目录

[1 认识Git 4](#_Toc26145)

[1.1 Git的起源 4](#_Toc29335)

[1.2 Git是什么？ 4](#_Toc24404)

[1.3 Git的安装和配置 5](#_Toc12530)

[1.4 获得帮助 5](#_Toc18948)

[2 Git的基础操作 6](#_Toc5308)

[2.1 获取Git仓库 6](#_Toc29633)

[2.2 提交代码修改 6](#_Toc22104)

[2.3 查看提交历史 7](#_Toc14350)

[2.4 撤销操作 8](#_Toc7468)

[2.5 远程操作 8](#_Toc6908)

[2.6 打标签 9](#_Toc30272)

[2.7 遗留问题 9](#_Toc4552)

[3 Git分支 9](#_Toc29926)

[3.1 分支简介 10](#_Toc12636)

[3.2 分支管理 10](#_Toc30135)

[3.3 远程分支 11](#_Toc31201)

[3.4 变基 11](#_Toc32357)

[4 Git服务器 12](#_Toc28606)

[4.1 协议 13](#_Toc18741)

[4.1.1本地协议：共享文件系统 13](#_Toc30139)

[4.1.2 HTTP协议：智能HTTP协议、哑HTTP协议 13](#_Toc16663)

[4.1.3 SSH协议 13](#_Toc14631)

[4.1.4 Git协议 13](#_Toc7098)

[4.2 搭建Git服务器 14](#_Toc18474)

[4.2.1 简单的Git服务器，可供几个人使用 14](#_Toc26897)

[4.2.2 小型安装，实现了用户管理 14](#_Toc27563)

[4.2.3 配置Git服务器 14](#_Toc27677)

[5 分布式Git 15](#_Toc27000)

[5.1 常见Git工作流 16](#_Toc28417)

[5.2 常见项目贡献方式 16](#_Toc14477)

[5.3 维护项目 16](#_Toc5281)

[5.4 整合提交 17](#_Toc16556)

[5.5 打标签并发布 17](#_Toc31844)

[5.5.1 签名问题 18](#_Toc18343)

[5.5.2 生成一个构建号 18](#_Toc28874)

[5.5.3 准备一次发布 18](#_Toc523)

[5.5.4 制作提交简报 18](#_Toc24569)

[6 Git托管平台 18](#_Toc10764)

[6.1 以拉取为中心的特殊合作流程 19](#_Toc5487)

[7 Git工具 19](#_Toc22240)

[7.1 选择修订版本 20](#_Toc9845)

[7.1.1指定修订版本： 20](#_Toc24557)

[7.1.2引用日志 20](#_Toc20239)

[7.1.3祖先提交 20](#_Toc12921)

[7.1.4提交区间 21](#_Toc11128)

[7.2 贮藏和清理 21](#_Toc30137)

[7.3 签署工作 22](#_Toc23145)

[7.3.1配置签名 22](#_Toc30834)

[7.3.2应用签名 22](#_Toc14016)

[7.4 搜索 23](#_Toc31880)

[7.5 重写历史 23](#_Toc2886)

[7.6 重置 24](#_Toc22582)

[7.6.1 Git的三棵树：HEAD、Index、Working Directory 24](#_Toc31133)

[7.6.2 reset工作流程 25](#_Toc25018)

[7.6.3 通过路径来重置 25](#_Toc21914)

[7.7 高级合并 25](#_Toc450)

# 1 认识Git

## 1.1 Git的起源

1991-2002年，Linux内核开源项目有着众多的参与者，绝大部分维护工作都花在了补丁提交、保存归档这些繁琐的事物上。

2002-2005年，分布式版本控制系统Bitkeeper的公司授权Linux社区免费使用该系统。

2005至今，2005年Bitkeeper的公司收回了Linux社区的免费使用权，Linux的缔造者和开源社区的成员们被迫开发自己的版本控制系统，Git应运而生。

