git

是目前世界上最先进的分布式版本控制系统

补充说明

git命令 很多人都知道，Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。

Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！

你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git，一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

Git常用命令清单

语法

```
git [--version] [--help] [-C <path>] [-c name=value]
   [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
   [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
   [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
   <command> [<args>]
```

选项

```
add             将文件内容添加到索引
bisect          通过二进制查找引入错误的更改
```

branch	列出，创建或删除分支
checkout	检查分支或路径到工作树
clone	将存储库克隆到新目录中
commit	将更改记录到存储库
diff	显示提交，提交和工作树等之间的更改
fetch	从另一个存储库下载对象和引用
grep	打印匹配图案的行
init	创建一个空的Git仓库或重新初始化一个现有的
log	显示提交日志
merge	加入两个或更多的开发历史
mv	移动或重命名文件，目录或符号链接
pull	从另一个存储库或本地分支获取并合并
push	更新远程引用以及相关对象
rebase	转发端口本地提交到更新的上游头
reset	将当前HEAD复位到指定状态
rm	从工作树和索引中删除文件
show	显示各种类型的对象
status	显示工作树状态
tag	创建，列出，删除或验证使用GPG签名的标签对象

例子

init

```
git init #初始化
```

status

```
git status #获取状态
```

add

```
git add file # .或*代表全部添加
git rm --cached <added_file_to_undo> # 在commit之前撤销git add操作
git reset head # 好像比上面git rm --cached更方便
```

commit

```
git commit -m "message" #此处注意乱码
```

remote

```
git remote add origin git@github.com:JSLite/test.git #添加源
```

push

```
git push -u origin master # push同事设置默认跟踪分支
git push origin master
git push -f origin master # 强制推送文件，缩写 -f□全写--force□
```

clone

```
git clone git://github.com/JSLite/JSLite.js.git
```

`git clone git://github.com/JSLite/JSLite.js.git mypro` #克隆到自定义文件夹
`git clone [user@]example.com:path/to/repo.git/` #SSH协议还有另一种写法。

git clone支持多种协议，除了HTTP(s)以外，还支持SSH、Git、本地文件协议等，下面是一些例子。`git clone <版本库的网址> <本地目录名>`

```
$ git clone http[s]://example.com/path/to/repo.git/
$ git clone ssh://example.com/path/to/repo.git/
$ git clone git://example.com/path/to/repo.git/
$ git clone /opt/git/project.git
$ git clone file:///opt/git/project.git
$ git clone ftp[s]://example.com/path/to/repo.git/
$ git clone rsync://example.com/path/to/repo.git/
```

配置

首先是配置帐号信息 `ssh -T git@github.com` 测试。

修改项目中的个人信息

```
git help config # 获取帮助信息，查看修改个人信息的参数
git config --global user.name "小弟调调"     # 修改全局名字
git config --global user.email "wowohoo@qq.com"  # 修改全局邮箱
git config --list          # 查看配置的信息
```

配置自动换行

自动转换坑太大，提交到git是自动将换行符转换为lf

```
git config --global core.autocrlf input
```

常见使用场景

创建SSH密钥

这个密钥用来跟 github 通信，在本地终端里生成然后上传到 github

```
ssh-keygen -t rsa -C 'wowohoo@qq.com' # 生成密钥
ssh-keygen -t rsa -C "wowohoo@qq.com" -f ~/.ssh/ww_rsa # 指定生成目录文件名字
ssh -T git@github.com # 测试是否成功
```

多账号ssh配置

1.生成指定名字的密钥

`ssh-keygen -t rsa -C "邮箱地址" -f ~/.ssh/jslite_rsa`
会生成 `jslite_rsa` 和 `jslite_rsa.pub` 这两个文件

