基础

切换分支 : **git** **checkout** -**b** **dev** **origin**/**dev**

拉取代码与远程库同步(所有分支) : **git** **pull**

查状态 : **git** **status**

查提交日志：**git log**

提交暂存区 : **git** **add** -**A**全部 **git** **add** **x**.**txt** 单个文件 **git add \*.java**某类文件

提交本地仓库 : **git** **commit** -**m** '注释'

切换分支 : **git checkout master**

合并代码 : **git** **merge** **dev**

提交远程分支 : **git** **push** **origin** **head**:**refs**/**for**/**master （gerrit**的规则，先审核后入库**）**

**或者**

**git** **push** **origin dev （**git的规则**）**

同步当前分支 : **git** **merge master**

git本地新建一个分支后，必须要做远程分支关联。先新建远程分支再关联。【追踪】

【修改最后一个 **commit**的注释】：# **git commit --amend** 然后在出来的编辑界面，直接编辑注释的信息,保存退出 (**:wq!**)

**git commit -a -m "提交的描述信息"**

git commit 命令的-a 选项可只将所有被修改或者已删除的且已经被git管理的文档提交倒仓库中。如果只是修改或者删除了已被Git 管理的文档，是没必要使用git add 命令的。

git add .命令除了能够判断出当前目录（包括其子目录）所有被修改或者已删除的文档，还能判断用户所添加的新文档，并将其信息追加到索引中。

**git add -A** 保存所有的修改

**git add .** 保存新的添加和修改，但是不包括删除

**git add -u** 保存修改和删除，但是不包括新建文件

(use "git reset HEAD <file>..." to unstage)

**git remote –v :** 查看当前远程服务器地址

**生成秘钥：**

依次运行以下命令

**git config --global user.name** XXXX #请将XXXX替换为你的gerrit中的“ Username”

**git config --global user.email** YYYY #将YYYY替换为你的公司邮箱

**ssh-keygen** （设置密码处按enter即可）

**eval `ssh-agent`; ssh-add;**

**cat ~/.ssh/id\_rsa.pub** #复制显示出的文件内容

打开ssh-key设置页面，点击"Add key"将上一步骤中复制的内容粘贴并追加到秘钥列表

查看用户名和邮箱：

**$ git config user.name**

**$ git config user.email**

克隆

**clone**地址 : **git** **clone** **ssh**://**zhangsan**@**gerrit**.**dev**.**baidu**.**com**:29418/**bridge**.**git**

第一次**clone**一个项目(**Clone** **with** **commit**-**msg** **hook**)

**git** **clone** **ssh**://**zhangsan**@**gerrit**.**dev**.**baidu**.**com**:29418/**IronMan** && **scp** -**p** -**P** 29418 **zhangsan**@**gerrit**.**dev**.**baidu**.**com**:**hooks**/**commit**-**msg** **IronMan**/.**git**/**hooks**/

拉取代码

**git** **pull**命令的作用是，取回远程主机某个分支的更新，再与本地的指定分支合并。它的完整格式稍稍有点复杂。

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

比如，取回**origin**主机的**dev**分支，与本地的**master**分支合并，需要写成下面这样。(可以在本地做关联之后 省略远程分支名)

$ **git** **pull** **origin** **dev**:**master**

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** **pull** **origin** **dev**

上面命令表示，取回**origin**/**dev**分支，再与当前分支合并。实质上，这等同于先做**git** **fetch**，再做**git** **merge**。

$ **git** **fetch** **origin |**

| **=** $ **git** **pull** **origin** **dev**

$ **git** **merge** **origin**/**dev |**

分支管理

1、【创建分支】

创建本地分支很简单：**git** **branch** <分支名>

创建远程分支/推送本地分支：(先基于某一分支创建本地分支，然后不做任何修改把分支推送到远程，否则gerrit权限不允许)

