GitHub 学习日志

Draft by Chao Li on 6/21/2014

# Summary：

学习这个可以很大的节省商用的成本，想创建private的仓库只需每月交7美金左右，还是挺划算的，毕竟可以为我们节省很多的麻烦，提高我们的效率。这个学习日志的主要目的就是让我能够精通github的使用，为自己项目的建立创造条件。

Github相比于SVN（subversion）， CVS和clearcase其他版本管理工具来说最重要和最主要的区别是GitHub是支持本地和远程仓库，支持离线commit。

# Reference:

<https://github.com/webcoding/useGit/blob/master/README.md>

<http://blog.csdn.net/ithomer/article/details/7529022>

<https://git-scm.com/docs>

# Git第一次配置：

git config --global user.name "Li,Chao"

git config --global user.email "chao.li@honeywell.com"

# Github安装：

直接google输入就可以找到Github的window安装、mac安装和linux安装的版本。

Window版本安装地址

<https://windows.github.com>

mac版本安装地址

www.**macwindows**.com/index.html

# Github的基本命令的使用：

usage: git [--version] [--help] [-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>]

The most commonly used git commands are:

add Add file contents to the index

bisect Find by binary search the change that introduced a bug

branch List, create, or delete branches

checkout Checkout a branch or paths to the working tree

clone Clone a repository into a new directory

commit Record changes to the repository

diff Show changes between commits, commit and working tree, etc

fetch Download objects and refs from another repository

grep Print lines matching a pattern

init Create an empty Git repository or reinitialize an existing one

log Show commit logs

merge Join two or more development histories together

mv Move or rename a file, a directory, or a symlink

pull Fetch from and integrate with another repository or a local branch

push Update remote refs along with associated objects

rebase Forward-port local commits to the updated upstream head

reset Reset current HEAD to the specified state

rm Remove files from the working tree and from the index

show Show various types of objects

status Show the working tree status

tag Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some

concept guides. See 'git help <command>' or 'git help <concept>'

to read about a specific subcommand or concept.

1. Git--help(两个中划线)

显示基本的一些命令，这个命令也会有一些变种

Git help <command>

1. git clone

这是一种较为简单的初始化方式，当你已经有一个远程的Git版本库，只需要在本地克隆一份

git  clone  git://github.com/someone/some\_project.git   some\_project

上面的命令就是将'git://github.com/someone/some\_project.git'这个URL地址的远程版本库，完全克隆到本地some\_project目录下

举例：

Git clone [http://github.com/zju-chaoli/Test.git Test#2](http://github.com/zju-chaoli/Test.git%20Test#2)

Clone 远程Test.git项目到本地并且创建Test#2文件夹（本地depository）

1. 如何删除本地库

rm –rf Test#2

1. git init 和 **git remote**

这种方式稍微复杂一些，当你本地创建了一个工作目录，你可以进入这个目录，使用'git init'命令进行初始化；Git以后就会对该目录下的文件进行版本控制，这时候如果你需要将它放到远程服务器上，可以在远程服务器上创建一个目录，并把可访问的URL记录下来，此时你git就可以利用'git remote add'命令来增加一个远程服务器端

git  remote  add  origin  git://github.com/someone/another\_project.git

上面的命令就会增加URL地址为'git: //github.com/someone/another\_project.git'，名称为origin的远程服务器，以后提交代码的时候只需要使用 origin

git remote add origin https://github.com/zju-chaoli/Test.git

git push -u origin master

或者直接

git remote push <http://github.com/zju-chaoli/Test.git> master

其中master为branch， 也可以换成其他的branch名字

1. 远程仓库相关命令

检出仓库：        $ git clone git://github.com/jquery/jquery.git

查看远程仓库：$ git remote -v

添加远程仓库：$ git remote add [name] [url]

删除远程仓库：$ git remote rm [name]

修改远程仓库：$ git remote set-url --push [name] [newUrl]

拉取远程仓库：$ git pull [remoteName] [localBranchName]

推送远程仓库：$ git push [remoteName] [localBranchName]

\*如果想把本地的某个分支test提交到远程仓库，并作为远程仓库的master分支，或者作为另外一个名叫test的分支，如下：

$git push origin test:master         // 提交本地test分支作为远程的master分支

$git push origin test:test              // 提交本地test分支作为远程的test分支

1. **分支(branch)操作相关命令**

查看本地分支：$ git branch

查看远程分支：$ git branch -r

创建本地分支：$ git branch [name] ----注意新分支创建后不会自动切换为当前分支

切换分支：$ git checkout [name]