2.密钥复制到托管平台上

`vim ~/.ssh/jslite_rsa.pub`

打开公钥文件 jslite_rsa.pub，并把内容复制至代码托管平台上

3.修改config文件

vim ~/.ssh/config #修改config文件，如果没有创建 config

```
Host jslite.github.com
  HostName github.com
  User git
  IdentityFile ~/.ssh/jslite_rsa

Host work.github.com
  HostName github.com
  # Port 服务器open-ssh端口（默认：22,默认时一般不写此行）
  # PreferredAuthentications 配置登录时用什么权限认证
  #                          publickey|password publickey|keyboard-
interactive等
  User git
  IdentityFile ~/.ssh/work_rsa
```

- Host 这里是个别名可以随便命名
- HostName 一般是网站如：git@ss.github.com:username/repo.git 填写 github.com
- User 通常填写git
- IdentityFile 使用的公钥文件地址

4.测试

```
ssh -T git@jslite.github.com  # `@`后面跟上定义的Host
ssh -T work.github.com         # 通过别名测试
ssh -i ~/公钥文件地址 Host别名 #如 ssh -i ~/.ssh/work_rsa work.github.com
```

5.使用

```
# 原来的写法
git clone git@github.com:<jslite的用户名>/learngit.git
# 现在的写法
git clone git@jslite.github.com:<jslite的用户名>/learngit.git
git clone git@work.github.com:<work的用户名>/learngit.git
```

5.注意

如果你修改了id_rsa的名字，你需要将ssh key添加到SSH agent中，如：

```
ssh-add ~/.ssh/jslite_rsa
ssh-add -l  # 查看所有的key
ssh-add -D  # 删除所有的key
ssh-add -d  ~/.ssh/jslite_rsa # 删除指定的key
```

免密码登录远程服务器

```
$ ssh-keygen -t rsa -P '' -f ~/.ssh/aliyunserver.key
$ ssh-copy-id -i ~/.ssh/aliyunserver.key.pub root@192.168.182.112 # 这里需要
```

```
输入密码一次
```

编辑 ~/.ssh/config

```
Host aliyun1
  HostName 192.168.182.112
  User root
  PreferredAuthentications publickey
  IdentityFile ~/.ssh/aliyunserver.key
```

上面配置完了，可以通过命令登录，不需要输入IP地址和密码 ssh aliyun1

https协议下提交代码免密码

```
git clone https://github.com/username/rep.git
```

通过上面方式克隆可能需要密码，解决办法：进入当前克隆的项目 vi rep/.git/config 编辑 config，按照下面方式修改，你就可以提交代码不用输入密码了。

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
-   url = https://github.com/username/rep.git
+   url = https://用户名:密码@github.com/username/rep.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

文件推向3个git库

1. 增加3个远程库地址

```
git remote add origin https://github.com/JSLite/JSLite.git
git remote set-url --add origin https://gitlab.com/wang/JSLite.js.git
git remote set-url --add origin https://oschina.net/wang/JSLite.js.git
```

2. 删除其中一个 set-url 地址

```
usage: git remote set-url [--push] <name> <newurl> [<oldurl>]
   or: git remote set-url --add <name> <newurl>
   or: git remote set-url --delete <name> <url>
```

```
git remote set-url --delete origin https://oschina.net/wang/JSLite.js.git
```

3.推送代码

```
git push origin master
git push -f origin master  # 强制推送
```

4.拉代码

只能拉取 `origin` 里的一个url地址，这个fetch-url
默认为你添加的到 `origin`的第一个地址

```
git pull origin master
git pull --all # 获取远程所有内容包括tag
git pull origin next:master # 取回origin主机的next分支，与本地的master分支合并
git pull origin next # 远程分支是与当前分支合并

# 上面一条命令等同于下面两条命令
git fetch origin
git merge origin/next
```

如果远程主机删除了某个分支，默认情况下git pull 不会在拉取远程分支的时候，删除对应的本地分支。
这是为了防止，由于其他人操作了远程主机，导致git pull不知不觉删除了本地分支。
但是，你可以改变这个行为，加上参数 -p 就会在本地删除远程已经删除的分支。

```
$ git pull -p
# 等同于下面的命令
$ git fetch --prune origin
$ git fetch -p
```

