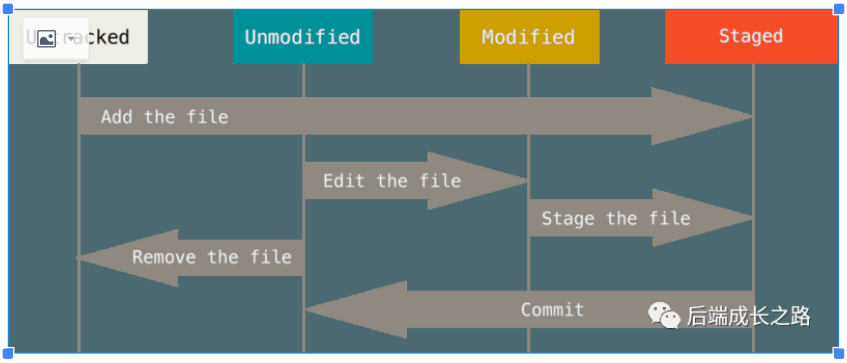
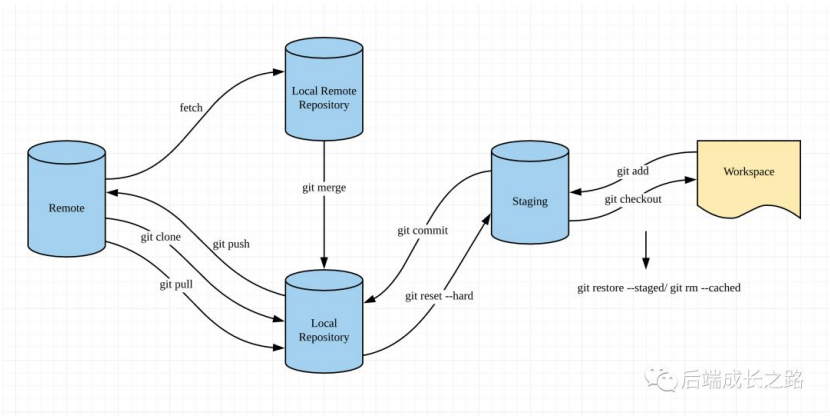
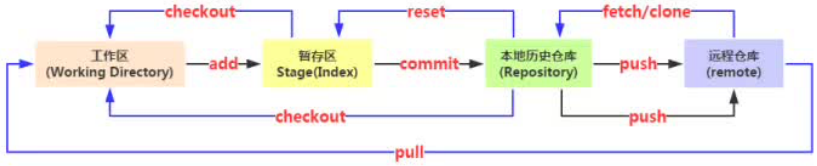
Git是一个分布式版本控制软件，最初由Linux Torvalds创作，于2005年以GPL发布。最初目的是为了更好地管理Linux内核开发而设计。