创建新分支并立即切换到新分支：$ git checkout -b [name]

删除分支：$ git branch -d [name] ---- -d选项只能删除已经参与了合并的分支，对于未有合并的分支是无法删除的。如果想强制删除一个分支，可以使用-D选项

合并分支：$ git merge [name] ----将名称为[name]的分支与当前分支合并

创建远程分支(本地分支push到远程)：$ git push origin [name]

删除远程分支：$ git push origin :heads/[name] 或 $ git push origin :[name]

\*创建空的分支：(执行命令之前记得先提交你当前分支的修改，否则会被强制删干净没得后悔)

$git symbolic-ref HEAD refs/heads/[name]

$rm .git/index

$git clean -fdx

1. **版本(tag)操作相关命令**

查看版本：$ git tag

创建版本：$ git tag [name]

删除版本：$ git tag -d [name]

查看远程版本：$ git tag -r

创建远程版本(本地版本push到远程)：$ git push origin [name]

删除远程版本：$ git push origin :refs/tags/[name]

合并远程仓库的tag到本地：$ git pull origin --tags

上传本地tag到远程仓库：$ git push origin --tags

创建带注释的tag：$ git tag -a [name] -m 'yourMessage'

1. **子模块(submodule)相关操作命令**

添加子模块：$ git submodule add [url] [path]

如：$git submodule add git://github.com/soberh/ui-libs.git src/main/webapp/ui-libs

初始化子模块：$ git submodule init  ----只在首次检出仓库时运行一次就行

更新子模块：$ git submodule update ----每次更新或切换分支后都需要运行一下

删除子模块：（分4步走哦）

1) $ git rm --cached [path]

2) 编辑“.gitmodules”文件，将子模块的相关配置节点删除掉

3) 编辑“ .git/config”文件，将子模块的相关配置节点删除掉

4) 手动删除子模块残留的目录

1. **忽略一些文件、文件夹不提交**

在仓库根目录下创建名称为“.gitignore”的文件，写入不需要的文件夹名或文件，每个元素占一行即可，如

target

bin

\*.db

1. **Git基本的命令**

**git pull：**从其他的版本库（既可以是远程的也可以是本地的）将代码更新到本地，例如：'git pull origin master'就是将origin这个版本库的代码更新到本地的master主枝，该功能类似于SVN的**update**

**git add：**是将当前更改或者新增的文件加入到Git的索引中，加入到Git的索引中就表示记入了版本历史中，这也是提交之前所需要执行的一步，例如'git add app/model/user.rb'就会增加app/model/user.rb文件到Git的索引中，该功能类似于SVN的**add**

**git rm：**从当前的工作空间中和索引中删除文件，例如'git rm app/model/user.rb'，该功能类似于SVN的**rm、del**

**git commit：**提交当前工作空间的修改内容，类似于SVN的commit命令，例如'git commit -m story #3, add user model'，提交的时候必须用-m来输入一条提交信息，该功能类似于SVN的**commit**

**git push：**将本地commit的代码更新到远程版本库中，例如'git push origin'就会将本地的代码更新到名为orgin的远程版本库中

**git log：**查看历史日志，该功能类似于SVN的**log**

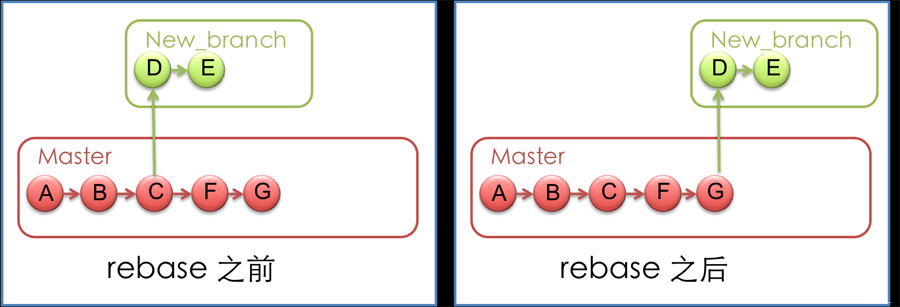
**git revert：**还原一个版本的修改，必须提供一个具体的Git版本号，例如'git revert bbaf6fb5060b4875b18ff9ff637ce118256d6f20'，Git的版本号都是生成的一个哈希值