5.更改pull

只需要更改config文件里，那三个url的顺序即可fetch-url会直接对应排行第一的那个utl连接。

修改远程仓库地址

```
git remote remove origin  # 删除该远程路径
git remote add origin git@jslite.github.com:JSLite/JSLite.git  # 添加远程路径
```

撤销远程记录

```
git reset --hard HEAD~1 # 撤销一条记录
git push -f origin HEAD:master # 同步到远程仓库
```

放弃本地的文件修改

```
git reset --hard FETCH_HEAD # FETCH_HEAD表示上一次成功git pull之后形成的commit点。
然后git pull
```

`git reset --hard FETCH_HEAD` 出现错误

```
git pull
You are not currently on a branch, so I cannot use any
'branch.<branchname>.merge' in your configuration file.
Please specify which remote branch you want to use on the command
line and try again (e.g. 'git pull <repository> <refspec>').
```

```
See git-pull(1) FOR details.
```

解决方法：

```
git checkout -b temp # 新建+切换到temp分支
git checkout master
```

最简单放弃本地修改内容

```
# 如果有的修改以及加入暂存区的话
git reset --hard
# 还原所有修改，不会删除新增的文件
git checkout .
# 下面命令会删除新增的文件
git clean -xdf
```

通过存储暂存区stash在删除暂存区的方法放弃本地修改。

```
git stash && git stash drop
```

回滚到某个commit提交

```
git revert HEAD~1 # 撤销一条记录 会弹出 commit 编辑
git push # 提交回滚
```

回退到某一个版本

```
git reset --hard <hash>
# 例如 git reset --hard a3hd73r
# --hard代表丢弃工作区的修改，让工作区与版本代码一模一样，与之对应，
# --soft参数代表保留工作区的修改。
```

去掉某个commit

```
# 实质是新建了一个与原来完全相反的commit抵消了原来commit的效果
git revert <commit-hash>
```

新建一个空分支

```
# 这种方式新建的分支(gh-pages)是没有 commit 记录的
git checkout --orphan gh-pages
# 删除新建的gh-pages分支原本的内容，如果不删除，提交将作为当前分支的第一个commit
git rm -rf .
# 查看一下状态 有可能上面一条命令，没有删除还没有提交的的文件
git state
```

合并多个commit

```
# 这个命令，将最近4个commit合并为1个HEAD代表当前版本。
# 将进入VIM界面，你可以修改提交信息。
git rebase -i HEAD~4
```

```
# 可以看到其中分为两个部分，上方未注释的部分是填写要执行的指令，
# 而下方注释的部分则是指令的提示说明。指令部分中由前方的命令名称commit hash 和 commit message 组成
# 当前我们只要知道 pick 和 squash 这两个命令即可。
# --> pick 的意思是要会执行这个 commit
# --> squash 的意思是这个 commit 会被合并到前一个commit

# 我们将 需要保留的 这个 commit 前方的命令改成 squash 或 s然后输入:wq以保存并退出
# 这是我们会看到 commit message 的编辑界面

# 其中，非注释部分就是两次的 commit message，你要做的就是将这两个修改成新的 commit message
#
# 输入wq保存并推出，再次输入git log查看 commit 历史信息，你会发现这两个 commit 已经合并了。
# 将修改强制推送到前端
git push -f origin master
```

修改远程Commit记录

```
git commit --amend
# amend只能修改没有提交到线上的，最后一次commit记录
git rebase -i HEAD~3
# 表示要修改当前版本的倒数第三次状态
# 将要更改的记录行首单词 pick 改为 edit
pick 96dc3f9 doc: Update quick-start.md
pick f1cce8a test(Transition):Add transition test (#47)
pick 6293516 feat(Divider): Add Divider component.
# Rebase eeb03a4..6293516 onto eeb03a4 (3 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
```

