版本控制系统与git 基本介绍

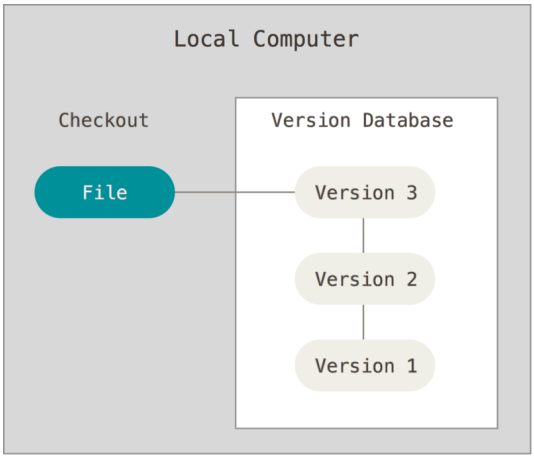
> 成文时间：2020-09-09
>
> 联系作者：qiao_jinming@foxmail.com
>
> 文章作者：乔金明

1. 什么是版本控制

- 记录项目从开始到结束的全部开发过程与改动细节，可以轻松的恢复到原先的样子
  - 记录所有文件的历史变化
  - 随时可以恢复到项目建立到当前的任意时间节点
- 可以确保每一个人开发项目保证同步，任何一个人都可以毫无顾虑的修改
  - 如果没有版本控制，当你修改好一个文件后，需要通知所有人改动的地方
- 备份（分布式版本控制系统），任何开发者都是一个完整的项目副本
  - 主服务器宕机后，可以通过任何一个开发者后备份这个项目

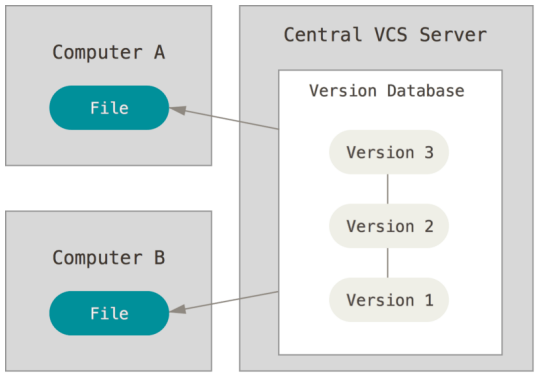
2. 版本控制的基本类别

- 本地版本控制系统
  - 原始的本地控制就是复制整个项目目录，一般是改个名字或者名字上加一个时间字符串用来区分不同的版本，但是很容易混淆这些项目，查看每个文件的修改历史也很麻烦，因此后来就发展成通过简单的数据库记录这些项目和文件的更新差异，进而发展成为了本地版本控制系统，例如：RCS(Revision Control System)
    - 优点：
      1. 简单，在目前的很多系统中都有内置，例如计算机系统的补丁集合
      2. 很适合管理文本，例如项目的配置文件等
    - 缺点：
      1. 很难进行整个大型项目的管理，只能管理少量的文件
      2. 支持的文件类型也很单一
      3. 无法进行网络传输等相关操作，就代表无法协同开发
    - 本地版本控制系统：图源

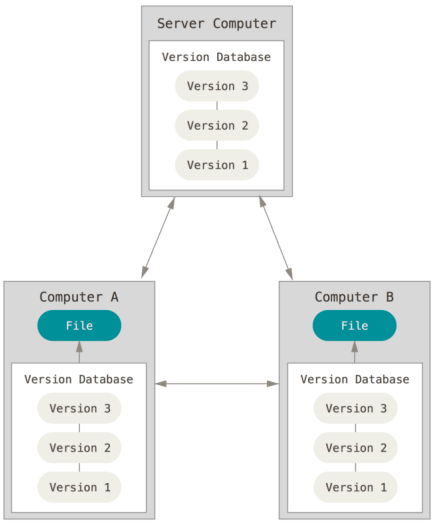


- 集中式版本控制系统

- 集中式版本的提出是因为本地版本控制无法令不同系统的开发者协同开发，为了解决这个问题，建立了一个集中管理的服务器，用来保存所有文件的修订版本，协同工作的开发者通过客户端连接这台服务器，拉取最新代码或者提交更新，例如：CVS，SVN(Subversion)，SVN的影响深远，现在还有很多重要的项目使用SVN进行版本管理
  - 优点：
    1. 管理员可以轻松掌控每一个开发者的权限，而且只需维护一台服务器
    2. 每个人都可以参与协同开发，可以看到其他人在做什么
  - 缺点：
    1. 服务器宕机后，所有人都无法提交更新，也就无法进行协同工作
    2. 服务器损坏后，将会丢失所有的数据包括变更历史，每个开发者只剩下单独的项目副本
    3. 必须进行联网才能获取历史变更记录和协同开发
  - 集中式版本控制系统：图源