## 1.2 Git是什么？

Git是分布式版本控制系统，区别于传统的集中式版本控制系统：

1、Git相当于一个小型文件系统，直接记录快照，而非保存文件变更列表。

2、Git存储每个文件与初始版本的差异。

3、Git保证版本完整性，所以数据在存储前都会计算校验和（SHA-1散列），然后以校验和来引用。所以如果文件有丢失或者损坏，Git都能发现。

4、Git一般只添加数据。

5、Git的三种状态：已修改、已暂存、已提交。

6、Git项目的三个阶段：工作区、暂存区/索引、Git仓库。

工作区：是从该项目的Git压缩数据库中提取出来的某个版本的内容

暂存区：保存了下一次需要提交的文件列表信息

Git目录：Git用来保存项目元数据和对象数据库的地方

## 1.3 Git的安装和配置

1、Git的三个配置文件：

/etc/gitconfig：git config添加--system命令可以读写该配置文件（需要管理员权限）。

~/.gitconfig或~/.config/git/config：git config添加--global命令可以读写该文件。

.git/config：git config添加--local命令可以读写该文件（级别最高）。

2、git config --list --show-origin 查看git配置并显示它们所在的文件

## 1.4 获得帮助

git help config

git config --help

man git config

# 2 Git的基础操作

## 2.1 获取Git仓库

git init 对已存在的目录进行git初始化

git clone ssh:xxx/xxx

git clone user@xxx:xxx/xxx

git clone <https://xxx/xxx>

## 2.2 提交代码修改

git status 查看文件当前状态（参数 -s或--short可以使输出更精简）

git add 将已修改的已跟踪文件或者未跟踪文件加入暂存区（参数是文件名或者路径名）

git diff 显示尚未暂存的修改内容，工作区与暂存区相对比

git diff --staged/--cached 显示已暂存的修改内容，暂存区与版本库想对比

git commit 将暂存区内容提交到版本库

-v 选项将git diff 的输出以注释形式呈现在提交说明的文本中

-m 选项直接在命令行添加提交说明

-a 自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过add步骤

--amend 重新提交，将暂存区的内容提交并修改上次的提交说明

git config --global core.editor 设置用来提交说明的文本编辑器vim或者emacs

## 2.3 查看提交历史

git log 查看提交历史

-p/--patch 显示每次提交所引入的差异（git diff）

-<n> 显示n条提交历史

--stat 显示每次提交的文件修改总结（git status）

--pretty 显示不同于默认格式的方式展示提交历史

=oneline

=short

=full

=fuller

=format:"%h - %an, %ar : %s"定制显示格式

%H/%h 提交的完整哈希值/简写哈希值

%T/%t 树的完整哈希值/简写哈希值

%an 作者

%ar 作者修订日期，按多久以前的方式显示

%s 提交说明

--graph 形象展示分支发展历程，一般配合--pretty=format使用

--since/--until/--before 时间过滤器

--grep 关键词搜索过滤器

-S function\_name 显示添加过或删除了function\_name的提交

-- 限定文件

## 2.4 撤销操作

git rm 在暂存区中删除已跟踪文件，并在工作删除该文件

-f 用于删除之前修改过或已经放到暂存区的文件

--cached 在暂存区中删除已跟踪文件，但在工作区中保留

git mv a b 重命名，相当于：

mv a b

git rm a

git add b

git reset HEAD <file> 取消暂存，暂存区的文件恢复到和版本库保持一致

git checkout -- <file> 撤销修改，文件恢复到和暂存区保持一致

## 2.5 远程操作

git remote add <shortname> <url> 添加远程仓库

git fetch <remote> 抓取，将数据下载到本地仓库，但是不会自动合并或修改当前工作区，也不会在远程分支的基础上创建本地分支

git pull 拉取，自动抓取后合并远程分支到当前分支

git push <remote> <branch> 推送

git remote show origin 显示远程仓库的信息

git remote rename 修改远程仓库的简写名，相关跟踪的分支同步修改

git remote remove 移除远程仓库，相关的跟踪分支和配置信息也一起删除

## 2.6 打标签

git tag -a <tag\_name> -m <commit\_descriprion> 创建附注标签