保存并退出，会弹出下面提示

```
# You can amend the commit now, with
#
#   git commit --amend
#
# Once you are satisfied with your changes, run
#
#   git rebase --continue

# 通过这条命令进入编辑页面更改commit保存退出
git commit --amend
```

```
# 保存退出确认修改，继续执行 rebase，
git rebase --continue
# 如果修改多条记录反复执行上面两条命令直到完成所有修改

# 最后，确保别人没有提交进行push，最好不要加 -f 强制推送
git push -f origin master
```

添加忽略文件

```
echo node_modules/ >> .gitignore
```

利用commit关闭一个issue

这个功能在Github上可以玩儿，Gitlab上特别老的版本不能玩儿哦，那么如何跟随着commit关闭一个issue呢? 在confirm merge的时候可以使用一下命令来关闭相关issue:

`fixes #xxx, fixed #xxx, fix #xxx, closes #xxx, close #xxx, closed #xxx,`

同步fork的上游仓库

Github教程同步fork教程，在Github上同步一个分支(fork)

设置添加多个远程仓库地址。

在同步之前，需要创建一个远程点指向上游仓库(repo).如果你已经派生了一个原始仓库，可以按照如下方法做。

```
$ git remote -v
# List the current remotes (列出当前远程仓库)
# origin  https://github.com/user/repo.git (fetch)
# origin  https://github.com/user/repo.git (push)
$ git remote add upstream https://github.com/otheruser/repo.git
# Set a new remote (设置一个新的远程仓库)
$ git remote -v
# Verify new remote (验证新的原唱仓库)
# origin    https://github.com/user/repo.git (fetch)
# origin    https://github.com/user/repo.git (push)
# upstream  https://github.com/otheruser/repo.git (fetch)
# upstream  https://github.com/otheruser/repo.git (push)
```

同步更新仓库内容

同步上游仓库到你的仓库需要执行两步：首先你需要从远程拉去，之后你需要合并你希望的分支到你的本地副本分支。从上游的存储库中提取分支以及各自的提交内容。 `master` 将被存储在本地分支机构 `upstream/master`

```
git fetch upstream
# remote: Counting objects: 75, done.
# remote: Compressing objects: 100% (53/53), done.
# remote: Total 62 (delta 27), reused 44 (delta 9)
# Unpacking objects: 100% (62/62), done.
# From https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY
```

```
#  * [new branch]      master     -> upstream/master
```

检查你的 fork's 本地 `master` 分支

```
git checkout master
# Switched to branch 'master'
```

合并来自 `upstream/master` 的更改到本地 master 分支上。 这使你的前 fork's `master` 分支与上游资源库同步，而不会丢失你本地修改。

```
git merge upstream/master
# Updating a422352..5fdff0f
# Fast-forward
#  README                    |    9 -------
#  README.md                 |    7 ++++++
#  2 files changed, 7 insertions(+), 9 deletions(-)
#  delete mode 100644 README
#  create mode 100644 README.md
```

批量修改历史commit中的名字和邮箱

1.克隆仓库

注意参数，这个不是普通的clone，clone下来的仓库并不能参与开发

```
git clone --bare https://github.com/user/repo.git
cd repo.git
```

2.命令行中运行代码

OLD_EMAIL原来的邮箱
CORRECT_NAME更正的名字
CORRECT_EMAIL更正的邮箱

将下面代码复制放到命令行中执行

```
git filter-branch -f --env-filter '
OLD_EMAIL="wowohoo@qq.com"
CORRECT_NAME="小弟调调"
CORRECT_EMAIL="更正的邮箱@qq.com"
if [ "$GIT_COMMITTER_EMAIL" = "$OLD_EMAIL" ]
then
    export GIT_COMMITTER_NAME="$CORRECT_NAME"
    export GIT_COMMITTER_EMAIL="$CORRECT_EMAIL"
fi
if [ "$GIT_AUTHOR_EMAIL" = "$OLD_EMAIL" ]
then
    export GIT_AUTHOR_NAME="$CORRECT_NAME"
    export GIT_AUTHOR_EMAIL="$CORRECT_EMAIL"
fi
' --tag-name-filter cat -- --branches --tags
```