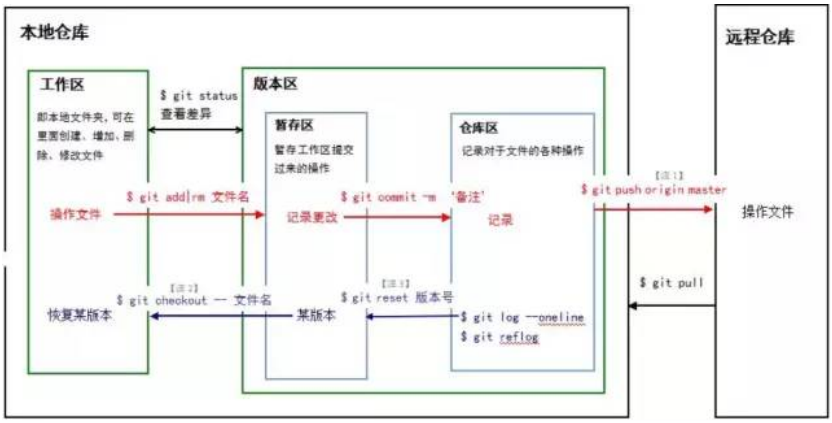
Git文件状态变化图解



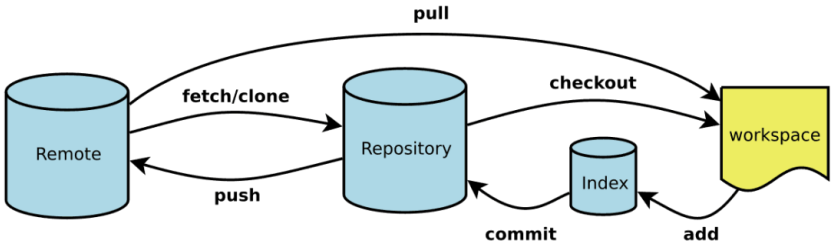
Git各个区域流程图解







- 工作区即磁盘上的文件集合。
- 版本区(版本库)即.git文件。
- 版本库=暂存区(stage)+分支(master)+指针HEAD



- Workspace：工作区
- Index / Stage：暂存区
- Repository：仓库区（或本地仓库）
- Remote：远程仓库

工作区

程序员进行开发改动的地方，是你当前看到的，也是最新的。
平常我们开发就是拷贝远程仓库中的一个分支，基于该分支进行开发。在开发过程中就是对工作区的操作。

暂存区

.git目录下的index文件, 暂存区会记录 **git add** 添加文件的相关信息(文件名、大小、timestamp...)，不保存文件实体, 通过id指向每个文件实体。可以使用 **git status** 查看暂存区的状态。暂存区标记了你当前工作区中，哪些内容是被git管理的。
当你完成某个需求或功能后需要提交到远程仓库，那么第一步就是通过 **git add** 先提交到暂存区，被git管理。

本地仓库

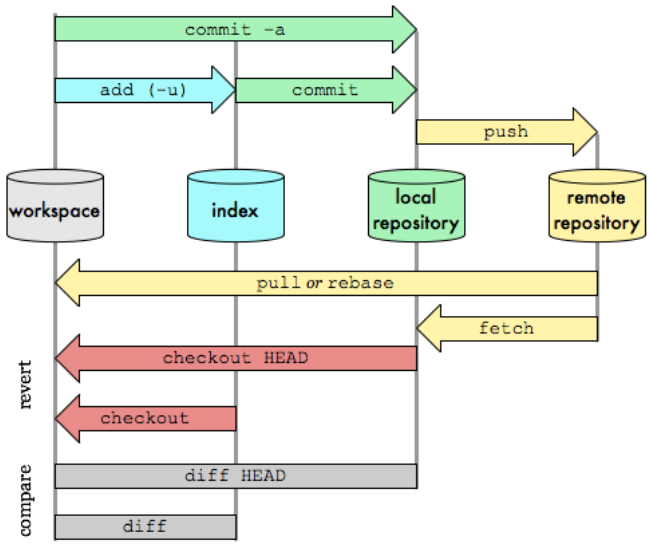
保存了对象被提交 过的各个版本，比起工作区和暂存区的内容，它要更旧一些。
git commit 后同步index的目录树到本地仓库，方便从下一步通过 **git push** 同步本地仓库与远程仓库的同步。

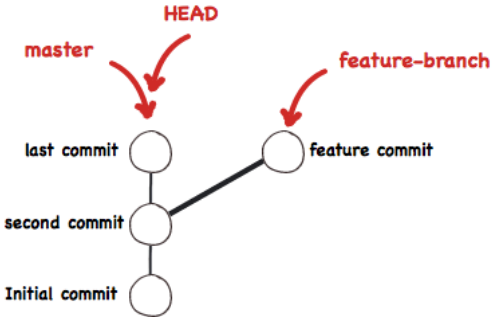
远程仓库

远程仓库的内容可能被分布在多个地点的处于协作关系的本地仓库修改，因此它可能与本地仓库同步，也可能不同步，但是它的内容是最旧的。

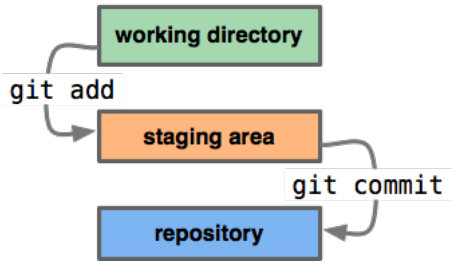
小结：

1. 任何对象都是在工作区中诞生和被修改；
2. 任何修改都是从进入index区才开始被版本控制；
3. 只有把修改提交到本地仓库，该修改才能在仓库中留下痕迹；
4. 与协作者分享本地的修改，可以把它们push到远程仓库来共享。、