git tag -a <tag\_name> -m <commit\_descriprion> <commit\_id> 根据历史提交记录创建tag

git tag <tag\_name> 创建轻量标签

git show <tag\_name> 查看标签信息

git push <remote> <tag\_name> 推送标签到远程服务器

git push <remote> --tags 推送所有标签

git tag -d <tag\_name> 删除本地标签

git push <remote> :refs/tags/<tag\_name> 更新远程仓库的tag标签，冒号前的空值推送到远程标签名

git push <remote> --delete <tag\_name> 删除远程仓库的tag标签

## 2.7 遗留问题

检出标签？

# 3 Git分支

## 3.1 分支简介

Git保存快照：

为每一个文件计算校验和，并在Git仓库中用blob对象保存文件快照。为根目录和每一个子目录计算校验和，并保存为树对象，记录目录结构和blob对象索引。创建一个提交对象，包含指向根目录树对象的指针和所有提交信息。第二次提交，提交对象中会包含指向上一次提交对象的指针。

Git的分支，本质上是指向提交对象的可变指针，默认分支名是master，每次进行提交，master分支自动向前移动。

## 3.2 分支管理

使用git branch可以创建一个可移动的新分支。HEAD，一个指针指向当前所在的本地分支。

git log --decorate查看各个分支当前所指对象

git checkout <branch> 切换分支

git log --oneline --decorate --graph --all 查看项目的分支分叉情况

git checkout -b <branch> 创建并切换分支，相当于：

git branch <branch>

git checkout <branch>

git checkout <branch> 当本地不存在该分支且有个名字与之匹配的远程分支时，相当于：

git checkout --track <remote>/<branch> 相当于：

git checkout -b <local\_branch> <remote>/<branch> 基于远程分支的基础上创建本地跟踪分支，并切换到本跟踪地分支

git merge <branch> 合并分支

git branch -d <branch> 删除分支

git mergetool 可视化合并工具

冲突文件解冲突以后，加入暂存区再提交

git branch -v 查看所有分支的最后一次提交

git branch --merged <branch> 查看已合并到当前分支的分支（可以删除，不会失去任何东西）

--no-merged <branch> 查看尚未合并到当前分支的分支

## 3.3 远程分支

git ls-remote <remote> 获取远程引用的完整列表

git remote show <remote> 获取远程分支的信息

git clone -o <remote> 拉取仓时为远程仓服务器命名

git remote -vv 列出本地所有分支及其跟踪分支

git fetch --all 所有本地分支从跟踪的远程分支上抓取数据

git pull 从当前分支跟踪的远程分支上抓取数据，并尝试合并，相当于：

git fetch + git merge

git push <remote> --delete <branch> 删除远程分支，通常只是移除指针，数据保留到垃圾回收运行

## 3.4 变基

变基：相对于直接合并（git merge），最终结果一致，但是提交历史更简洁

git rebase <base\_branch> <topic\_branch> 相当于：

git checkout <base\_branch>;git rebase <topic\_branch>

git checkout <topic\_branch>

git merge <base\_branch>

git rebase --onto master server client 取出client分支，找出从server分支分歧以后的提交，然后把这些提交在master分支上重放一遍，让client看起来像是直接基于 master修改一样

变基 vs. 合并:

只对尚未推送或分享给别人的本地修改执行变基操作清理历史， 从不对已推送至别处的提交执行变基操作，这样，你才能享受到两种方式带来的便利。

# 4 Git服务器

## 4.1 协议

服务器上的仓库通常是一个裸仓库，只包含.git子目录下的内容。

Git服务器的四种协议：本地协议、HTTP协议、SSH协议、Git协议。

### 4.1.1本地协议：共享文件系统

git clone /srv/git/project.git

Git会尝试使用硬链接或者直接复制所需要的文件

git clone <file:///srv/git/project.git>

Git会触发用于网路传输资料的进程，传输效率更低，但是可以获得一个没有外部参考或对象的干净版本库副本。缺点：共享文件系统难配置，多地办公不方便、访问速度也不快、且不利于权限管理。