执行过程

```
Rewrite 160d4df2689ff6df3820563bfd13b5f1fb9ba832 (479/508) (16 seconds
passed, remaining 0 predicted)
Ref 'refs/heads/dev' was rewritten
Ref 'refs/heads/master' was rewritten
```

3.同步到远程仓库

同步到push远程git仓库

```
git push --force --tags origin 'refs/heads/*'
```

我还遇到了如下面错误lab默认给master分支加了保护，不允许强制覆盖。Project(项目)->Setting->Repository 菜单下面的Protected branches把master的保护去掉就可以了。修改完之后，建议把master的保护再加回来，毕竟强推不是件好事。

```
remote: GitLab: You are not allowed to force push code to a protected branch
on this project.
```

当上面的push 不上去的时候，先 git pull 确保最新代码

```
git pull  --allow-unrelated-histories
# 或者指定分枝
git pull origin master --allow-unrelated-histories
```

查看某个文件历史

```
git log --pretty=oneline 文件名 # 列出文件的所有改动历史
git show c178bf49   # 某次的改动的修改记录
git log -p c178bf49 # 某次的改动的修改记录
git blame 文件名  # 显示文件的每一行是在那个版本最后修改。
git whatchanged 文件名 # 显示某个文件的每个版本提交信息：提交日期，提交人员，版本号，提交备注（没有修改细节）
```

打造自己的git命令

```
git config --global alias.st status
git config --global alias.br branch
git config --global alias.co checkout
git config --global alias.ci commit
```

配置好后再输入git命令的时候就不用再输入一大段了，例如我们要查看状态，只需：

```
git st
```

中文乱码的解决方案

```
git config --global core.quotepath false
```

新建仓库

init

`git init` #初始化

status

`git status` #获取状态

add

```
git add file # .或*代表全部添加
git rm --cached <added_file_to_undo> # 在commit之前撤销git add操作
git reset head # 好像比上面git rm --cached更方便
```

commit

`git commit -m "message"` #此处注意乱码

remote

`git remote add origin git@github.com:JSLite/test.git` #添加源

push

```
git push -u origin master # push同事设置默认跟踪分支
git push origin master
git push -f origin master # 强制推送文件，缩写 -f□全写--force□
```

clone

```
git clone git://github.com/JSLite/JSLite.js.git
git clone git://github.com/JSLite/JSLite.js.git mypro #克隆到自定义文件夹
git clone [user@]example.com:path/to/repo.git/ #SSH协议还有另一种写法。
```

git clone支持多种协议，除了HTTP(s)以外，还支持SSH□Git□本地文件协议等，下面是一些例子。`git clone <版本库的网址> <本地目录名>`

```
$ git clone http[s]://example.com/path/to/repo.git/
$ git clone ssh://example.com/path/to/repo.git/
$ git clone git://example.com/path/to/repo.git/
$ git clone /opt/git/project.git
$ git clone file:///opt/git/project.git
$ git clone ftp[s]://example.com/path/to/repo.git/
$ git clone rsync://example.com/path/to/repo.git/
```

本地

help

```
git help config # 获取帮助信息
```

add

```
git add *   # 跟踪新文件
git add -u [path]   # 添加[指定路径下]已跟踪文件
```

rm

```
rm *&git rm *          # 移除文件
git rm -f *            # 移除文件
git rm --cached *      # 取消跟踪
git mv file_from file_to  # 重命名跟踪文件
git log   # 查看提交记录
```

commit

```
git commit #提交更新
git commit -m 'message' #提交说明
git commit -a #跳过使用暂存区域，把所有已经跟踪过的文件暂存起来一并提交
git commit --amend #修改最后一次提交
git commit log #查看所有提交，包括没有push的commit
git commit -m "#133" #关联issue 任意位置带上# 符号加上issue号码
git commit -m "fix #133" commit关闭issue
git commit -m '概要描述'$'\n\n''1.详细描述'$'\n''2.详细描述' #提交简要描述和详细描述
```