HEAD,它始终指向当前所处分支的最新的提交点。你所处的分支变化了，或者产生了新的提交点，HEAD就会跟着改变。



#将工作区修改内容提交到暂存区，交由git管理

```
#添加当前目录的所有文件到暂存区,即添加所有的修改到Staging区
git add .
git add --all
#添加指定目录到暂存区，包括子目录
git add [dir]
#添加指定文件到暂存区
git add [file1]
#添加多个修改的文件到Staging区
git add [file1] [file2]
#添加所有src目录下main开头的所有文件到Staging区
git add src/main*
#添加所有被tracked文件中被修改或删除的文件信息到暂存区，不处理untracked的文件
git add -u
#添加所有被tracked文件中被修改或删除的文件信息到暂存区，包括untracked文件
```

```
git add -A
#进入交互界面模式，按需添加文件到缓存区
git add -i
```

#将暂存区的内容提交到本地仓库，并使得当前分支的HEAD向后移动一个提交点

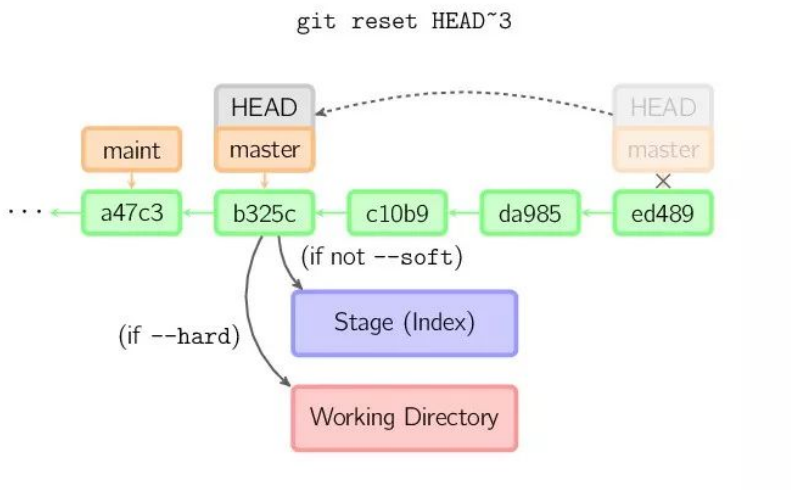
```
#提交暂存区到本地仓库，message代表说明信息
git commit -m [message]
#跳过缓存区操作，直接把工作内容提交到本地仓库
git commit -a -m [message]
#提交暂存区的指定文件到本地仓库
git commit [file1] -m [message]
#使用一次新的commit，替代上一次提交
git commit --amend -m [message]
```

#分支操作

```
#列出所有本地分支
git branch
#列出本地所有分支，并显示最后一次提交的哈希值
git branch -v
#在-v的基础上，并且显示上游分支的名字
git branch -vv
#列出所有远程分支
git branch -r
#列出所有本地分支和远程分支
git branch -a
#新建一个分支，但依然停留在当前分支
git branch [branch-name]
#新建一个分支，并切换到该分支
git checkout -b [branch-name]
#设置上分支上游
git branch --set-upstream-to origin/master
#新建一个分支，与指定的远程分支建立追踪关系
git branch -track [branch][remote-branch]
#创建本地分支并关联远程分支
git checkout -b local-branch orgin/remote-branch
#切换到指定分支，并更新工作区
git checkout [branch-name]
#丢弃修改
git checkout -- [filename]
#删除分支
git branch -d [branch-name]
#删除远程分支
git push origin --delete [branch-name]
```