**$ git push origin dev**

2、【切换分支】

**git** **checkout** <分支名>

该语句和上一个语句可以和起来用一个语句表示：**git** **checkout** -**b** <分支名>

创建新分支：**git** **branch** **branchName**

切换到新分支：**git** **checkout** **branchName**

上面两个命令也可以合成为一个命令：**git** **checkout** -**b** **branchName**

3、【分支合并】

比如，如果要将开发中的分支（**develop**），合并到稳定分支（**master**），

首先切换的**master**分支，并更新。

然后执行合并操作：**git** **merge** **develop**。

如果有冲突，会提示你，调用**git** **status**查看冲突文件。

解决冲突，然后调用**git** **add**或**git** **rm**将解决后的文件暂存。

**所有冲突解决后，git commit 提交更改**。

4、【分支衍合】

分支衍合和分支合并的差别在于，分支衍合不会保留合并的日志，不留痕迹，而 分支合并则会保留合并的日志。

要将开发中的分支（**develop**），衍合到稳定分支（**master**）。

首先切换的**master**分支：**git** **checkout** **master**。

然后执行衍和操作：**git** **rebase** **develop**。

如果有冲突，会提示你，调用**git** **status**查看冲突文件。

解决冲突，然后调用**git** **add**或**git** **rm**将解决后的文件暂存。

所有冲突解决后，**git** **rebase** --**continue** 提交更改。

5、【删除分支】

( 创建远程分支/推送本地分支： **$ git push origin dev** )

git 删除本地分支：**git** **branch** -**d** <分支名>

如果该分支没有合并到主分支会报错，

可以用以下命令**强制**删除**git** **branch** -**D** <分支名>

git 删除远程分支：**git push origin :dev**  (origin 后面有空格)

关联本地与远程的分支：（查看.git/config文件是否被修改）

$ git branch --set-upstream dev origin/dev（**官方已不推荐**）

或者

$ git branch --set-upstream-to=origin/dev dev

Branch dev set up to track remote branch dev from origin.

取消对当前分支dev的跟踪

git branch --unset-upstream **dev**

.git/config文件：

…

[branch "master"]

remote = origin

merge = refs/heads/master

[branch "dev"]

remote = origin

merge = refs/heads/dev

6. 【分支合并】[**git**](http://lib.csdn.net/base/git)**merge** 用来做分支合并，将其他分支中的内容合并到当前分支中。

7. 【查看所有分支（本地和远程分支）】

**$ git branch –a** （需要先更新）

（前面带\*号的代表你当前工作目录所处的分支）

8. 【重命名分支】：**git branch -m devel develop**

（先取消远程分支关联、删除远程分支、重命名本地分支、推送到远程创建新分支、关联远程分支）

撤销

【恢复某个已修改的文件（撤销未提交的修改）（无法恢复！）】：

**git** **checkout** **file**-**name**

如：**git** **checkout** \*.**java**

撤销所有修改**git** **checkout** .

【还原已提交的修改：**revert】**

**git revert** #原理：根据你要回退的提交所做的改动做相反的改动，然后重新提交代码，使代码达到没有这些旧提交所能达到的状态。

**git revert** #用于反转提交,执行**revert**命令时要求工作树必须是干净的.

(所以执行两次git revert head代码就会恢复这两次前的样子)

还原最近一次提交的修改：$ **git** **revert** **HEAD**

还原指定版本的修改：$ **git** **revert** **commit**-**id**

**git** **checkout** . #本地所有修改的。没有的提交的，都返回到原来的状态

**git** **stash** #把所有没有提交的修改暂存到**stash**里面。可用**git** **stash** **pop**恢复。

【回退到某个版本：**reset】**

git reset 在提交层面上，reset将一个分支的末端指向另一个提交。

**git** **reset** --**hard** **HASH** #回退到某个节点，不保留修改。

**git** **reset** --**soft** **HASH** #回退到某个节点。保留修改

**（ 默认是mixed参数）**

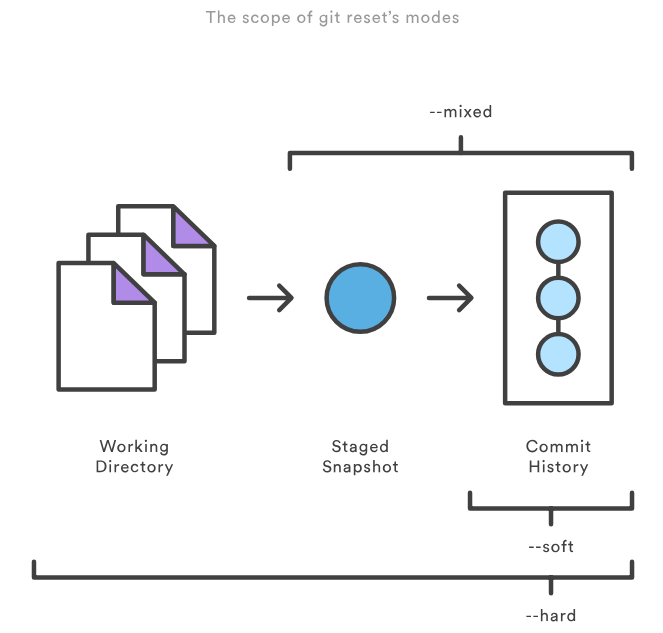
**git reset HEAD #**任何事情都不会发生，这是因为我们告诉GIT重置这个分支到HEAD，而这个正是它现在所在的位置。

