Git配置

git config

- 查看

```
git config -l
```

- 系统config

```
git config --system --list
```

位置

```
/etc/gitconfig
C:\Program Files\Git\mingw64\etc\gitconfig
```

- 当前用户（global）配置

```
git config --global  --list
```

位置

```
~/.gitconfig
```

- 当前仓库配置信息

```
git config --local  --list
```

位置

```
.git/config
```

设置

- 设置用户名与邮箱

```
git config --global user.name "etony"  #名称
git config --global user.email etony.an@gmail.com   #邮箱
--global为全局配置，不加为某个项目的特定配置
```

- 添加或删除配置项

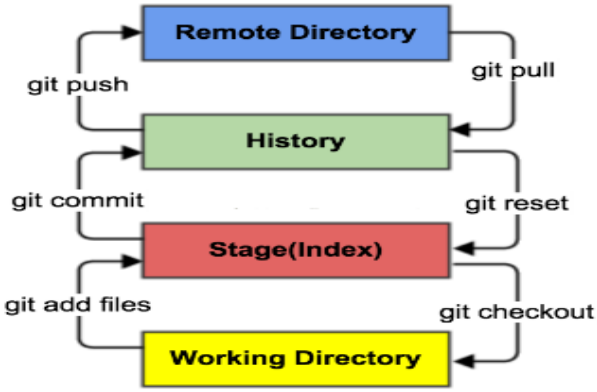
```
git config [--local|--global|--system]  section.key value
[--local|--global|--system]  #可选的，对应本地，全局，系统不同级别的设置
section.key #区域下的键
value #对应的值
--local 项目级
--global 当前用户级
--system 系统级
```

```
git config [--local|--global|--system] --unset section.key
```

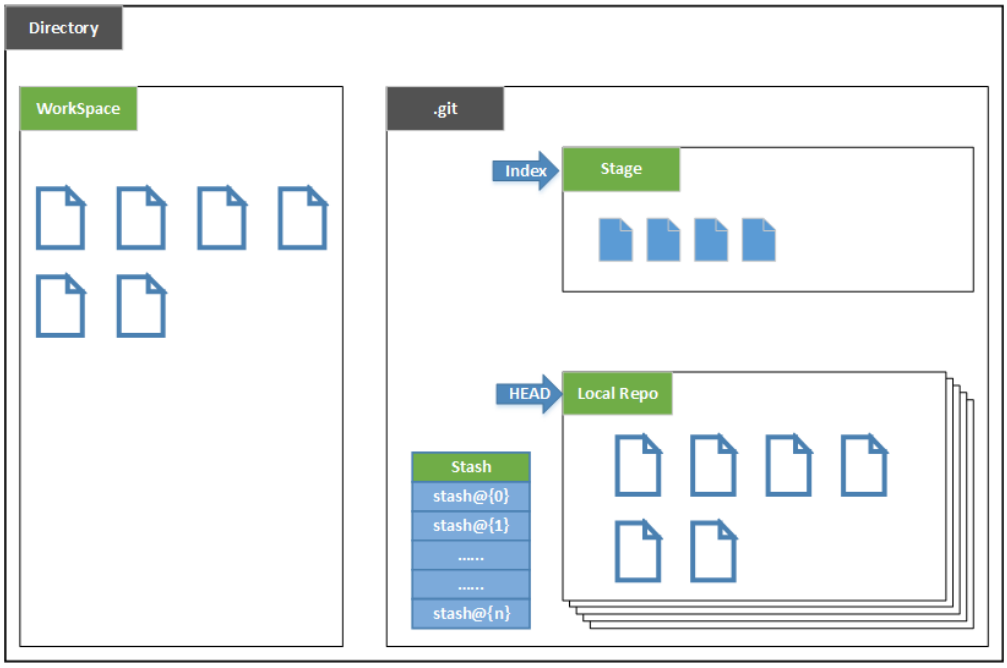
Git理论基础

工作区域

三个工作区域：工作目录（Working Directory）、暂存区(Stage/Index)、资源库(Repository或Git Directory)

