master会指向远程版本库的master最新的那个commit
          - git merge origin/master — 将本地最新远程仓库分支合并
          - 以上两个命令一起执行相当于git pull
  - 命令
    - 提交流程
      - 概述
        - git config 就是修改 git中的对应config文件的配置信息
      - git config 修改 — 例
        - git config
          - 加 --global
            - 就是说整个电脑用户范围内的配置，对机器上的其他用户来说的
          - 加 --system
            - 就是对当前机器上所有用户都生效
        - 例
          - 就是对当前所在的git项目本身生效
          - git config --global user.name "zhss"
          - 修改/添加 用户及邮箱
          - git config --global user.email "zhss@163.com"
      - config查看
        - 加 --list
          - 查看所有的配置项
        - 加 配置名
          - 单查看某个配置项
          - 加 git config user.name
    - 分支管理
      - 概述
        - 分支指针
        - 分支版本
        - 分支切换
        - 分支合并
        - 新建分支
      - git pull
        - 远程仓库commit拉取到本地(更新)
      - git add --all
        - 将工作区所有修改/新增加入到暂存区
        - 加 --all 如果有文件被删除，也会将文件被删除的状态加入暂存区中
      - git commit -m "one commit"
        - 将暂存区代码提交到工作区
      - git push
        - 本地commit推送到远程仓库/推送
      - git branch
        - 加 testing
          - 创建一个新的分支
          - 此时这个分支的指针会指向当前你所在的分支所指向的commit object上
        - 加 -u origin/分支名称
          - 可以让当前本地分支track(追踪)某个指定的远程分支
        - 加 -vv
          - 查看每个本地分支track的远程分支
        - 加 -d 分支名称
          - 命令删除一个本地分支
        - 加 -D 分支名称
          - 强制删除一个没有合并到目前的分支的分支
          - 如果你有一个分支，还没有合并到master分支去，此时git不让你删除的，可以使用 git branch -D
      - git merge 分支名称
        - 将分支代码合并到当前所在分支上去
        - 可选择本地或者本地追踪的远程分支合并
      - git checkout
        - 加 分支名称
          - 切换分支
        - 加 -b 分支名称
          - 创建本地分支同时切换到该分支
          - git checkout -b; 相当于两个命令，git branch dev，git checkout dev，先创建分支，再切换到分支
        - 加 -b 分支名称 origin/分支名称
          - 将本地获取到一个分支origin/分支名称 关联起来
      - git fetch origin
        - 让本地和远程保持同步
        - 同时拉取下来所有最新的远程分支
      - git push
        - 加 -u origin 分支名称
          - 将本地分支推送到远程仓库，默认分支名称一般是master
          - 一这个选项是用来本地分支和远程分支关联起来
          - 第一次推送远程仓库记录分支，就是将本地分支推送到远程仓库里，形成一个同名的远程分支
        - 加 origin --delete 分支名称
          - 删除远程分支
      - git tag
        - 加 v1.0
          - 轻量级的tag，直接就指向了当前分支指向的那个commit object
        - 加 -a v1.1 -m 'test tag'
          - 重量级的tag，创建一个带注释的标签
          - 会生成一个新的tag object的hash
    - 版本控制
      - git log
        - 不加直接用
          - 查看提交历史
        - 加 --patch -2
          - 可以显示每次提交之间的diff
          - 同时--n可以指定显示最近几个commit
        - 加 --stat
          - 可以显示每次commit的统计信息，包括修改了几个文件，有多少行插入，多少行删除
        - 加 --pretty=oneline
          - 可以每个commit显示一行，就是一个commit SHA-1校验和一个提交说明
        - 加 --oneline
          - 显示一行
        - 加 --abbrev-commit
          - 显示标识符，就是一个SHA-1 hash值，默认是7位显示
        - 加 --graph
          - 显示图形化的commit历史
        - 加组合 --decorate --graph --all
          - 可以打印出整个commit树，同时告诉你各个分支当前指向哪个commit object
        - 加组合 --oneline --abbrev-commit --graph
          - 这是最有用的，可以看到整个commit树结构，包括标签和分支指针，就是只看commit的SHA-1和提交说明，同时SHA-1显示缩短
      - git reset
        - 加 --hard HEAD^
          - 回退上一个commit ^一个^两个，类推
        - 加 --hard HEAD~5
          - 退回到HEAD之前的倒数第5个commit的状态
        - 加 --hard d324644
          - 指定一个commit的hash值回退
        - git reset 只能查看当前HEAD之前的commit
      - git reflog
        - 查看所有分支的所有操作记录 — HEAD之前，之后都可以查看
        - 包括已经被删除的 commit 记录和 reset 的操作
      - git show
        - 加 master
          - 可以查看master分支指向的那个commit
        - git show HEAD^
          - 查看上一个版本那个commit
        - 指针相关组合令都可以冲着HEAD^、HEAD~数字、hash来指定
      - 其他命令
        - git pull --rebase
          - rebase：变基，就是改变commit之间依赖的基础commit
        - git rebase feature/001
        - git rebase --onto a22e8f8 9c4bf8c
        - git commit-tree 9c4bf8c^{tree}
        - 将原来的一个仓库拆分成两个仓库
        - 让多个项目共享一个子项目的代码修改权利
          - submodule等组合命令
          - Subtree等组合命令
            - git以前是推荐使用submodule来管理子项目，但是从1.5.2之后就推荐使用subtree了
            - subtree的优点，就是子项目的维护权限只来说几乎就是透明的，不需要引入太多的维护成本