reset

```
git reset HEAD *  # 取消已经暂存的文件
git reset --mixed HEAD * # 同上
git reset --soft HEAD *  # 重置到指定状态，不会修改索引区和工作树
git reset --hard HEAD *  # 重置到指定状态，会修改索引区和工作树
git reset -- files *     # 重置index区文件
```

revert

```
git revert HEAD   # 撤销前一次操作
git revert HEAD~  # 撤销前前一次操作
git revert commit # 撤销指定操作
```

checkout

```
git checkout -- file  # 取消对文件的修改（从暂存区——覆盖worktree file
git checkout branch|tag|commit -- file_name  # 从仓库取出file覆盖当前分支
git checkout HEAD~1 [文件] # 将会更新 working directory 去匹配某次 commit
git checkout -- .          # 从暂存区取出文件覆盖工作区
git checkout -b gh-pages  0c304c9  # 这个表示 从当前分支 commit 哈希值为 0c304c9 的节点，分一个新的分支gh-pages出来，并切换到 gh-pages
```

diff

```
git diff file     # 查看指定文件的差异
git diff --stat   # 查看简单的diff结果
git diff  # 比较Worktree和Index之间的差异
git diff --cached   # 比较Index和HEAD之间的差异
git diff HEAD       # 比较Worktree和HEAD之间的差异
git diff branch     # 比较Worktree和branch之间的差异
git diff branch1 branch2  # 比较两次分支之间的差异
git diff commit commit    # 比较两次提交之间的差异
git diff master..test   # 上面这条命令只显示两个分支间的差异
git diff master...test    # 你想找出‘master’,‘test’的共有 父分支和'test'分支之间的差异，你用3个‘ .'来取代前面的两个'.'
```

stash

```
git stash # 将工作区现场（已跟踪文件）储藏起来，等以后恢复后继续工作。
git stash list  # 查看保存的工作现场
git stash apply # 恢复工作现场
git stash drop  # 删除stash内容
git stash pop   # 恢复的同时直接删除stash内容
git stash apply stash@{0} # 恢复指定的工作现场，当你保存了不只一份工作现场时。
```

merge

```
git merge --squash test # 合并压缩，将test上的commit压缩为一条
```

cherry-pick

```
git cherry-pick commit    # 拣选合并，将commit合并到当前分支
git cherry-pick -n commit # 拣选多个提交，合并完后可以继续拣选下一个提交
```

rebase

```
git rebase master   # 将master分之上超前的提交，变基到当前分支
git rebase --onto master 169a6  # 限制回滚范围□rebase当前分支从169a6以后的提交
git rebase --interactive # 交互模式，修改commit
git rebase --continue    # 处理完冲突继续合并
git rebase --skip        # 跳过
git rebase --abort       # 取消合并
```

分支branch

删除

```
git push origin :branchName  # 删除远程分支
git push origin --delete new # 删除远程分支new
git branch -d branchName     # 删除本地分支，强制删除用-D
git branch -d test      # 删除本地test分支
git branch -D test      # 强制删除本地test分支
```

```
git remote prune origin # 远程删除了，本地还能看到远程存在，这条命令删除远程不存在的分支
```

提交

```
git push -u origin branchName # 提交分支到远程origin主机中
```

拉取

```
git fetch -p #拉取远程分支时，自动清理 远程分支已删除，本地还存在的对应同名分支。
```

分支合并

```
git merge branchName      # 合并分支 - 将分支branchName和当前所在分支合并
git merge origin/master   # 在本地分支上合并远程分支。
git rebase origin/master  # 在本地分支上合并远程分支。
git merge test            # 将test分支合并到当前分支
```