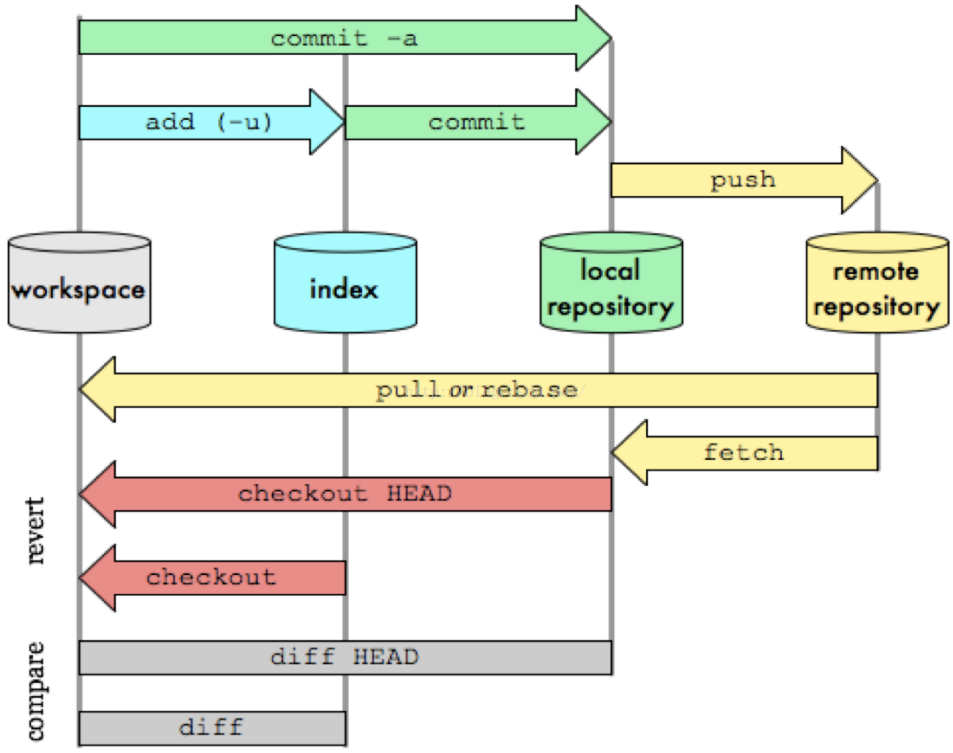


- Workspace：工作区，就是你平时存放项目代码的地方
- Index / Stage：暂存区，用于临时存放你的改动，事实上它只是一个文件，保存即将提交到文件列表信息
- Repository：仓库区（或本地仓库），就是安全存放数据的位置，这里面有你提交到所有版本的数据。其中HEAD指向最新放入仓库的版本
- Remote：远程仓库，托管代码的服务器，可以简单的认为是你项目组中的一台电脑用于远程数据交换



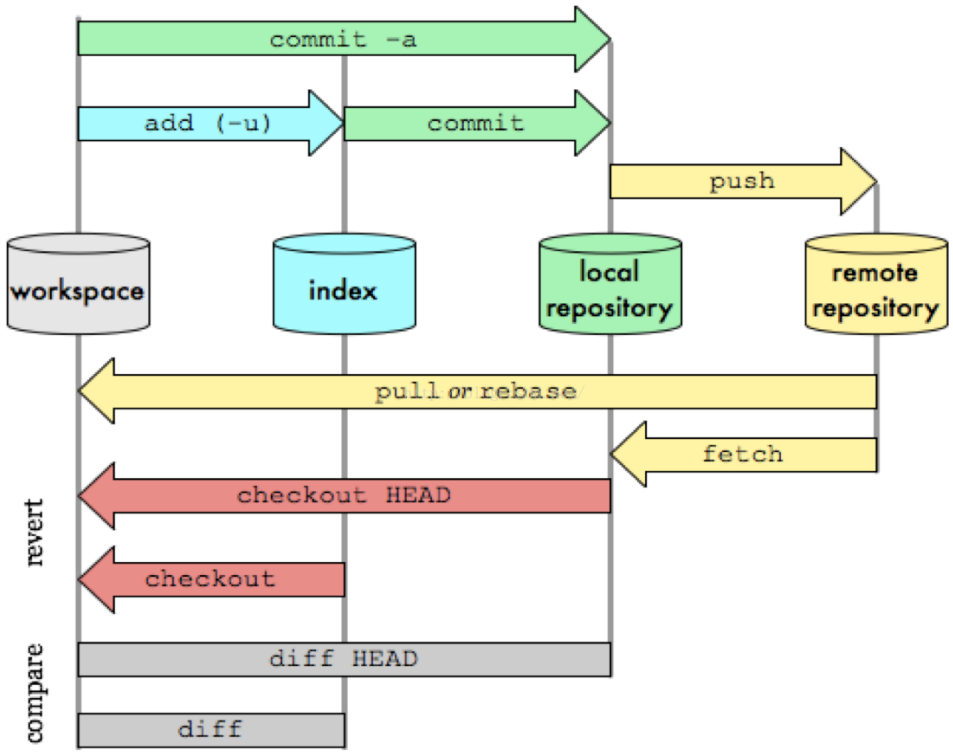
- Directory：使用Git管理的一个目录，也就是一个仓库，包含我们的工作空间和Git的管理空间。
- WorkSpace：需要通过Git进行版本控制的目录和文件，这些目录和文件组成了工作空间。
- .git：存放Git管理信息的目录，初始化仓库的时候自动创建。
- Index/Stage：暂存区，或者叫待提交更新区，在提交进入repo之前，我们可以把所有的更新放在暂存区。
- Local Repo：本地仓库，一个存放在本地的版本库；HEAD会只是当前的开发分支（branch）。
- Stash：隐藏，是一个工作状态保存栈，用于保存/恢复WorkSpace中的临时状态