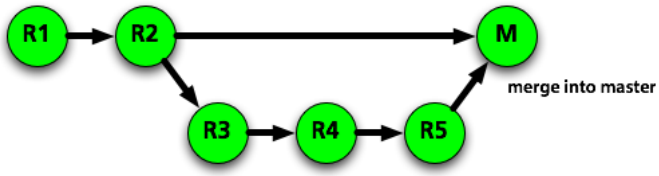
#上传本地仓库分支到远程仓库分支，实现同步

```
#上传本地指定分支到远程仓库
git push [remote][branch]
#强行推送当前分支到远程仓库，即使有冲突
git push [remote] --force
#推送所有分支到远程仓库
git push [remote] -all
#同步远程仓库
git push -u origin master
```



#把当前分支指向另一个位置，并且相应的变动工作区和暂存区(代码回滚)

```
#恢复成上次提交的版本
git reset HEAD^
#恢复成上上次提交的版本，就是多个^
git reset HEAD^^
git reflog
#只改变提交点，暂存区和工作目录的内容都不变
git reset --soft [commit]
#改变提交点，同时改变暂存区的内容
git reset --mixed [commit]
#暂存区、工作区的内容都会被修改到与提交点完全一致的状态
git reset --hard [commit]
#让工作区回到上次提交时的状态
git reset --hard HEAD
```



#分支合并

```
#merge之前先拉一下远程仓库最新代码
git fetch [remote]
#合并指定分支到当前分支
git merge [branch]
#加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并
git merge --no-ff -m "merge with no-ff" dev
```

一般在merge之后，会出现conflict，需要针对冲突情况，手动解除冲突。主要是因为两个用户修改了同一文件的同一块区域。

rebase又称为衍合，是合并的另外一种选择。
在开始阶段，我们处于new分支上，执行git rebase dev，那么new分支上新的commit都在master分支上重演一遍，最后checkout切换回到new分支。这一点与merge一样的，合并前后所处分支没有改变。git rebase dev，通俗讲就是new分支想站在dev的分支继续下去。rebase有需要手动解决冲突。

rebase与merge区别：

现在我们有这样的两个分支，test和master，提交如下：

```
        D——E test
       /
A——B——C——F master
```

在master执行 **git merge test**，然后会得到如下结果：

```
        D————E
       /      \
A——B——C——F——G   test, master
```

在master执行 **git rebase test**，然后得到如下结果：

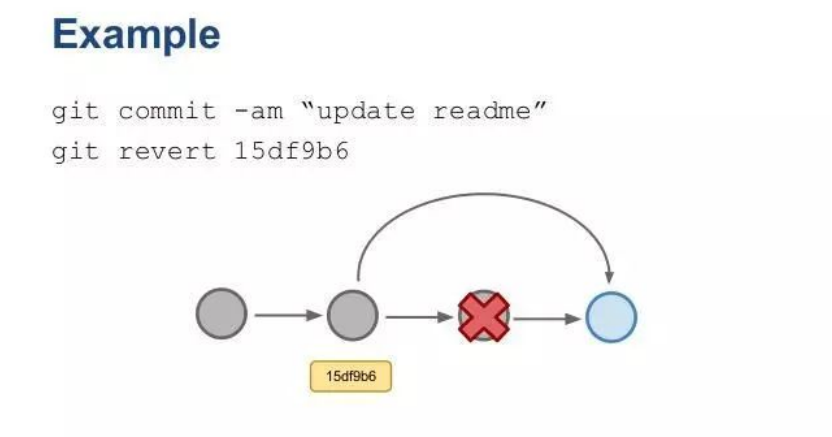
```
A——B——D——E——C'——F'   test, master
```

可以看到，merge操作会生成一个新的节点，之前的提交分开显示。而rebase操作不会生成新的节点，是将两个分支融合成一个线性的提交。

如果你想要一个干净的，没有merge commit的线性历史树，那么你应该选择git rebase 如果你想保留完整的历史记录，并且想要避免重写commit history的风险，你应该选择使用git merge。