### 4.1.2 HTTP协议：智能HTTP协议、哑HTTP协议

### 4.1.3 SSH协议

架设相对简单、使用普遍、访问安全、传输高效，但不支持匿名访问

git clone ssh://[user@]server/project.git

如果不指定用户名，则默认使用当前登录的名字。

git clone [user@]server:project.git

### 4.1.4 Git协议

传输速度最快，不需要进行用户授权，但是Git协议最难架设，需要开放9418端口，通常企业不会开放或者要求封锁。

## 4.2 搭建Git服务器

### 4.2.1 简单的Git服务器，可供几个人使用

1、在现有仓库上导出裸仓库：git clone --bare my\_project my\_project.git 相当于：

cp -rf my\_project/.git my\_project.git

2、通过scp命令将裸仓库放到服务器某个目录下，如：/srv/git/my\_project.git。

如果一个用户ssh登录到该服务器并拥有/srv/git/my\_project.git的可写权限，那么他将自动获得推送权限。建立SSH连接，只要每个用户可以获得主机的shell访问权限，任何SSH授权机制都是有效的。

git init --bare --shared 修改该仓库目录的组权限为可写

### 4.2.2 小型安装，实现了用户管理

1、为每个人创建账户

2、建立公用的git账户，将需要写权限的人发送一个ssh公钥，加入服务器git账户的~/.ssh/authorized\_keys文件中

3、通过LDAP服务，或者其他已经设定好的集中授权机制。

### 4.2.3 配置Git服务器

1、将用户的ssh公钥加入~/.ssh/authorized\_keys文件中

2、将git-shell设置为用户git的登录shell，cat /etc/shells;which git-shell;sudo -e /etc/shells

3、自定义git-shell

# 5 分布式Git

## 5.1 常见Git工作流

集中式工作流（2层：shared repository、developer）

集成管理者工作流（3层：blessed repository、integration manager、developer，三步：developer private -> developer public -> integration manager -> blessed repository）

主管与副主管工作流（4层：blessed repository、lieutenant、dictator、developer，四步：developer private -> developer public -> lieutenant -> dictator -> blessed repository）

## 5.2 常见项目贡献方式

私有小型团队：贡献者在推代码之前抓取远程分支的更新内容，并合并到本地分支，然后再推代码。

私有管理团队：贡献者把更新得代码推到远程仓库的其他分支上，维护者将远程仓库中的其他分支代码合并到master分支。

通过邮件的公开项目：将提交转换成电子邮件，发送给项目维护者。

## 5.3 维护项目

git diff生成的补丁

git apply patch\_name.patch 应用patch

git apply --check patch\_name.patch 检查patch是否可以顺利应用

git format-patch -M <remote>/<branch> 生成补丁

git am patch\_name.patch 应用patch

-3 : 使Git尝试进行三方合并，要求补丁是基于公共提交的

-i : 交互模式下，通常用于应用多个补丁的mbox文件

抓取分支并合并：

git remote add <remote> <remote\_url>

git fetch <remote>

git checkout -b <local\_branch> <remote>/<remote\_branch>

一次性抓取：适用于非持续性的合作

git pull <remote\_url>

查看相对于master分支的引用：

git log <branch> --not master

-p : 在提交后面显示差异（git diff）

## 5.4 整合提交

合并工作流：依次合并主题分支到master分支，项目更大时，将主题分支先合并到develop分支，当打标签发布的时候，将master分支快进到已经稳定的develop分支。

大项目合并工作流：包含四个长期分支：master、next、用于新工作的pu分支和用于维护性向后移植工作的maint分支

变基和拣选工作流：为了保持线性的提交历史。rerere（reuse recorded resolution）是一种简化冲突解决的方法，它将已解决的冲突进行缓存，之后遇到类似的场景直接应用。

## 5.5 打标签并发布

### 5.5.1 签名问题

gpg --list-keys 列出所有key

gpg -a --export <key> | git hash-object -w --stdin 导出key并将key导入到Git的数据库中

git tag -a <tag\_name> <key\_hash\_value> 创建tag直接指向这个key

git push --tags 将tag推到远程库上

git show <tag\_name> | gpg --import 从Git数据库中拉取key的blob对象并将key导入到GPG中

### 5.5.2 生成一个构建号

git describe <branch> 需要带注解的标签，轻量标签在命令后加--tags选项，由于是简短的SHA-1值，不一定有效 v1.6.2-rc1-20-g8c5b85c