- 分布式版本控制系统
  - 集中式版本控制受制于单一的服务器，所以又发展了一种新的形式，即每一台参与开发的客户机都把代码仓库完整的克隆下来，其中也包括历史记录，这样任何一处开发项目镜像发生故障时，都可以使用任何一个镜像来进行仓库的恢复，例如： Git、Mercurial、Bazaar、Darcs，当今趋势非常流行，例如github网站就是基于git的，由上面进行托管的项目规模可见一斑
    - 优点：
      1. 具有集中式版本控制的系统的全部优点
      2. 可以离线工作，便于协同开发，可以查看和恢复至历史的所有版本
      3. 每一个参与者都是一个完成的项目仓库
      4. 可以和若干个不同的远端代码仓库进行交互，设定不同的协作流程，分别与不同的小组合作
      5. 分布式记录的是文件快照，是按元数据存储，所以切换版本时是本地操作，直接切换指向数据即可；而集中式是按照文件存储，记录的文件差异，切换时需要逐级差异计算，速度慢并且需要网络进行传输，在大型项目的速度差异上相当大
    - 分布式版本控制系统：图源



3. git简史

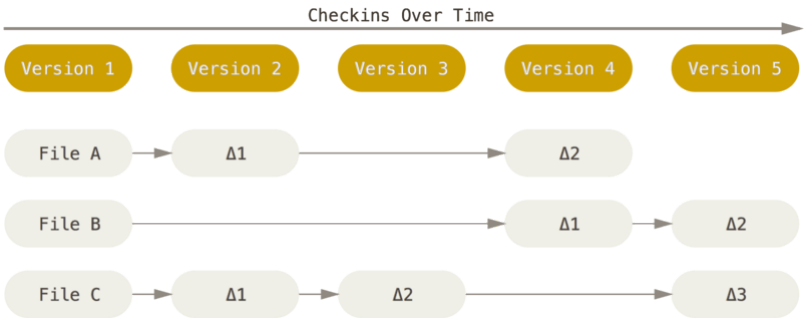
- 最开始的Linux内核有很多开发参与者，每次的开发版本都需要把源代码通过diff方式发给Linux之父Linus，通过手工方式进行代码合并，这些维护工作耗费了大量时间，这种状态维持了11年之久（1991-2002年）
- 虽然在此期间有诸如CVS、SVN等免费的版本控制系统，但是Linus十分反对，因为这些版本控制系统不仅慢而且需要联网，同时期也存在一些商用版本控制系统，相较于更加好用，但需要付费，不符合开源精神，也被Linus反对掉了
- 2002年，Linux社区的参与者越来越多，代码规模越来越大，手动维护已经很难了，而且遭到了很多参与者的诟病，所以Linux项目组开始启用了一个专有的分布式版本控制系统BitKeeper来进行代码维护和管理，BitKeeper的所有者BitMover公司也授权Linux社区免费使用
- 2005年，Linux社区的某些大牛试图破解BitKeeper协议，尤其是Andrew在进行破解时，被BitMover监控到，对于版权被侵犯非常愤怒，扬言要收回免费使用权
- Linus没有就此情况与BitMover达成和解，而是使用c语言花了两周时间写了一个分布式版本控制系统，即git诞生，一个月后，Linux系统源码由git进行维护管理
- 自此之后，git由于其开源和易用等特性迅速流行，2008年，github网站上线，提供免费的存储服务，无数的开源项目迁移至此网站，2011年，git实时安装量超过其他版本控制工具，2014年，使用git的程序员数量超过了使用SVN的程序员数量，2018年，github被微软收购