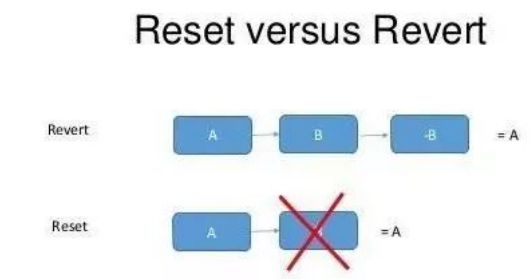
标签命令

```
#创建带有说明的标签
git tag -a v1.4 -m 'my version 1.4'
#打标签命令，默认为HEAD
git tag tag-name
#显示所有标签
git tag
#给某个commit版本添加标签
git tag tag-name commit-id
#删除标签
 git tag -d tag-name
 #显示某个标签的详细信息
 git show LABEL
```



git revert用一个新提交来消除一个历史提交所做的任何修改。

```
#撤销最近的一个提交，即撤销前一次commit
git revert HEAD
#撤销前前一次commit
git revert HEAD^
#撤销某次commit
git revert commit-id
```



revert和reset的区别：

- git revert是用一次新的commit来回滚之前的commit，git reset是直接删除指定的commit
- 在回滚这一操作上看，效果差不多。但是在日后继续merge以前老版本时有区别。因为git revert使用一次逆向的commit中和之前的提交，因此日后合并老的branch时，导致这部分改变不会再次出现，减少冲突。但是git reset是把某些commit在某个branch上删除，因而和老的branch再次merge时，那些被回滚的commit应该还会被引入，产生很多冲突。
- git reset是把HEAD向后移动了一下，而git revert是HEAD继续前进，只是新的commit的内容和要revert的内容正好相反，能够抵消要被revert的新内容。

git比较文件异同将文件添加到仓库：

```
#工作区与暂存区的差异
git diff
#工作区与某分支的差异，远程分支这样写：remotes/origin/分支名
git diff 分支名
#工作区与HEAD指针指向的内容差异
git diff HEAD
#工作区某文件当前版本与历史版本的差异
git diff 提交id 文件路径
#工作区文件与上次提交的差异(1.6版本前用 --cached)
```

```
git diff --stage
#查看从某个版本后都改动内容
git diff 版本TAG
#比较从分支A和分支B的差异(也支持比较两个TAG)
git diff 分支A  分支B
#比较两分支在分开后各自的改动
git diff 分支A...分支B
#另外：如果只能统计哪些文件被改动，多少行被改动，可以添加--stat参数
```

git查看历史记录：

```
#查看所有commit记录(SHA-A校验和，作者名称，邮箱，提交时间，提交说明)
git log
#查看最近多少次的提交记录
git log -p -次数
#简略显示每次提交的内容更改
git log --stat
#仅显示已修改的文件清单
git log --name-only
#显示新增，修改，删除的文件清单
git log --name-status
#让提交记录以精简的一行输出
git log --oneline
#图形展示分支的合并历史
git log --graph -all --online
#查询作者的提交记录(和grep同时使用要加一个--all--match参数)
git log --author=作者
#列出提交信息中包含过滤信息的提交记录
git log --grep=过滤信息
#和--grep类似，s和查询内容间没有空格
git log -S查询内容
#查看某文件的修改记录，找背锅专用
git log fileName
```

git 相关配置：

```
#安装完Git后第一件要做的事，设置用户信息(global可换成local在单独项目生效)
git config --global user.name "用户名"
git config --global user.email "用户邮箱"
#查看用户名是否配置成功
git config --global user.name
#查看邮箱是否配置成功
git config --global user.email
#设置当前项目提交代码的用户名
git config user.name name
#查看其它相关配置
#查看全局设置相关参数列表
git config --global --list
#查看本地设置相关参数列表
git config --local --list
#查看系统配置参数列表
git config --system --list
#查看所有Git的配置(全局+本地+系统)
git config --list
#配置git图形界面编码为utf-8
git config --global gui.encoding=utf-8
```

```
#显示git相关颜色
git config --global color.ui true
```

#其它命令

```
#查看远程库信息
git remote //-v显示更详细的信息
#增加一个新的远程仓库
git remote add name url
#初始化仓库，即在当前目录新建一个git仓库
git init
#打开git仓库图形界面
gitk
#删除版本库文件
 git rm [filename]
#从远程仓库克隆到本地
git clone git@github.com:git账号名/仓库名.git
#本地仓库内容推送到远程仓库
git remote add origin git@github.com:账号名/仓库名.git
#显示有变更的文件
git status
#显示当前分支的版本历史
git log
#显示暂存区和工作区的差异
git diff
#显示工作区域当前分支最新commit之间的差异
git diff HEAD
#选择一个commit，合并进当前分支
git cherry-pich [commit]
```