### 5.5.3 准备一次发布

git archive <branch> --prefix='project/' | gzip > $(git describe <branch>).tar.gz 建立最新的快照归档

git archive <branch> --prefix='project/' --format=zip > $(git describe <branch>).tar.gz zip格式的包

### 5.5.4 制作提交简报

git shortlog --no-merges <branch> --not <tag> 生成一份自<tag>以来所有提交

# 6 Git托管平台

## 6.1 以拉取为中心的特殊合作流程

1、Fork一个项目

2、从master分支创建新分支

3、提交修改

4、推送分支到GitHub的派生项目

5、创建一个拉取请求

6、讨论，根据实际情况继续修改

7、项目的拥有者合并或关闭拉取请求

1. 将更新后的master分支同步到派生项目中

# 7 Git工具

## 7.1 选择修订版本

### 7.1.1指定修订版本：

git log --abbrev-commit --pretty=oneline 输出结果显示剪短且唯一的SHA-1指

当topic1分支指向提交ca82a6d...时，commit引用和分支引用等价，例如：

git show ca82a6dff817ec66f44342007202690a93763949

git show topic1

### 7.1.2引用日志

HEAD和分支引用所指向的历史，只存在于本地仓库

git reflog 查看引用日志，HEAD和分支引用所指向的历史

git show HEAD@[5] 输出最近5条reflog的历史记录

git log -g master 查看类似于git log 输出格式的引用日志信息

### 7.1.3祖先提交

父提交：在引用的尾部加上一个脱字符”^“或者“~”

git show HEAD^ 显示HEAD的父提交，等价于

git show HEAD~

区别在于：

^2代表第二父提交，只适用于合并提交

~2代表第一父提交的第一父提交，也就是祖父提交

两者可以同时使用

### 7.1.4提交区间

双点：选出在一个引用中而不在另一个引用中的提交，也就是第二个引用减去两个引用的交集

git log <branchA>..<branchB> 在branchB中而不在branchA中的提交

git log <remote>/<branch>..HEAD 显示当前分支中即将push到远程分支的本地提交

git log <remote>/<branch>.. 效果同上，Git默认用HEAD代替留空的一边

git log refA..refB

git log ^refA refB

git log refB --not refA

等价：

git log refA refB ^refC

git log refA refB --not refC

三点：两个引用包含的引用并集减去交集

git log --left-right refA...refB 显示两个引用的非公共引用，并通过尖括号区分属于哪边的引用

## 7.2 贮藏和清理

git stash (push) 贮藏修改，工作区和暂存区对于已跟踪的文件修改将被贮藏推送到栈上

git stash list 查看贮藏的东西

git stash apply 应用最近的一次贮藏工作内容，文件被修改，但是暂存区内容不会被恢复

git stash apply stash@{2} 应用stash@{2}贮藏内容

git stash apply --index 重新恢复修改的文件，并恢复暂存区内容

git stash drop stash@{0} 移除stash@{0}贮藏，相当于：

git stash pop

git stash -u/--include-untracked 贮藏未跟踪的文件

git stash -a/--all 贮藏所有文件，包括忽略的文件

git stash --patch 不贮藏任何修改过的东西，但交互式的提示那些需要保存在工作目录中

git stash branch <branch> 从贮藏创建一个分支，应用成功后移除该贮藏

git clean -f -d --dry-run/-n 演习强制移除未被跟踪文件和空目录，并显示将要移除什么

git stash --all/-a 将所有文件贮藏，达到清理工作空间的目的

git clean -x 被忽略的文件也将被清理

git clean -ff 清理仓库下的子仓库

## 7.3 签署工作

### 7.3.1配置签名

gpg --gen-key 生成个人秘钥

gpg --list-keys 列出秘钥

git config --global user.signingkey 0A46826A

### 7.3.2应用签名

git tag -s v1.5 -m "my signed 1.5 tag" 给标签签名

git tag -v v1.4.2.1 验证标签

git commit -a -S -m "signed commit" 给提交签名

git log --show-signature -1 查看及验证提交的签名

git log --pretty=format:"%h %G? %aN %s" 以%G?格式列在输出中

git merge/pull --verify-signatures -S <branch> 检查并拒绝没有携带可信GPG签名的提交，并为merge命令生成的合并提交签名