工作流程



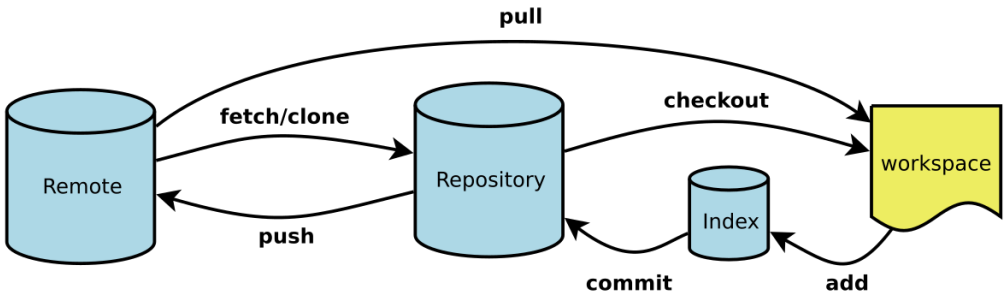
git管理的文件有三种状态：已修改（modified）,已暂存（staged）,已提交(committed)

图解教程



Git操作

常用指令



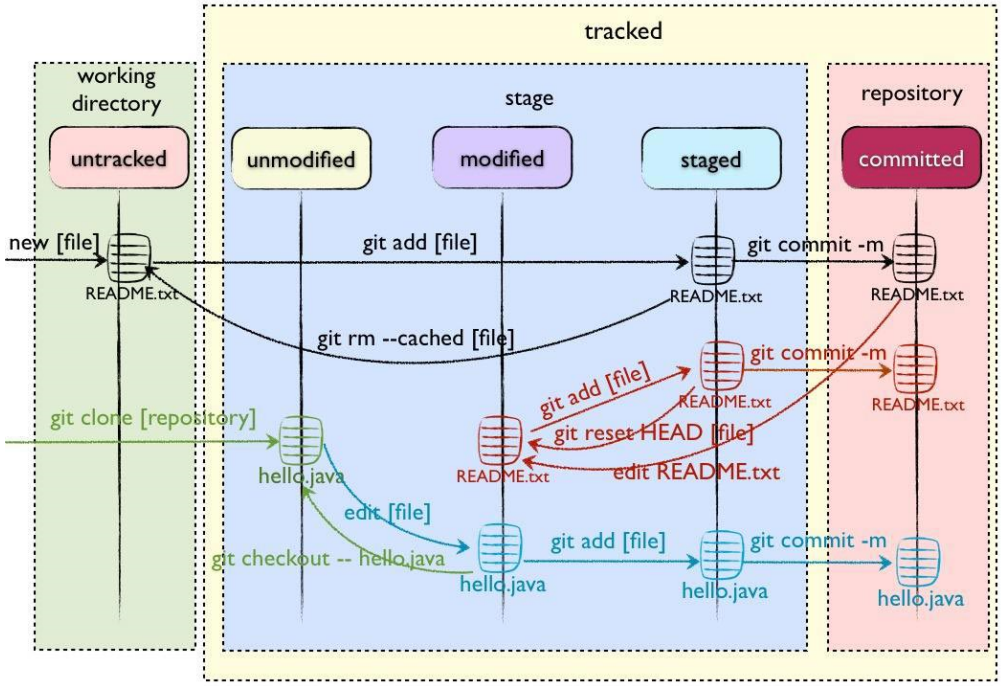
- 创建GIT仓库

```
git init #创建全新仓库
it代码库
git init [project-name]
```