**git reset HEAD^ #**回退到上一次提交（工作区保留修改），与**git** **revert** **HEAD**类似

上上一个版本就是HEAD^^，当然往上100个版本可以写成HEAD~100。

还可以**git reset master、** **git reset commit-id** 如果不加参数，实际上使用的是默认的参数mixed

**git reset HEAD <file> #**取消已经暂存的文件（撤销**git add**的文件）



【撤销一个“已公开”的改变】

场景: 你已经执行了 **git** **push**, 把你的修改发送到了 **GitHub**，现在你意识到这些 **commit** 的其中一个是有问题的，你需要撤销那一个 **commit**.

方法: **git** **revert** <**SHA**>

原理: **git** **revert** 会产生一个新的 **commit**，它和指定 **SHA** 对应的 **commit** 是相反的（或者说是反转的）。如果原先的 **commit** 是“物质”，新的 **commit** 就是“反物质” — 任何从原先的 **commit** 里删除的内容会在新的 **commit** 里被加回去，任何在原先的 **commit** 里加入的内容会在新的 **commit** 里被删除。

这是 **Git** 最安全、最基本的撤销场景，因为它并不会改变历史 — 所以你现在可以 **git** **push** 新的“反转” **commit** 来抵消你错误提交的 **commit**。

【在撤销“本地修改”之后再恢复】

场景: 你提交了几个 **commit**，然后用 **git** **reset** --**hard** 撤销了这些修改（见上一段），接着你又意识到：你希望还原这些修改！

方法: **git** **reflog** 和 **git** **reset** 或 **git** **checkout**

**或者**

**git** **log**和**git** **reset** --**hard** 3628164(**commit**\_**id**不必写全)

原理: **git** **reflog** 对于恢复项目历史是一个超棒的资源。你可以恢复几乎 任何东西 — 任何你 **commit** 过的东西 — 只要通过 **reflog**。

你可能已经熟悉了 **git** **log** 命令，它会显示 **commit** 的列表。 **git** **reflog** 也是类似的，不过它显示的是一个 **HEAD** 发生改变的时间列表.

【一些注意事项】：

它涉及的只是 **HEAD** 的改变。在你切换分支、用 **git** **commit** 进行提交、以及用 **git** **reset** 撤销 **commit** 时，**HEAD** 会改变，但当你用 **git** **checkout** -- <**bad** **filename**> 撤销时（正如我们在前面讲到的情况），**HEAD** 并不会改变 — 如前所述，这些修改从来没有被提交过，因此 **reflog** 也无法帮助我们恢复它们。

**git** **reflog** 不会永远保持。**Git** 会定期清理那些 “用不到的” 对象。不要指望几个月前的提交还一直躺在那里。

你的 **reflog** 就是你的，只是你的。你不能用 **git** **reflog** 来恢复另一个开发者没有 **push** 过的 **commit**。

查看日志

git log --graph命令可以看到分支合并图。

git log --oneline --graph –decorate可以看到一个仓的直线提交图标。

git log –stat查看当前分支提交了哪些文件

git reflog可以查看所有分支的所有操作记录，包括已经被删除的commit记录

参数：  
-n (n是一个正整数)，查看最近n次的提交信息 如git log -3

对比

本地与远程对比：

git diff <local branch> <remote>/<remote branch>

统计文件的改动：

git diff --stat local\_branch origin/remote\_branch

对比当前版本与上一版本 \是转义的作用：

git diff HEAD\^ HEAD

Git比较两个分支间所有变更的文件列表：

[Git](http://lib.csdn.net/base/git) diff branch1 branch2 --stat

git diff <file> # 比较当前文件和暂存区文件差异 git diff

git diff <id1><id2> # 比较两次提交之间的差异

git diff <branch1>..<branch2> # 在两个分支之间比较

git diff --staged # 比较暂存区和版本库差异

git diff --cached # 比较暂存区和版本库差异

git diff --stat # 仅仅比较统计信息

覆盖更新（慎用！！）

git fetch --all拉取 (git fetch origin dev)

git reset --hard origin/master覆盖

标签管理

我们常常在代码封板时,使用git 创建一个tag ,这样一个不可修改的历史代码版本就像被我们封存起来一样,不论是运维发布拉取,或者以后的代码版本管理,都是十分方便的.