4. git创始人对系统制定的目标

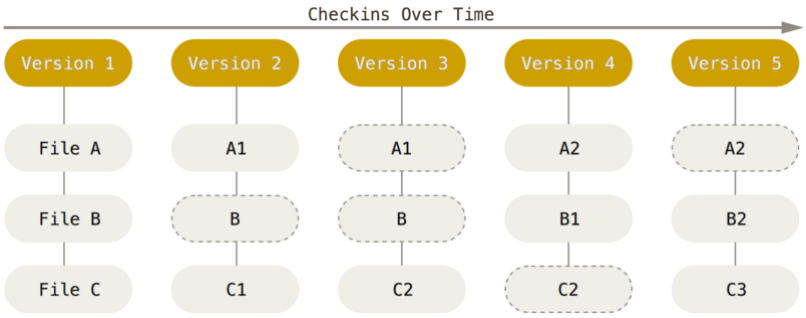
- 速度
- 简单的设计
- 对非线性开发模式的强力支持（允许成千上万个并行开发的分支）
- 完全分布式
- 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

5. git架构

- git是直接记录快照，而不是基于差异的比较
- 基于差异的版本控制
  - 一组基本文件和每个文件随时间逐步积累的差异，以文件变更列表的方式存储信息
  - 示意图：图源



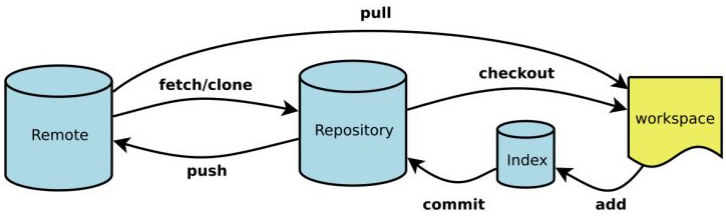
- 快照流
  - 将数据当做小型文件系统的一系列快照，每一次的更新或者保存项目状态时，基本都会对当时的全部文件创建一个快照并且保存这个快照的索引，如果文件没有被修改，git不会重新存储文件，而是保留一个链接指向之前存储的文件
  - 示意图：图源



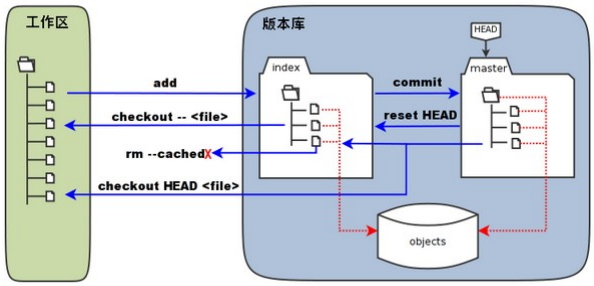
- git的操作几乎全部都是本地执行的，每一个git仓库都是完整的备份，即使在离线状态下也可以进行项目开发，可以在有网络时提交并合并自己的修改
- git在存储数据前都会计算校验和（通俗理解为数据的唯一身份证号），并通过校验和来进行引用，所以git不会任意更改任何文件，同时git也可以发现丢失的信息或者损坏的文件
    - git保证了完整性
    - 举个例子，git如果通过文件名来索引，当文件损坏时是发现不了的，但是通过校验和SHA-1（40位十六进制字符串）就可以很容易发现
- 如果提交了快照，git几乎不会丢失数据，如果定期推送到其他仓库，就会更安全，但是如果没有提交还是编辑阶段的话还是有可能丢失数据，因为git默认不会自动提交数据
    - 有了这个特性，无论在使用还是学习git过程中，都可以肆无忌惮的尝试，因为任何一个提交的状态都是可以回溯的
    - 但是在实际开发时最好别这样做，当然开发规范也不允许随意尝试

6. git的三种状态

- git在使用中具有三种状态，分别为：已修改(modified) 、 已暂存(staged)、和已提交(committed)
    - 已修改表示修改了文件，但还没保存到数据库中
    - 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中
    - 已提交表示数据已经安全地保存在本地数据库中
- 相应的，三种状态也使git存在三个阶段：工作区、暂存区以及 Git 存储库
    - 工作区是对项目的某个版本独立的具体数据，这些从 Git 仓库获取的文件，放在磁盘上供使用或修改
    - 暂存区是一个文件，保存了下次将要提交的文件列表信息， 按照 Git 的术语叫做“索引”，不过一般说法还是叫“暂存区”
    - Git 仓库目录（存储库）是 Git 用来保存项目的元数据和对象数据库的地方，这是 Git 中最重要的部分，从其它计算机克隆仓库时，复制的就是这里的数据
- git原理/流程图：图源