Git stash

```
#将修改过，为add到Staging区的文件，暂时存储起来
git stash
#恢复之前stash存储的内容
git stash apply
#s恢复同时删除stash内容
git stash pop
#保存stash，并写message
git stash save "stash test"
#查看stash了那些存储
 git stash list
#将stash@{1}存储的内容还原到工作区
 git stash apply stash@{1}
#删除stash@{1}存储的内容
git stash drop stash@{1}
#stash恢复后，stash内容部删除
git stash drop
#删除所有缓存的stash
 git stash clear
```

Git show

```
#查看指定标签的提交信息
git show tag-name
#查看具体的某次改动
```

```
git show commit-id
```

Git restore

```
#恢复第一次add的文件，同git rm --cached
git restore --staged file
#移除staging区的文件，同git checkout
git restore file
```

BUG分支

> 廖雪峰老师提到，工作中每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。但如果你手上有分支在工作中，你的上级要你改另外的分支的BUG。
>
> 你要把现在正在工作的分支保存下来，git stash,把当前工作现场"存储"起来，等以后恢复后继续工作。当你解决BUG后，git checkout other回到自己的分支。用git stash list查看你刚刚"存放"起来的工作去哪里了。

此时你要恢复工作：

- git stash apply恢复却不删除stash内容，git stash drop删除stash内容。
- git stash pop恢复的同时把stash内容也删了.
- 此时，用git stash list查看，看不到任何stash 内容。

总结：修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场

Git命令不能自动补全？（Mac版）

安装 bash-completion

```
brew install bash-completion
```

添加 bash-completion 到 ~/.bash_profile:

```
if [ -f $(brew --prefix)/etc/bash_completion ]; then
    . $(brew --prefix)/etc/bash_completion
fi
```

代码没写完，突然要切换到别的分支怎么办?

暂存未提交的代码

```
git stash
```

还原暂存的代码

```
git stash apply
```

怎么合并其他分支的指定Commit?

使用 cherry-pick命令

```
git cherry-pick 指定commit-id
```

本地临时代码不想提交，怎么一次性清空?

还原 未commit 本地更改的代码

```
git reset --hard
```

还原包含commit的代码，到跟远程分支相同

```
git reset --hard origin/master
```

已经提交的代码，不需要了，怎么当做没提交过?

还原到上次commit

```
git reset --hard ~HEAD
```

还原到当前之前的几次commit

```
git reset --hard ^3
```

强制推送到远程分支，确保没有其他人在push，不然可能会丢失代码

```
git push origin develop --force
```

怎么保证团队成员提交的代码都是可运行的?

这里想说的是使用 git hooks，一般在项目目录 .git/hooks，客户端可以使用hooks，控制团队commit提交规范，或者push之前，自动编译项目校验项目可运行。服务端可以使用hooks，控制push之后自动构建项目，merge等自动触发单元测试等。

git reset--hard命令，执行错了，能恢复吗?

当前commit log

误操作 `git reset--hard8529c`

执行 `git reflog`

还原到之前的样子

公司使用GitLab，平时还用GitHub，多账号SSH，如何配置?

编辑 `~/.ssh/config`文件 没有就创建

```
# github
Host github.com
Port 22
HostName github.com
PreferredAuthentications publickey
AddKeysToAgent yes
IdentityFile ~/.ssh/github_id_rsa
UseKeychain yes
User iisheng

# gitlab
Host gitlab.iisheng.cn
Port 22
HostName gitlab.iisheng.cn
PreferredAuthentications publickey
AddKeysToAgent yes
IdentityFile ~/.ssh/gitlab_id_rsa
UseKeychain yes
User iisheng
```

Git Commits如何变得清爽起来?

使用 `git rebase`，放弃 `git merge`。`git rebase`会更改commit历史，请谨慎使用。

下面的图是Guava项目的Commit记录。

如何修改已经提交的Commit?

原始git提交记录是这样的

执行 `git rebase-i070943d9465177f869359a52b01ec1cfcecaa41b`，对指定commitId之前的提交，进行修改

修改后git提交记录变成了这样