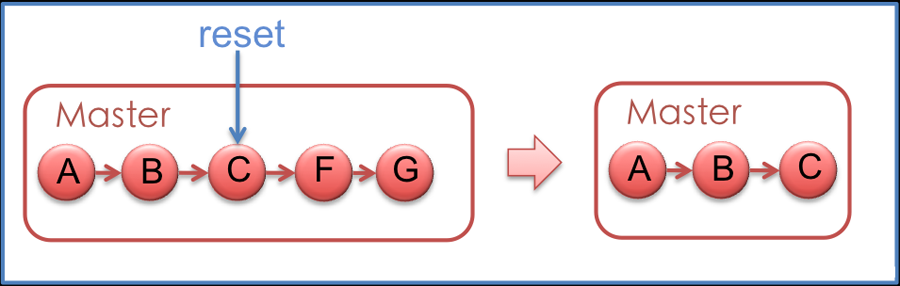
上面的命令几乎都是每个版本控制工具所公有的，下面就开始尝试一下Git独有的一些命令：

**git branch：**对分支的增、删、查等操作，例如'git branch new\_branch'会从当前的工作版本创建一个叫做new\_branch的新分支，'git branch -D new\_branch'就会强制删除叫做new\_branch的分支，'git branch'就会列出本地所有的分支

**git checkout：**Git的checkout有两个作用，其一是在不同的branch之间进行切换，例如'git checkout new\_branch'就会切换到new\_branch的分支上去；另一个功能是还原代码的作用，例如'git checkout app/model/user.rb'就会将user.rb文件从上一个已提交的版本中更新回来，未提交的内容全部会回滚

**git rebase：**用下面两幅图解释会比较清楚一些，rebase命令执行后，实际上是将分支点从C移到了G，这样分支也就具有了从C到G的功能

  
**git reset：**将当前的工作目录完全回滚到指定的版本号，假设如下图，我们有A-G五次提交的版本，其中C的版本号是 bbaf6fb5060b4875b18ff9ff637ce118256d6f20，我们执行了'git reset bbaf6fb5060b4875b18ff9ff637ce118256d6f20'那么结果就只剩下了A-C三个提交的版本

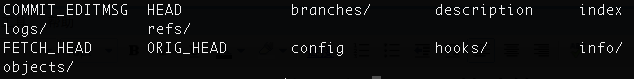


**git stash：**将当前未提交的工作存入Git工作栈中，时机成熟的时候再应用回来，这里暂时提一下这个命令的用法，后面在技巧篇会重点讲解

**git config：**利用这个命令可以新增、更改Git的各种设置，例如'git config branch.master.remote origin'就将master的远程版本库设置为别名叫做origin版本库，后面在技巧篇会利用这个命令个性化设置你的Git，为你打造独一无二的 Git

**git tag：**可以将某个具体的版本打上一个标签，这样你就不需要记忆复杂的版本号哈希值了，例如你可以使用'git tag revert\_version bbaf6fb5060b4875b18ff9ff637ce118256d6f20'来标记这个被你还原的版本，那么以后你想查看该版本时，就可以使用 revert\_version标签名，而不是哈希值了

Git 之所以能够提供方便的本地分支等特性，是与它的文件存储机制有关的。Git存储版本控制信息时使用它自己定义的一套文件系统存储机制，在代码根目录下有一个.git文件夹，会有如下这样的目录结构：