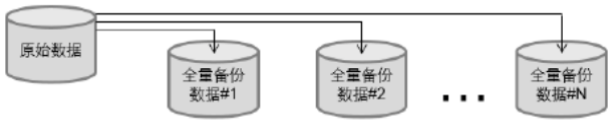


- 基本操作流程：
    1. 在工作区中进行文件编辑与修改
    2. 将想要的下次提交的更改进行选择性暂存，即将更改的部分添加到暂存区
    3. 提交更新，将暂存区中的快照永久性的存储到git目录
    4. 将更新推送到远程仓库，或者从远程仓库拉取更新
- git三个阶段对应本地的目录关系
    - 工作区：本地可以看到的项目目录
    - 暂存区：项目根目录下隐藏文件夹.git/下的stage或者index
    - 版本库：项目根目录下隐藏文件夹.git/
- git暂存区与版本库之间的关系：图源

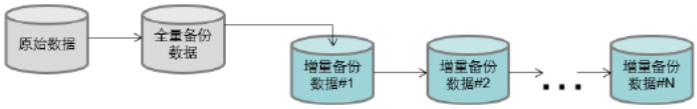


7. 快照与备份的区别

- 快照记录的是数据存储的某一时刻的状态，可以理解为照片，在某一时刻拍下照片，需要回溯时按照照片进行恢复即可，快照是一个特定时间点对数据状态的保护，只保存那些完整拷贝以外有变化的数据，如果数据没有变化，快照是不会保存额外数据的，快照可以看成是对某个特定时间点的数据的冻结
    - COW（Copy On Write），如果没有数据的写入，那么快照卷的指针还是指向原始卷的数据块，若有数据写入，系统会重新分配一个数据块，将老数据块整体拷贝到新分配的数据块，老数据块写入新数据，快照卷之前对应老数据块的指针指向新数据块，原始卷的指针不变，适合读多写少的业务
    - ROW（Redirect On Write），与COW不同，如果有数据修改，则将修改的数据写入新数据块，原始卷指针指向新数据块，快照卷指针不变，适合写密集型业务
- 备份是数据存储的某一个时刻的副本，这是另一份真实的相同内容数据，实实在在存储在磁盘上的，图源
    - 全量备份：复制的一整份项目副本

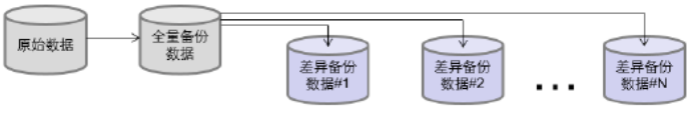


    - 增量备份：表现形式上与快照十分相似，但是快照之间记录的是两次项目地址之间的对应关系的差异，增量备份把新增地址所对应的数据也同时复制了一份



    - 差异备份：备份上一次全量备份之后发生变化的数据量，具有增量备份的备份时间短、节省磁盘空间的优势，又具有全量备份恢复时间短的特点，但是会存在一定量的重复数据，即前一份的差异备份数据都和后一份差异备份数据

      有重复



- 全量备份是对所有数据的一个拷贝，会将数据保存在不同的地方，和原始卷没有任何关系，是独立的存在；而快照还是依赖于原始卷，并且只有变化的数据块才会拷贝
- 增量备份是将上次备份之后变化的数据拷贝出来，和原始卷是没有依赖关系的，但是和上次的备份之间有依赖关系，会有一条依赖链接，一直链接到上次的全量备份
- 差异备份是将上次全量备份之后变化的数据拷贝出来，同样也是和原始卷没有依赖关系的，差异备份之间也没有任何关系，但是和上次的全量备份之间有依赖
- COW的快照之间没有依赖关系，但是都和原始卷有依赖，对于变化的数据块，会分配新的数据块将老的数据拷贝过去