重命名

```
git branch -m old new #重命名分支
```

查看

```
git branch      # 列出本地分支
git branch -r   # 列出远端分支
git branch -a   # 列出所有分支
git branch -v   # 查看各个分支最后一个提交对象的信息
git branch --merge      # 查看已经合并到当前分支的分支
git branch --no-merge   # 查看为合并到当前分支的分支
git remote show origin  # 可以查看remote地址，远程分支
```

新建

```
git branch test # 新建test分支
git branch newBrach 3defc69 # 指定哈希3defc69□新建分支名字为newBrach
git checkout -b newBrach origin/master # 取回远程主机的更新以后，在它的基础上创建一个新的分支
git checkout -b newBrach 3defc69 # 以哈希值3defc69□新建 newBrach 分支，并切换到该分支
```

连接

```
git branch --set-upstream dev origin/dev     # 将本地dev分支与远程dev分支之间建立链接
git branch --set-upstream master origin/next # 手动建立追踪关系
```

分支切换

```
git checkout test     # 切换到test分支
git checkout -b test  # 新建+切换到test分支
git checkout -b test dev # 基于dev新建test分支，并切换
```

远端

```
git fetch <远程主机名> <分支名>   # fetch取回所有分支branch的更新
git fetch origin remotebranch[:localbranch]   #  从远端拉去分支[到本地指定分支]
git merge origin/branch   # 合并远端上指定分支
git pull origin remotebranch:localbranch  #  拉去远端分支到本地分支
git push origin branch    # 将当前分支，推送到远端上指定分支
git push origin localbranch:remotebranch  # 推送本地指定分支，到远端上指定分支
git push origin :remotebranch   # 删除远端指定分支
git checkout -b [--track] test origin/dev # 基于远端dev分支，新建本地test分支[同时设置跟踪]
```

submodule

克隆项目同时克隆submodule

```
git clone https://github.com/jaywcjlove/handbook.git --depth=1 --recurse-submodules
```

克隆项目，之后再手动克隆 submodule 子项目

```
git submodule add --force '仓库地址''路径'
# 其中，仓库地址是指子模块仓库地址，路径指将子模块放置在当前工程下的路径。
# 注意：路径不能以 / 结尾（会造成修改不生效）、不能是现有工程已有的目录（不能顺利 Clone
git submodule init # 初始化submodule
git submodule update # 更新submodule(必须在根目录执行命令)
git submodule update --init --recursive  # 下载的工程带有submodule
```

当使用git clone下来的工程中带有submodule时，初始的时候submodule的内容并不会自动下载下来的，此时，只需执行如下命令：

```
git submodule foreach git pull  # submodule 里有其他的 submodule 一次更新
git submodule foreach git pull origin master # submodule更新

git submodule foreach --recursive git submodule init
git submodule foreach --recursive git submodule update
```

删除文件

```
git rm -rf node_modules/
```

remote

git是一个分布式代码管理工具，所以可以支持多个仓库，在git里，服务器上的仓库在本地称之为remote个人开发时，多源用的可能不多，但多源其实非常有用。

```
git remote add origin1 git@github.com:yanhaijing/data.js.git
git remote    # 显示全部源
git remote -v # 显示全部源+详细信息
```

```
git remote rename origin1 origin2 # 重命名
git remote rm origin    # 删除
git remote show origin  # 查看指定源的全部信息
```

标签tag

当开发到一定阶段时，给程序打标签是非常棒的功能。

```
git tag -a v0.1 -m 'my version 1.4' # 新建带注释标签
git push origin --tags              # 一次性推送所有分支
git push origin v1.5                # 推送单个tag到orgin源上
git tag -v v1.4.2.1                 # 验证标签，验证已经签署的标签
git show v1.5                       # 看到对应的 GPG 签

git tag        # 列出现有标签
git tag v0gi.1 # 新建标签
git checkout tagname   # 切换到标签
git tag -d v0.1 # 删除标签
git push origin :refs/tags/v0.1 # 删除远程标签
git pull --all # 获取远程所有内容包括tag
git --git-dir='<绝对地址>/.git' describe --tags HEAD # 查看本地版本信息
```