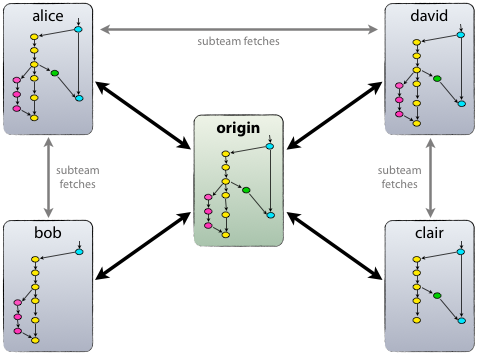


有几个比较重要的文件和目录需要解释一下：HEAD文件存放根节点的信息，其实目录结构就表示一个树型结构，Git采用这种树形结构来存储版本信息，那么HEAD就表示根；refs目录存储了你在当前版本控制目录下的各种不同引用（引用指的是你本地和远程所用到的各个树分支的信息），它有heads、remotes、stash、tags四个子目录，分别存储对不同的根、远程版本库、Git栈和标签的四种引用，你可以通过命令'git show-ref'更清晰地查看引用信息；logs目录根据不同的引用存储了日志信息。因此，Git只需要代码根目录下的这一个.git目录就可以记录完整的版本控制信息，而不是像SVN那样根目录和子目录下都有.svn目录。那么下面就来看一下Git与SVN的区别吧

# 怎么样利用Github在一个产品线（支持多个型号）的开发：

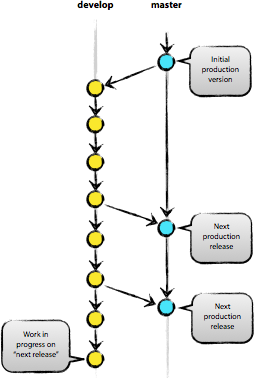
**1**

[**Git**](http://lib.csdn.net/base/git)，在技术层面上，绝对是一个无中心的分布式**[版本控制](http://lib.csdn.net/base/git" \t "_blank" \o "Git知识库)**系统，但在管理层面上，我建议你保持一个中心版本库。

[](http://roclinux.cn/wp-content/uploads/2010/12/centr-decentr.png)

**2**

我建议，一个中心版本库(我们叫它origin)至少包括两个分支，即“主分支(master)”和“开发分支(develop)”

[](http://roclinux.cn/wp-content/uploads/2010/12/git-master-develop.png)

**3**

要确保：团队成员从主分支(master)获得的都是处于可发布状态的代码，而从开发分支(develop)应该总能够获得最新开发进展的代码。

**4**

在一个团队开发协作中，我建议，要有“辅助分支”的概念。

**5**

“辅助分支”，大体包括如下几类：“管理功能开发”的分支、“帮助构建可发布代码”的分支、“可以便捷的修复发布版本关键BUG”的分支，等等

**6**

“辅助分支”的最大特点就是“生命周期十分有限”，完成使命后即可被清除。

**7**

我建议至少还应设置三类“辅助分支”，我们称之为“Feature branches”，“Release branches”，“Hotfix branches”。

至此，我们形成了如下这张最重要的组织组，包含了两个粗体字分支（master/develop）和三个细体字分支（feature/release/hotfixes）。

[](http://roclinux.cn/wp-content/uploads/2010/12/bigpicture-git-branch-all.png)

**8**

“Feature branches”，起源于develop分支，最终也会归于develop分支。

**9**

“Feature branches”常用于开发一个独立的新功能，且其最终的结局必然只有两个，其一是合并入“develop”分支，其二是被抛弃。最典型的“Fearture branches”一定是存在于团队开发者那里，而不应该是“中心版本库”中。

**10**

“Feature branches”起源于“develop”分支，实现方法是：

git checkout -b myfeature develop

**11**

“Feature branches”最终也归于“develop”分支，实现方式是：

git checkout devleop

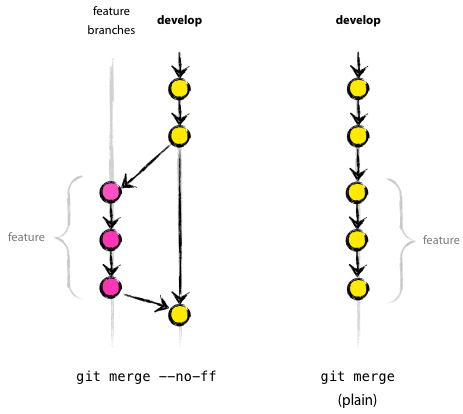
git merge --no-ff myfeature

（--no-ff，即not fast forward，其作用是：要求git merge即使在fast forward条件下也要产生一个新的merge commit）

（此处，要求采用--no-ff的方式进行分支合并，其目的在于，希望保持原有“Feature branches”整个提交链的完整性）

git branch -d myfeature

git push origin develop

[](http://roclinux.cn/wp-content/uploads/2010/12/merge-without-ff.png)

**12**

“Release branch”，起源于develop分支，最终归于“develop”或“master”分支。这类分支建议命名为“release-\*”