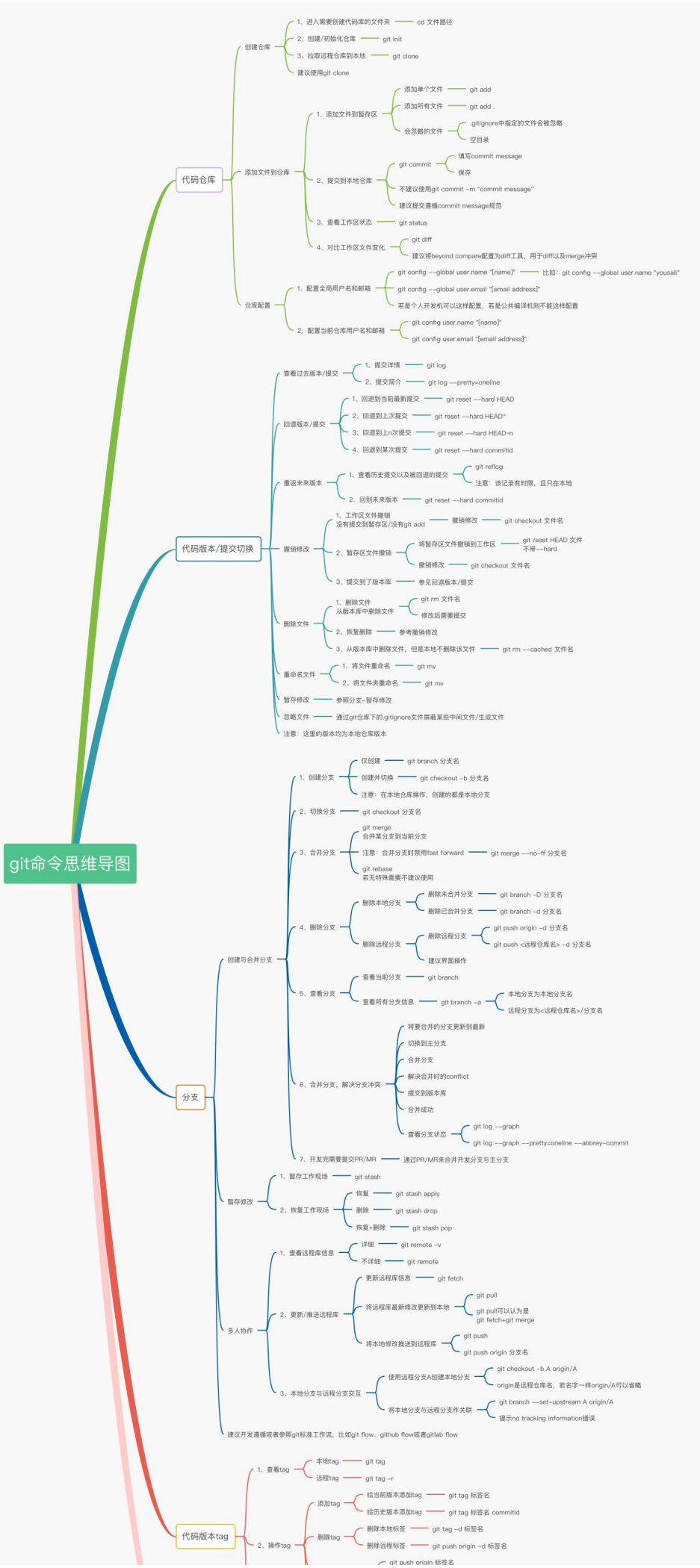
- ROW的快照之间会有快照链，也和原始卷有依赖，当删除快照的时候，因为要把数据都提交给原始卷，所以会把此快照时间点之后的快照都一并删除
- 快照与备份的优势对比
    - 备份的数据安全性更好：如果原始数据损坏，快照回滚是无法恢复出正确的数据的，而备份可以
    - 快照的速度比备份快得多：生成快照的速度比备份速度快的多，另外备份可能会带来的各种问题，例如IO占用、数据一致性等，所以很多备份软件是先生成快照，然后按照快照所记录的对应关系去读取底层数据来生成备份
    - 占用空间不同：备份会占用成倍的存储空间，而快照所占用的存储空间则取决于快照的数量以及数据变动情况，快照可能会只占用1%不到的存储空间，也可能会占用数十倍的存储空间，但平均来说快照占用空间更小

8. 为什么git需要暂存区

- 在开始使用开发时，都会自然产生这样的问题，git的暂存区作用，开始开发项目添加暂存区操作与提交操作经常同时使用，为什么不直接进行提交，（实际上可以直接提交，后续介绍），暂存区的作用是什么
    - 可以进行选择提交，比如先提交哪些后提交哪些，不至于混淆
    - 提高历史版本的清晰度，方便回滚
    - 提高原子性操作，保证每次提交都是干净的，降低提交的力度，这也是主要优点，原子性提交可以便捷的把整个项目还原到某个阶段