- 克隆远程仓库

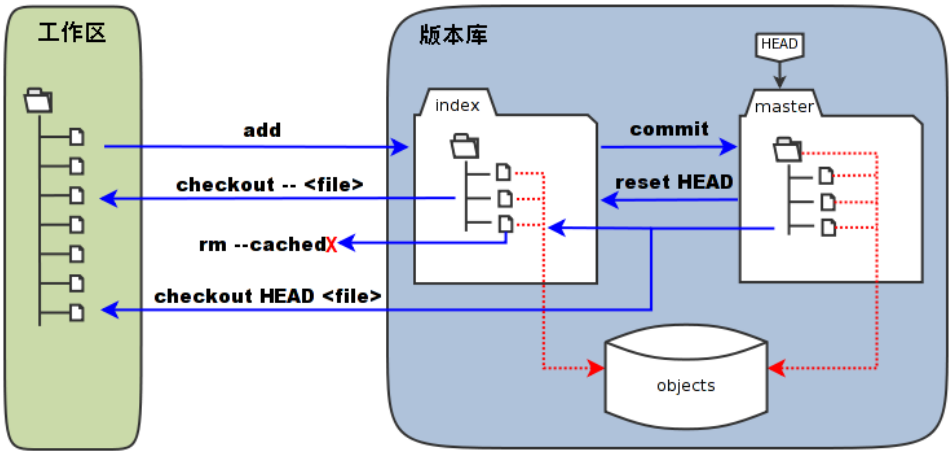
```
git clone [url]
```

文件操作



- 文件4种状态
  - Untracked: 未跟踪, 此文件在文件夹中, 但并没有加入到git库, 不参与版本控制. 通过git add状态变为Staged.
  - UUnmodify: 文件已经入库, 未修改, 即版本库中的文件快照内容与文件夹中完全一致. 这种类型的文件有两种去处, 如果它被修改, 而变为Modified. 如果使用git rm移出版本库, 则成为Untracked文件
  - UModified: 文件已修改, 仅仅是修改, 并没有进行其他的操作. 这个文件也有两个去处, 通过git add可进入暂存staged状态, 使用git checkout 则丢弃修改过, 返回到unmodify状态, 这个git checkout即从库中取出文件, 覆盖当前修改
  - UStaged: 暂存状态. 执行git commit则将修改同步到库中, 这时库中的文件和本地文件又变为一致, 文件为Unmodify状态. 执行git reset HEAD filename取消暂存, 文件状态为Modified
- 移除文件与目录

```
git rm --cached <file> #直接从暂存区删除文件，工作区则不做出改变
git reset HEAD <file>...#如果已经用add 命令把文件加入stage了，就先需要从stage中撤销
git clean [options] #移除所有未跟踪文件,一般会加上参数-df，-d表示包含目录，-f表示强制清除。
git rm --cached readme.txt #只从stage中删除，保留物理文件
git rm readme.txt #不但从stage中删除，同时删除物理文件
git mv a.txt b.txt #把a.txt改名为b.txt
```



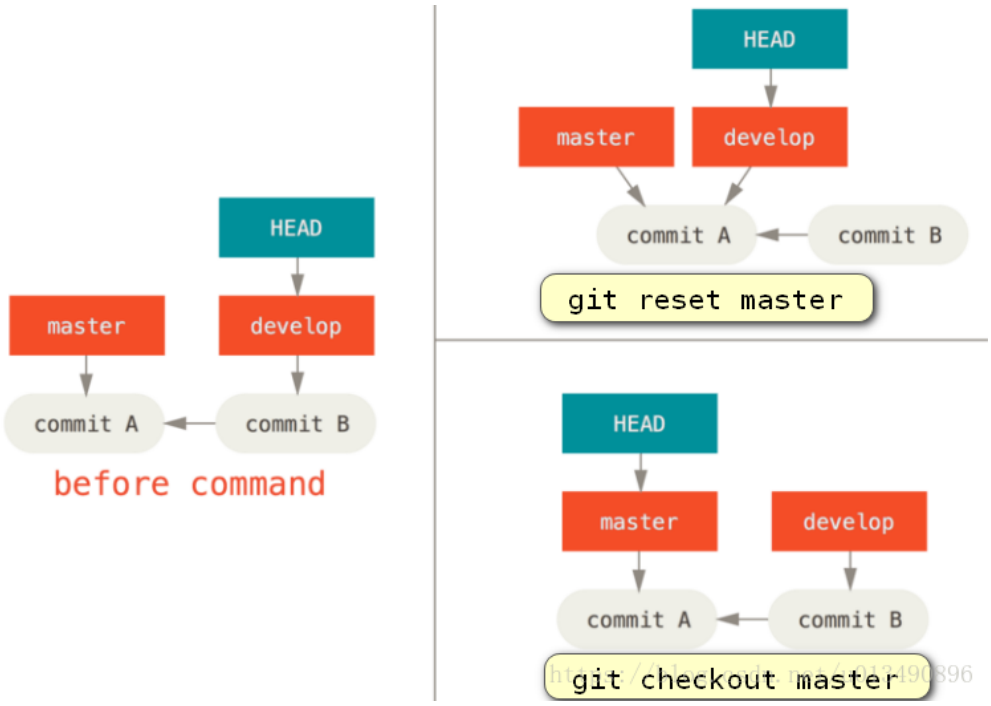
比较

```
git diff 1.txt  -> index (git add)
git diff HEAD 1.txt  -> localrepo (git commit)
git diff c3fb 1.txt -> localrepo (指定版本)
```