**13**

“Relase branch”通常负责“短期的发布前准备工作”、“小bug的修复工作”、“版本号等元信息的准备工作”。与此同时，“develop”分支又可以承接下一个新功能的开发工作了。

**14**

“Release branch”产生新提交的最好时机是“develop”分支已经基本到达预期的状态，至少希望新功能已经完全从“Feature branches”合并到“develop”分支了。

**15**

创建“Release branches”，方法是：

git checkout -b release-1.2 develop

./bump-version.sh 1.2 （这个脚本用于将代码所有涉及版本信息的地方都统一修改到1.2，另外，需要用户根据自己的项目去编写适合的bump-version.sh）

git commit -a -m "Bumped version number to 1.2"

**16**

在一段短时间内，在“Release branches”上，我们可以继续修复bug。在此阶段，严禁新功能的并入，新功能应该是被合并到“develop”分支的。

**17**

经过若干bug修复后，“Release branches”上的代码已经达到可发布状态，此时，需要完成三个动作：第一是将“Release branches”合并到“master”分支，第二是一定要为master上的这个新提交打TAG（记录里程碑），第三是要将“Release branches”合并回“develop”分支。

git checkout master

git merge --no-ff release-1.2

git tag -a 1.2 （使用-u/-s/-a参数会创建tag对象，而非软tag）

git checkout develop

git merge --no-ff release-1.2

git branch -d release-1.2

**18**

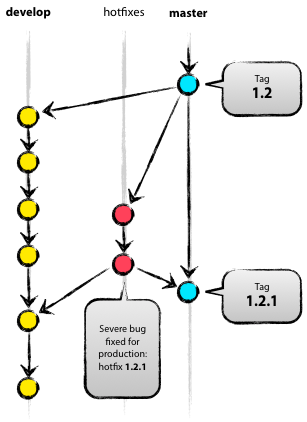
“Hotfix branches”源于“master”，归于“develop”或“master”，通常命名为“hotfix-\*”

**19**

“Hotfix branches”类似于“Release branch”，但产生此分支总是非预期的关键BUG。

**20**

建议设立“Hotfix branches”的原因是：希望避免“develop分支”新功能的开发必须为BUG修复让路的情况。

[](http://roclinux.cn/wp-content/uploads/2010/12/hotfix-branches1.png)

**21**

建立“Hotfix branches”，方法是：

git checkout -b hotfix-1.2.1 master

./bump-version.sh 1.2.1

git commit -a -m "Bumpt version to 1.2.1" （然后可以开始问题修复工作）

git commit -m "Fixed severe production problem" （在问题修复后，进行第二次提交）

**22**

BUG修复后，需要将“Hotfix branches”合并回“master”分支，同时也需要合并回“develop”分支，方法是：

git checkout master

git merge --no-ff hotfix-1.2.1

git tag -a 1.2.1

git checkout develop

git merge --no-ff hotfix-1.2.1

git branch -d hotfix-1.2.1

# Clone中断以后怎么继续clone TODO

???

# 远程GIT Server的维护，如何建分支，如何备份 TODO

????

# GIT submodule使用完整教程 TODO

http://www.kafeitu.me/git/2012/03/27/git-submodule.html

# GIT Server的安装和配置



# Reference:

## 常用 Git 命令清单

作者： [阮一峰](http://www.ruanyifeng.com/)

日期： [2015年12月 9日](http://www.ruanyifeng.com/blog/2015/12/)

我每天使用 Git ，但是很多命令记不住。

一般来说，日常使用只要记住下图6个命令，就可以了。但是熟练使用，恐怕要记住60～100个命令。





下面是我整理的常用 Git 命令清单。几个专用名词的译名如下。

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

一、新建代码库

# 在当前目录新建一个Git代码库

$ git init

# 新建一个目录，将其初始化为Git代码库

$ git init [project-name]

# 下载一个项目和它的整个代码历史

$ git clone [url]

$ git clone –o Jquery [url] #clone URL git的文件到本地并命名远程主机名字叫Jquery，否则就是默认的origin

下面例子是把本地的处在workspace的项目push到远程服务器上

git init

git add \*

git commit -m "first commit"

git remote add origin https://github.com/zju-chaoli/python\_test.git

git push -u origin master

二、配置

Git的设置文件为.gitconfig，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。

# 显示当前的Git配置

$ git config --list

# 编辑Git配置文件

$ git config -e [--global]

# 设置提交代码时的用户信息

$ git config [--global] user.name "[name]"

$ git config [--global] user.email "[email address]"

三、增加/删除文件

# 添加指定文件到暂存区

$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录

$ git add [dir]

# 添加当前目录的所有文件到暂存区

$ git add .