日志log

```
git config format.pretty oneline  #显示历史记录时，每个提交的信息只显示一行
git config color.ui true #彩色的 git 输出
git log #查看最近的提交日志
git log --pretty=oneline #单行显示提交日志
git log --graph --pretty=oneline --abbrev-commit
git log -num #显示第几条log（倒数）
git reflog #查看所有分支的所有操作记录
git log --since=1.day #一天内的提交；你可以给出各种时间格式，比如说具体的某一天（“2008-01-15”），或者是多久以前（“2 years 1 day 3 minutes ago”）
git log --pretty="%h - %s" --author=自己的名字 #查看自己的日志
git log -p -2 #展开两次更新显示每次提交的内容差异
git log --stat #要快速浏览其他协作者提交的更新都作了哪些改动
git log --pretty=format:"%h - %an, %ar : %s"#定制要显示的记录格式
git log --pretty=format:'%h : %s' --date-order --graph # 拓扑顺序展示
git log --pretty=format:'%h : %s - %ad' --date=short #日期YYYY-MM-DD显示
git log <last tag> HEAD --pretty=format:%s # 只显示commit
git config --global format.pretty '%h : %s - %ad' --date=short #日期YYYY-MM-DD显示 写入全局配置
```

%H	提交对象（commit）的完整哈希字串	%ad	作者修订日期（可以用 -date= 选项定制格式）
%h	提交对象的简短哈希字串	%ar	作者修订日期，按多久以前的方式显示
%T	树对象（tree）的完整哈希字串	%cn	提交者(committer)的名字
%t	树对象的简短哈希字串	%ce	提交者的电子邮件地址
%P	父对象（parent）的完整哈希字串	%cd	提交日期
%p	父对象的简短哈希字串	%cr	提交日期，按多久以前的方式显示
%an	作者（author）的名字	%s	提交说明

%ae 作者的电子邮件地址	-	-

Pretty Formats

重写历史

```
git commit --amend    # 改变最近一次提交
git rebase -i HEAD~3  # 修改最近三次的提交说明，或者其中任意一次
git commit --amend    # 保存好了，这些指示很明确地告诉了你该干什么
git rebase --continue # 修改提交说明，退出编辑器。
```

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

改成

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

删除仓库

```
cd ..
rm -rf repo.git
```

Github官方教程

其它

```
git help *  # 获取命令的帮助信息
git status  # 获取当前的状态，非常有用，因为git会提示接下来的能做的操作
```

报错问题解决

1. git fatal: protocol error: bad line length character: No s

解决办法：更换remote地址为 http/https 的

2. The requested URL returned error: 403 Forbidden while accessing

解决github push错误的办法：

```
#vim 编辑器打开 当前项目中的config文件
vim .git/config

#修改
[remote "origin"]
    url = https://github.com/jaywcjlove/example.git

#为下面代码
[remote "origin"]
```

```
    url = https://jaywcjlove@github.com/jaywcjlove/example.git
```

3. git status 显示中文问题

在查看状态的时候 git status 如果是中文就显示下面的情况

```
\344\272\247\345\223\201\351\234\200\346\261\202
```

解决这个问题方法是：

```
git config --global core.quotepath false
```

参考资料

- Git官网
- https://try.github.io
- Git参考手册
- Git简明手册
- Git Magic
- Git Community Book 中文版
- Pro Git
- 图解Git
- git-简明指南
- learnGitBranching 在线学习工具
- 初级教程
- 廖雪峰的Git教程
- 蒋鑫老师将带你入github的大门
- git详解
- oschina教程
- How to undo (almost) anything with Git撤销一切，汇总各种回滚撤销的场景，加强学习。
- Git 教程 | 菜鸟教程runoob.com
- Git 本地仓库和裸仓库
- 沉浸式学 Git
- Git进阶用法，主要是rebase高级用法