检出

```
git checkout 1.txt  <- index (git add)
git checkout HEAD 1.txt  <- localrepo (git commit) 回滚到复制最后一次提交
git checkout c3fb 1.txt  <- localrepo (指定版本)
```

checkout 与 rest




```
git checkout # 检出版本，本质上只是移动head指针
git reset #回退到历史某个版本
```

```
git reset --mixed <commit> (默认)：内容复制回暂存区
git reset --soft <commit>：内容区和暂存区保存原有状态
git reset --hard<commit>：内容复制回工作区和暂存区
git reset <branchName>^：回退到branchName的父提交
git reset <branchName>~n：回退到branchName前的第n次提交
```

忽略文件

```
.gitignore
```

提交

- 撤销提交

```
放弃工作区和index的改动，同时HEAD指针指向前一个commit对象
git reset --hard HEAD~1#撤销上一次的提交
git revert <commit-id> #把指定的提交的所有修改回滚，并同时生成一个新的提交。
```

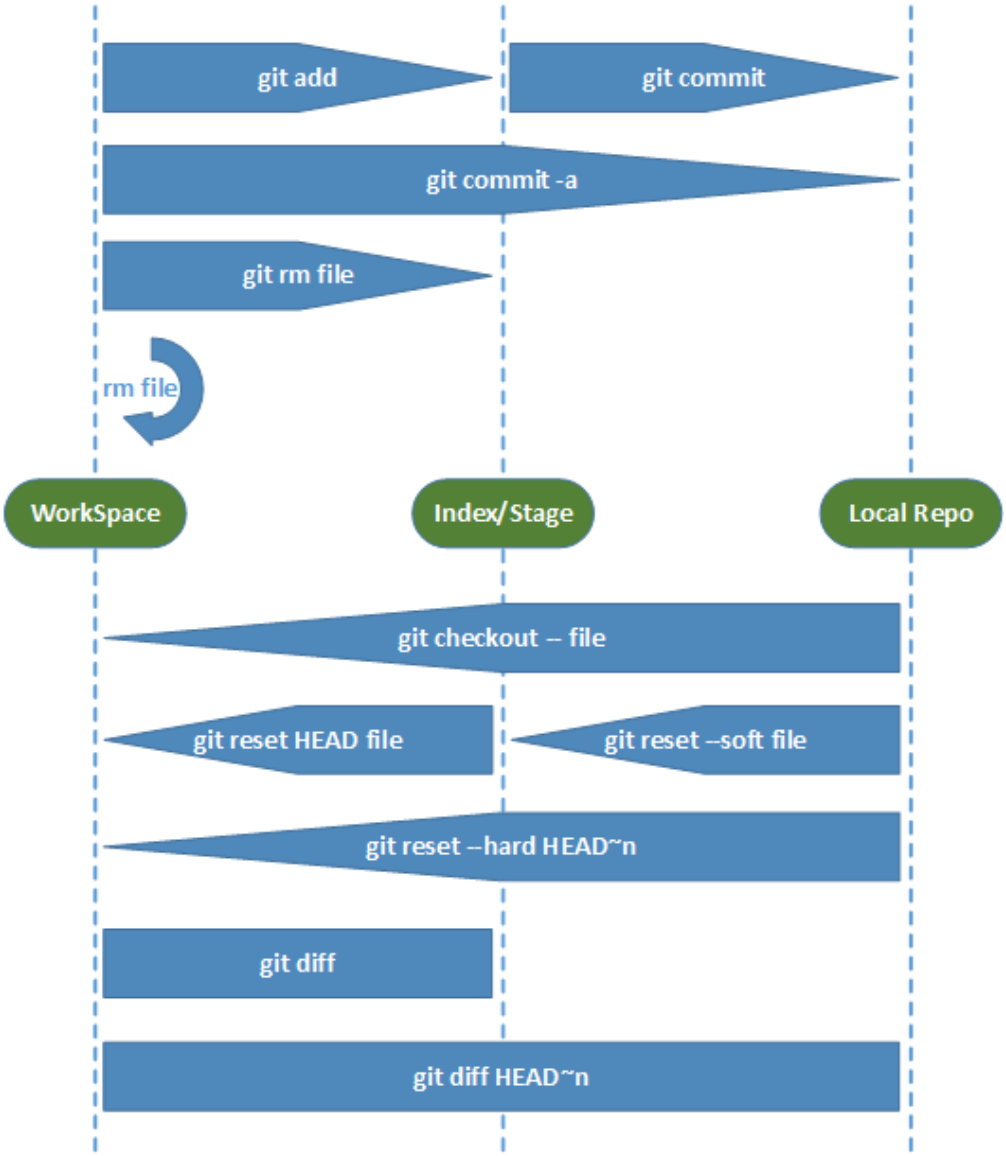
查看

```
git cat-file #提供仓库中对象实体的类型、大小和内容的信息
git cat-file (-t 显示对象的类型 | -s 显示对象的大小| -p 根据对象的类型显示其内容 )
<object>
```