或者添加所有

Git Add \*

# 添加每个变化前，都会要求确认

# 对于同一个文件的多处变化，可以实现分次提交

$ git add -p

# 删除工作区文件，并且将这次删除放入暂存区

$ git rm [file1] [file2] ...

# 停止追踪指定文件，但该文件会保留在工作区

$ git rm --cached [file]

# 改名文件，并且将这个改名放入暂存区

$ git mv [file-original] [file-renamed]

四、代码提交

# 提交暂存区到仓库区

$ git commit -m [message]

# 提交暂存区的指定文件到仓库区

$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区

$ git commit -a

# 提交时显示所有diff信息

$ git commit -v

# 使用一次新的commit，替代上一次提交

# 如果代码没有任何新变化，则用来改写上一次commit的提交信息

$ git commit --amend -m [message]

# 重做上一次commit，并包括指定文件的新变化

$ git commit --amend [file1] [file2] ...

五、分支

# 列出所有本地分支

$ git branch

# 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]

# 新建一个分支，指向指定commit

$ git branch [branch] [commit]

# 切换到指定分支，并更新工作区

$ git checkout [branch-name]

# 切换到上一个分支

$ git checkout -

# 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]

# 建立追踪关系，在现有分支与指定的远程分支之间

$ git branch --set-upstream [branch] [remote-branch]

1. git branch --set-upstream master origin/next

上面命令指定 master 分支追踪 origin/next 分支。

# 合并指定分支到当前分支 Branch应该是本地分支

$ git merge [branch]

$ git merge origin/master

# 或者

$ git rebase origin/master

$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>

注意，分支推送顺序的写法是<来源地>:<目的地>，所以git pull是<远程分支>:<本地分支>，而git push是<本地分支>:<远程分支>。

# 选择一个commit，合并进当前分支

$ git cherry-pick [commit]

# 删除分支

$ git branch -d [branch-name]

# 删除远程分支

$ git push origin --delete [branch-name]

$ git branch –d -r [remote/branch]

六、标签

# 列出所有tag

$ git tag

# 新建一个tag在当前commit

$ git tag [tag]

# 新建一个tag对象而非软TAG

$ git tag –u/-s/-a [tag]

# 新建一个tag在指定commit

$ git tag [tag] [commit]

# 删除本地tag

$ git tag -d [tag]

# 删除远程tag

$ git push origin :refs/tags/[tagName]

# 查看tag信息

$ git show [tag]

# 提交指定tag

$ git push [remote] [tag]

# 提交所有tag

$ git push [remote] --tags

# 新建一个分支，指向某个tag

$ git checkout -b [branch] [tag]

七、查看信息

# 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commit历史，以及每次commit发生变更的文件

$ git log --stat

# 搜索提交历史，根据关键词

$ git log -S [keyword]

# 显示某个commit之后的所有变动，每个commit占据一行

$ git log [tag] HEAD --pretty=format:%s

# 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 [tag] HEAD --grep feature

# 显示某个文件的版本历史，包括文件改名

$ git log --follow [file]

$ git whatchanged [file]

# 显示指定文件相关的每一次diff

$ git log -p [file]

# 显示过去5次提交

$ git log -5 --pretty --oneline

# 显示所有提交过的用户，按提交次数排序

$ git shortlog -sn

# 显示指定文件是什么人在什么时间修改过

$ git blame [file]

# 显示暂存区和工作区的差异

$ git diff

# 显示暂存区和上一个commit的差异

$ git diff --cached [file]

# 显示工作区与当前分支最新commit之间的差异

$ git diff HEAD

# 显示两次提交之间的差异

$ git diff [first-branch]...[second-branch]

# 显示今天你写了多少行代码

$ git diff --shortstat "@{0 day ago}"

# 显示某次提交的元数据和内容变化

$ git show [commit]

# 显示某次提交发生变化的文件

$ git show --name-only [commit]