9. git命令思维导图

- git命令思维导图：图源


git命令思维导图
代码仓库
创建仓库
1、进入需要创建代码库的文件夹
cd 文件路径
2、创建/初始化仓库
git init
3、拉取远程仓库到本地
git clone
建议使用git clone
添加文件到仓库
1、添加文件到暂存区
添加单个文件
git add
添加所有文件
git add .
会忽略的文件
.gitignore中指定的文件会被忽略
空目录
2、提交到本地仓库
git commit
填写commit message
保存
不建议使用git commit -m "commit message"
建议提交遵循commit message规范
3、查看工作区状态
git status
4、对比工作区文件变化
git diff
建议将beyond compare配置为diff工具，用于diff以及merge冲突
仓库配置
1、配置全局用户名和邮箱
git config --global user.name "[name]"
比如：git config --global user.name "yousali"
git config --global user.email "[email address]"
若是个人开发机可以这样配置，若是公共编译机则不能这样配置
2、配置当前仓库用户名和邮箱
git config user.name "[name]"
git config user.email "[email address]"
代码版本/提交切换
查看过去版本/提交
1、提交详情
git log
2、提交简介
git log --pretty=oneline
回退版本/提交
1、回退到当前最新提交
git reset --hard HEAD
2、回退到上次提交
git reset --hard HEAD^
3、回退到上n次提交
git reset --hard HEAD~n
4、回退到某次提交
git reset --hard commitid
重返未来版本
1、查看历史提交以及被回退的提交
git reflog
注意：该记录有时限，且只在本地
2、回到未来版本
git reset --hard commitid
撤销修改
1、工作区文件撤销
没有提交到暂存区/没有git add
撤销修改
git checkout 文件名
2、暂存区文件撤销
将暂存区文件撤销到工作区
git reset HEAD 文件
不带--hard
撤销修改
git checkout 文件名
3、提交到了版本库
参见回退版本/提交
删除文件
1、删除文件
从版本库中删除文件
git rm 文件名
修改后需要提交
2、恢复删除
参考撤销修改
3、从版本库中删除文件，但是本地不删除该文件
git rm --cached 文件名
重命名文件
1、将文件重命名
git mv
2、将文件夹重命名
git mv
暂存修改
参照分支-暂存修改
忽略文件
通过git仓库下的.gitignore文件屏蔽某些中间文件/生成文件
注意：这里的版本均为本地仓库版本
分支
创建与合并分支
1、创建分支
仅创建
git branch 分支名
创建并切换
git checkout -b 分支名
注意：在本地仓库操作，创建的都是本地分支
2、切换分支
git checkout 分支名
3、合并分支
git merge
合并某分支到当前分支
注意：合并分支时禁用fast forward
git merge --no-ff 分支名
git rebase
若无特殊需要不建议使用
4、删除分支
删除本地分支
删除未合并分支
git branch -D 分支名
删除已合并分支
git branch -d 分支名
删除远程分支
删除远程分支
git push origin -d 分支名
git push <远程仓库名> -d 分支名
建议界面操作
5、查看分支
查看当前分支
git branch
查看所有分支信息
git branch -a
本地分支为本地分支名
远程分支为<远程仓库名>/分支名
6、合并分支，解决分支冲突
将要合并的分支更新到最新
切换到主分支
合并分支
解决合并时的conflict
提交到版本库
合并成功
查看分支状态
git log --graph
git log --graph --pretty=oneline --abbrev-commit
7、开发完需要提交PR/MR
通过PR/MR来合并开发分支与主分支
暂存修改
1、暂存工作现场
git stash
2、恢复工作现场
恢复
git stash apply
删除
git stash drop
恢复+删除
git stash pop
多人协作
1、查看远程库信息
详细
git remote -v
不详细
git remote
2、更新/推送远程库
更新远程库信息
git fetch
将远程库最新修改更新到本地
git pull
git pull可以认为是
git fetch+git merge
将本地修改推送到远程库
git push
git push origin 分支名
3、本地分支与远程分支交互
使用远程分支A创建本地分支
git checkout -b A origin/A
origin是远程仓库名，若名字一样origin/A可以省略
将本地分支与远程分支作关联
git branch --set-upstream A origin/A
提示no tracking information错误
建议开发遵循或者参照git标准工作流，比如git flow、github flow或者gitlab flow
代码版本tag
1、查看tag
本地tag
git tag
远程tag
git tag -r
2、操作tag
添加tag
给当前版本添加tag
git tag 标签名
给历史版本添加tag
git tag 标签名 commitid
删除tag
删除本地标签
git tag -d 标签名
删除远程标签
git push origin -d 标签名
git push origin 标签名


推送到远端仓库
推送所有未提交的tag
git push origin --tags
更新到本地
git pull origin --tags
tag与branch的操作基本一致，因为tag就是一个仅可读的branch

其他生僻命令
git blame
git bisect
过二分查找定位引入 bug 的变更
git relog
...
可以使用git help查看git常用的命令，使用git help -a查看git可用的所有命令