日志与历史

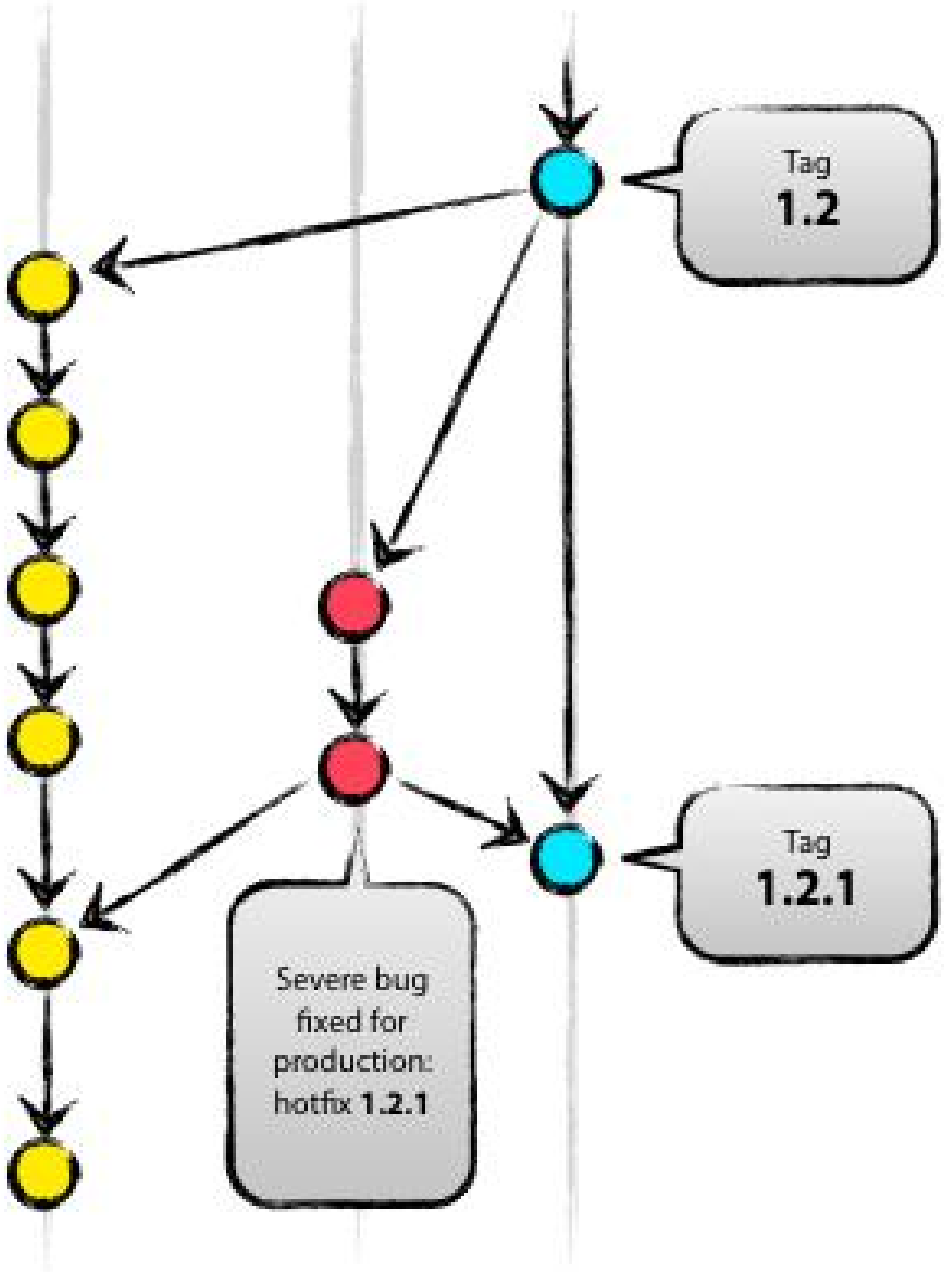
```
git log --graph #以图形化的方式显示提交历史的关系
git log -1 #显示1行。
git reflog #所有的分支的所有更新记录，包括已经撤销的更新。
```