# 显示某次提交时，某个文件的内容

$ git show [commit]:[filename]

# 显示当前分支的最近几次提交

$ git reflog

八、远程同步

# 下载远程仓库的所有变动

$ git fetch [remote]

# 显示所有远程仓库

$ git remote -v

# 显示某个远程仓库的信息

$ git remote show [remote]

# 增加一个新的远程仓库，并命名

$ git remote add [shortname] [url]

删除远程仓库：

$ **git** remote rm [name]

修改远程仓库：

$ **git** remote set-url --**push**[name][newUrl]

$ git remote rename <原主机名> <新主机名>

# 取回远程仓库的变化，并与本地分支合并

$ git pull [remote] [branch]

$ git pull <远程主机名> <远程分支名>:<本地分支名>

# 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

# 把本地Dev分支推送到远程 Master分支

Git push origin dev:master

# 把本地dev分支推送到远程 DEV分支，，如果远程没有Dev分支则创建Dev分支

[**Git**](http://lib.csdn.net/base/git)**push  origin  本地分支A : 远程分支B ---> push 本地分支A到远程库origin的分支B**

Git push origin dev:dev

# 删除远程分支dev

Git push origin :dev

**1、origin/master ： 一个叫 origin 的远程库的 master 分支**

**2、HEAD指向当前工作的branch，master不一定指向当前工作的branch**

**3、[Git](http://lib.csdn.net/base/git" \t "_blank" \o "Git知识库)  push  origin  本地分支A : 远程分支B ---> push 本地分支A到远程库origin的分支B**

git push origin master

origin指定了你要push到哪个remote

master其实是一个“refspec”，正常的“refspec”的形式为”+<src>:<dst>”，冒号前表示local branch的名字，冒号后表示remote repository下 branch的名字。注意，如果你省略了<dst>，git就认为你想push到remote repository下和local branch相同名字的branch。听起来有点拗口，再解释下，push是怎么个push法，就是把本地branch指向的commit push到remote repository下的branch，比如

$git push origin master:master (在local repository中找到名字为master的branch，使用它去更新remote repository下名字为master的branch，如果remote repository下不存在名字是master的branch，那么新建一个)

$git push origin master （省略了<dst>，等价于“git push origin master:master”）

$git push origin master:refs/for/mybranch (在local repository中找到名字为master的branch，用他去更新remote repository下面名字为mybranch的branch)

$git push origin HEAD:refs/for/mybranch （HEAD指向当前工作的branch，master不一定指向当前工作的branch，所以我觉得用HEAD还比master好些）

$git push origin    :mybranch （再origin repository里面查找mybranch，删除它。用一个空的去更新它，就相当于删除了）

九、撤销

# 恢复暂存区的指定文件到工作区

$ git checkout [file]

# 恢复某个commit的指定文件到暂存区和工作区

$ git checkout [commit] [file]

# 恢复暂存区的所有文件到工作区

$ git checkout .

# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变

$ git reset [file]

# 重置暂存区与工作区，与上一次commit保持一致

$ git reset --hard

# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变

$ git reset [commit]

git reset --hard HEAD

# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致

$ git reset --hard [commit]

git reset --hard HEAD

# 重置当前HEAD为指定commit，但保持暂存区和工作区不变

$ git reset --keep [commit]

# 新建一个commit，用来撤销指定commit

# 后者的所有变化都将被前者抵消，并且应用到当前分支

$ git revert [commit]

# 暂时将未提交的变化移除，稍后再移入

$ git stash

$ git stash pop

十、压缩包

# 生成一个可供发布的压缩包

$ git archive

十一、子模块

添加子模块：$ **git** submodule add [url] [path]

如：$ **git** submodule add **git**://github.com/soberh/ui-libs.**git** src/main/webapp/ui-libs

初始化子模块：$ **git** submodule init ----只在首次检出仓库时运行一次就行

更新子模块：$ **git** submodule update ----每次更新或切换分支后都需要运行一下

或者$ **git** submodule update –init --recursive

删除子模块：（分4步走哦）

1)$ **git** rm --cached [path]

2) 编辑“.gitmodules”文件，将子模块的相关配置节点删除掉

3) 编辑“.**git**/config”文件，将子模块的相关配置节点删除掉

4) 手动删除子模块残留的目录

十二、Rebase

添加子模块：$ **git** submodule add [url] [path]