首先我们了解下 git 的 tag 功能:

git 下打标签其实有2种情况

(1): 轻量级的

它其实是一个独立的分支,或者说是一个不可变的分支.指向特定提交对象的引用.

(2):带附注的

实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着标签的名字，标签说明，标签本身也允许使用 GNU Privacy Guard (GPG) 来签署或验证,电子邮件地址和日期，一般我们都建议使用含附注型的标签，以便保留相关信息.

所以我们推荐使用第二种标签形式.

创建标签:

|  |  |
| --- | --- |
| 1 | zhangzhi@moke:~/code/demo$git tag -a V1.2 -m 'WebSite version 1.2' |

上面的命令我们成功创建了本地一个版本 V1.2 ,并且添加了附注信息 'WebSite version 1.2'

查看标签:

|  |  |
| --- | --- |
| 1  2  3  4 | zhangzhi@moke:~/code/demo$ git tag  V1.0  V1.1  V1.2 |

我们看到本地有3个标签,最下面的 V1.2 就是刚才提交的,但是没有显示出来我们附注信息.

要显示附注信息,我们需要用 show 指令来查看.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | zhangzhi@moke:~/code/demo$ git show V1.2  tag V1.2  Tagger: zhangzhi <zzhi.net@gmail.com>  Date:   Wed Jul 21 18:42:35 2015 +0800    WebSite version 1.2    commit 853cecc248756d0189d2a328dc22fa6781a65265  Merge: 1f23b2b 00cfb5b  Author: zhangzhi <zzhi.net@gmail.com>  Date:   Wed Jul 21 18:38:36 2015 +0800        Merge branch 'master' into test |

上面显示出来了 V1.2 标签的所有提交信息.

但是目前这个标签仅仅是提交到了本地git仓库.如何同步到远程代码库

tag 远程推送:

|  |  |
| --- | --- |
| 1 | zhangzhi@moke:~/code/demo$git push origin --tags |

这样我们就把本地版本推送到了远程代码仓库.

 如果刚刚同步上去,你缺发现一个致命bug ,需要重新打版本,现在还为时不晚.

 删除标签:

|  |  |
| --- | --- |
| 1 | zhangzhi@moke:~/code/demo$git tag -d V1.2 |

到这一步我们只是删除了本地 V1.2的版本,可是线上V1.2的版本还是存在,如何办?

这时我们可以推送的空的同名版本到线下,达到删除线上版本的目标:

|  |  |
| --- | --- |
| 1 | zhangzhi@moke:~/code/demo$git push origin :refs/tags/V1.2 |

这时本地和远程的 V1.2 版本已经被我们移除掉.

如何获取远程版本:

|  |  |
| --- | --- |
| 1 | zhangzhi@moke:~/code/demo$git fetch origin tag V1.2 |

这样我们可以精准拉取指定的某一个版本.适用于运维同学部署指定版本.

新建仓库

1. 创建文件夹
2. 进去文件夹，然后执行 git init
3. 在远程创建仓库（私有仓库需要交费）
4. 本地与远程关联：git remote add origin [git@github.com:username/spring.git](mailto:git@github.com:username/spring.git)
5. 关联推送：git push --set-upstream origin master
6. 合并git pull origin master --allow-unrelated-histories
7. 下一次就可以用 git push origin和git pull

修改与远程仓库的关联地址

git remote –v 查看当前远程服务器地址

方法有三种：

1.修改命令

git remote set-url origin [url]

2.先删后加

git remote rm origin

git remote add origin [url]

3.直接修改config文件

跟踪

# 删除 untracked files（未跟踪文件）

git clean -f

# 连 untracked 的目录也一起删掉

git clean -fd

# 连 gitignore 的untrack 文件/目录也一起删掉 （慎用，一般这个是用来删掉编译出来的 .o之类的文件用的）

git clean -xfd

# 在用上述 git clean 前，墙裂建议加上 -n 参数来先看看会删掉哪些文件，防止重要文件被误删

git clean -nxfd

git clean -nf

git clean -nfd

#取消跟踪文件：

git rm --cached readme1.txt    删除readme1.txt的跟踪，并保留在本地。

git rm --f readme1.txt    删除readme1.txt的跟踪，并且删除本地文件。

Git中.gitignore文件不起作用

git rm -r --cached .

git add .

git commit -m 'update .gitignore'

问题：

git pull时发生分支合并(gerrit)

Merge made by the 'recursive' strategy.

git log 查看没有change-id

Merge branch 'xmen' of ssh://gerrit.dev.baidu.com:29418/Xmen into xmen