文件操作小结



分支

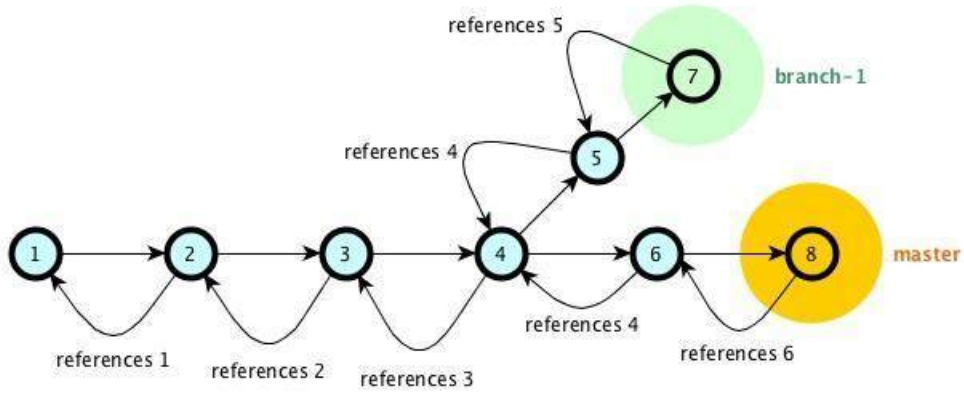

v0.1
v0.2
v1.0
Master
Release
Develop
Feature
Feature
develop
hotfixes
master


Tag
1.2
Tag
1.2.1
Severe bug
fixed for
production
hotfix 1.2.1

概念

- 分支其实上就是一个指针。简而言之，在git中分支只是一个指向单个commit的指针
- 分支引用

git的版本历史通过一个的commit往前推进存储。而分支引用则是相反从后王前引用的，如下图所示：



常用指令

```
git branch # 列出所有本地分支
git branch -r # 列出所有远程分支
git branch -a # 列出所有本地分支和远程分支
git branch [branch-name] # 新建一个分支，但依然停留在当前分支
git checkout -b [branch] # 新建一个分支，并切换到该分支
git branch [branch] [commit] # 新建一个分支，指向指定commit
git branch --track [branch] [remote-branch] # 新建一个分支，与指定的远程分支建立追踪关系
git checkout [branch-name] # 切换到指定分支，并更新工作区
git checkout - # 切换到上一个分支
git branch --set-upstream [branch] [remote-branch] # 建立追踪关系，在现有分支与指定的远程分支之间
git merge [branch] # 合并指定分支到当前分支
git cherry-pick [commit] # 选择一个commit，合并进当前分支
git branch -d [branch-name] # 删除分支
git push origin --delete [branch-name] # 删除远程分支
git branch -dr [remote/branch]
git branch -m [oldbranch] [newbranch] //重命名本地分支
```

创建分支

- 创建

```
git branch branch1
```

- 切换

```
git checkout branch1
```

- 创建并切换

```
git checkout -b branch1
```

- 删除

```
git branch -d branch1
```

- 分支信息

```
git branch -v ：显示现在的所有分支信息
```

git switch 与 git restore

因为目前 git checkout 命令承载了太多的功能，这让新手们感到困惑。git checkout 的核心功能包括两个方面，一个是分支的管理，一个是文件的恢复。这两个核心功能，未来将由 git switch 和 git restore 分别负责。

- 分支管理

```
#切换分支
git switch <分支名>  == git checkout <分支名> #

#创建分支
git switch -c <分支名>  == git checkout -b <分支名>
```

- 文件恢复撤销

```
# 暂存区（仅执行 add）
git restore --staged [file] : 表示从暂存区将文件的状态修改成 unstage 状态。当然，也可以不指定确切的文件 ，例如：
git restore --staged *.java 表示将所有暂存区的java文件恢复状态
git restore --staged . 表示将当前目录所有暂存区文件恢复状态
--staged 参数就是表示仅仅恢复暂存区的

#分支（已经commit提交）
git restore -s HEAD~1 READEME.md  // 该命名表示将版本回退到当前快照的前一个版本
git restore -s 91410eb9  READEME.md  // 改命令指定明确的 commit id ，回退到指定的快照中
git reset --soft HEAD^  // 该命令表示撤销 commit 至上一次 commit 的版本

#总结
git restore --worktree README.md    表示撤销 README.md 文件工作区的的修改    参数等同于 -W
git restore --staged README.md  表示撤销暂存区的修改，将文件状态恢复到未 add 之前 参数等同于 -S
git restore -s HEAD~1 README.md 表示将当前工作区切换到上个 commit 版本
git restore -s dbv213 README.md 表示将当前工作区切换到指定 commit id 的版本
```

帮助与代码统计

```
git status# 显示有变更的文件
git log# 显示当前分支的版本历史
git log --stat# 显示commit历史，以及每次commit发生变更的文件
git log -S [keyword]# 搜索提交历史，根据关键词
git log [tag] HEAD --pretty=format:%s # 显示某个commit之后的所有变动，每个commit占据一行
```

```
git log [tag] HEAD --grep feature# 显示某个commit之后的所有变动，其"提交说明"必须符合搜索条件
git log --follow [file]# 显示某个文件的版本历史，包括文件改名
git whatchanged [file]
git log -p [file]# 显示指定文件相关的每一次diff
git log -5 --pretty --oneline# 显示过去5次提交
git shortlog -sn# 显示所有提交过的用户，按提交次数排序
git blame [file]# 显示指定文件是什么人在什么时间修改过
git diff# 显示暂存区和工作区的差异
git diff --cached [file]# 显示暂存区和上一个commit的差异
git diff HEAD# 显示工作区与当前分支最新commit之间的差异
git diff [first-branch]...[second-branch]# 显示两次提交之间的差异
git diff --shortstat "@{0 day ago}"# 显示今天你写了多少行代码
git show [commit]# 显示某次提交的元数据和内容变化
git show --name-only [commit]# 显示某次提交发生变化的文件
git show [commit]:[filename]# 显示某次提交时，某个文件的内容
git reflog# 显示当前分支的最近几次提交
```