## 7.4 搜索

git grep -n/--line-number <str> 显示匹配行的行号

--count 显示每个文件的匹配数

-p/--show-function 显示每一个匹配的字符串所在的方法或函数

git log -S <str> --oneline 查看引入和删除该字符串的提交

-G <match> 使用正则表达式搜索

-L :<function\_name>:<file> 显示该函数从创建后所有变更对应的补丁

怎么使用正则表达式搜索？

## 7.5 重写历史

必须保证本地分支没有推送到远程分支，因为所有这些操作都将生成新SHA-1校验和。

git commit --amend --no-edit 对最后一次提交进行追加提交，不需要重新进行提交说明，但是会生成新SHA-1校验和

git rebase -i HEAD~3 交互式变基，显示的提交历史从旧到新

修改提交信息：将提交前面的pick改为edit，保存并退出后，一次对每个edit提交进行重新编辑

重新排序提交：直接删除一行的提交，保存并退出后，将移除那一次的提交

压缩提交：修改pick为squash，会将那一次提交合并到他的上一次非合并提交

拆分提交：修改pick为edit，然后git rebase HEAD^，接着重新提交多次，即可实现拆分提交

git filter-branch git-filter-repo更推荐

git filter-branch --tree-filter 'rm -rf passwords.txt' HEAD 从每个提交中删除这个密码文件

git filter-branch --subdirectory-filter trunk HEAD 使trunk目录作为新的根目录

git filter-branch --commit-filter '<commands>' HEAD 全局修改提交信息

## 7.6 重置

### 7.6.1 Git的三棵树：HEAD、Index、Working Directory

**HEAD**：当前分支引用的指针

git cat-file -p HEAD 查看某个引用的具体信息

git ls-tree -r HEAD

**Index**：索引是预期的下一次提交，是扁平的清单实现的，但是我们可以把它理解为树

git ls-files 显示出索引（暂存区）当前的样子

**Working Directory**：另外两棵树以一种高效但并不直观的方式，将它们的内容存储在.git文件夹中， 工作目录将它们解包为实际的文件以便编辑

当检出一个分支时，首先会修改HEAD指向新的分支引用，然后将索引填充为该次提交的快照，最后将索引的内容复制到工作目录中

### 7.6.2 reset工作流程

1、移动HEAD指向的分支的指向，与checkout做的改变HEAD指向的分支不同

2、更新索引，reset会用HEAD指向的当前快照来更新索引

3、更新工作目录，将索引内容复制到工作目录中

git reset --soft HEAD~ 移动HEAD，但是不会改变索引和工作区，相当于撤销了一次commit操作，到第一步停止

git reset --mixed HEAD~ 移动HEAD并改变索引，相当于撤销了add和commit命令，到第二步停止

git reset --hard HEAD~ 撤销了add、commit命令和工作目录所有的工作

### 7.6.3 通过路径来重置

git reset <file> 跳过移动HEAD分支的指向，停在第二步，根据HEAD指向的快照还原该目录或文件，相当于

git reset --mixed HEAD <file>

git reset <commit\_id> -- <file> 指定路径根据某一个提交的快照内容对索引进行重置

## 7.7 高级合并

git merge -Xignore-space-change <branch> 忽略空白

git ls-files -u 列出冲突文件的所有副本的索引

git show :1:<file> > common\_<file>

git show :2:<file> > our\_<file>

git show :3:<file> > their\_<file>

git checkout --ours/--theirs <file> 检出冲突文件，直接保留一方而丢弃另一方

git log --cc -p -1

git revert -m 1 HEAD 还原提交

git merge -Xours/-Xtheirs 在冲突时，直接选择ours或者theirs，而不是标记冲突文件

git merge-file --ours/--theirs