远程仓库

托管平台

- GitHub https://github.com/
- Gitlab https://about.gitlab.com/
- Bitbucket https://bitbucket.org/
- oschina http://git.oschina.net/
- coding https://coding.net/

远程仓库操作

- 常用操作指令

```
git fetch [remote]# 下载远程仓库的所有变动
git remote -v# 显示所有远程仓库
git remote show [remote]# 显示某个远程仓库的信息
git remote add [shortname] [url]# 增加一个新的远程仓库，并命名
git pull [remote] [branch]# 取回远程仓库的变化，并与本地分支合并
git push [remote] [branch]# 上传本地指定分支到远程仓库
git push [remote] --force# 强行推送当前分支到远程仓库，即使有冲突
git push [remote] --all# 推送所有分支到远程仓库
git remote  #简单查看远程---所有仓库（只能查看远程仓库的名字）
git remote show [remote-branch-name]#查看单个仓库
git remote add [branchname]  [url]#新建远程仓库
git remote rename [oldname] [newname]#修改远程仓库
git remote rm [remote-name]#删除远程仓库
git fetch [remote-name] #获取远程仓库数据(获取仓库所有更新，但不自动合并当前分支)
git pull (获取仓库所有更新，并自动合并到当前分支)
git push [remote-name] [branch]#上传数据，如git push origin master
```

- 克隆

```
git clone <版本库的网址>
```

- git remote

```
> git remote
origin
> git remote -v
origin  ssh://10.0.0.101:/home/git/python (fetch)
origin  ssh://10.0.0.101:/home/git/python (push)
> git remote show
> git remote show <主机名>
```

克隆版本库的时候，所使用的远程主机自动被Git命名为origin。如果想用其他的主机名，需要用git clone命令的-o选项指定。

- 修改主机名

```
git remote rename <原主机名> <新主机名>
```

- git fetch

```
git fetch <远程主机名> <分支名>
git fetch origin  #取回origin主机所有分支（branch）
git fetch origin master  #取回origin主机的master分支
```

```
git merge origin/master
# 或者
git rebase origin/master
#表示在当前分支上，合并origin/master。
```

- git pull

```
git pull <远程主机名> <远程分支名>:<本地分支名> # 如果远程分支是与当前分支合并，则冒号后面的部分可以省略。
```

- git push

```
git push <远程主机名> <本地分支名>:<远程分支名>
git push origin master #将本地的master分支推送到origin主机的master分支。如果后者不存在，则会被新建。
git push origin #将当前分支推送到origin主机的对应分支。
```

```
在git的全局配置中，有一个push.default属性
其用途分别为：
nothing - push操作无效，除非显式指定远程分支，例如git push origin develop（我觉得。。。可以给那些不愿学git的同事配上此项）。
current - push当前分支到远程同名分支，如果远程同名分支不存在则自动创建同名分支。
upstream - push当前分支到它的upstream分支上（这一项其实用于经常从本地分支push/pull到同一远程仓库的情景，这种模式叫做central workflow）。
simple - simple和upstream是相似的，只有一点不同，simple必须保证本地分支和它的远程upstream分支同名，否则会拒绝push操作。
matching - push所有本地和远程两端都存在的同名分支。
```

增强扩展

git-extras

- 主页 https://github.com/tj/git-extras
- 安装

```
git-extras-master> install.cmd  "C:\Program Files\Git" # git安装目录
# 安装目录 C:\Program Files\Git\mingw64\bin
# sudo apt-get install git-extras
```

- 常用命令
    - git setup 创建一个git repo,添加目录所有的文件，并作初始化提交
    - git ignore filename 创建 .gitignore 文件
    - gst == git status
    - gca 'message' == git commit -m 'message')
    - ga . == git add .
    - git summary 项目摘要信息
    - git effort 显示文件提交数量和活跃时间
    - git effort --above 10
    - touch history.md && git changelog 生成提交记录 history.md
    - git info 显示当前项目信息及配置信息

参考

- https://www.cnblogs.com/best/p/7474442.html
- https://www.cnblogs.com/yaozhongxiao/p/3811130.html
- https://www.cnblogs.com/zhumengke/articles/10801930.html
- http://marklodato.github.io/visual-git-guide/index-zh-cn.html
- https://www.jianshu.com/p/4142210eb2b2