## 技术分享（？）

**创建版本库(repository)**

mkdir [选项] [目录] 创建指定的名称的目录

cd 目录名 进入某个文件夹

pwd 显示当前目录

git init 初始化本地版本库（ls-ah看到隐藏的git目录）

**把文件添加到版本库**

注：所有的版本控制系统只能跟踪文本文件的改动，图片视频等二进制格式可管理不可跟踪。

编写文件（一定要放到创建的目录或其子目录下）

vi file 打开或新建文件（使用字母i或inster键进入最常用的插入编辑模式）

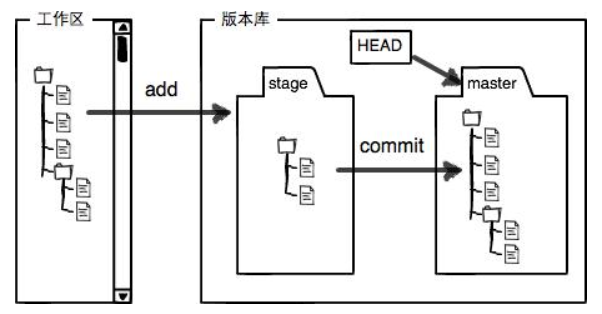
注：结束的时候按esc，输入”：w”仅保存；输入“：wq”保存并退出。；输入”：q!“取消所有修改并推出；输入”：e！“取消所有修改并不退出。

（其他关于vi/vim的操作见附C）

git add file 把文件添加到暂存区（stage）

git commit file 把暂存区的所有文件提交到当前分支（master）

（git commit -m”说明”）



git revert 用一个新提交来消除一个历史提交所做的任何修改

git revert commit-id revert指定的一个commit

git revert HEAD~n revert指定倒数第n+1个commit

git revert --abort 终止此次revert操作(代码还原至revert 之前)

git status 查看版本库当前状态

git diff file 查看difference

注：（关于工作区和暂存区）

工作区（working directory），电脑里能看到的目录；

版本库(repository)，隐藏目录；

暂存区(stage/index)，属于版本库；

指针（HEAD）；

git add→暂存区→git commit→一次性提交到分支

**版本回退**

git log 查看从最近到最远的提交日志

git log --pretty=oneline 仅查看commit id（版本号）

注：git的commit id 是一个SHA1计算出来的非常大的数字，用十六进制表示。

git reset --hard HEAD^ 回退到上一个版本

注：1、在git中HEAD代表当前版本，HEAD^代表上一版本，HEAD^^代表上上个版本；

但如果要回退到往上100个版本则需使用HEAD~100来表示往上100个版本。

1. git版本回退速度非常快，因为git内部有一个HEAD指针，当版本回退的时候git直接将指针从当前版本移动到下一个版本

git reset --hard commit id 回退到commit id所在版本

回退之后如何复原 （关键：找到commit id然后使用git reset --hard commit id指令）

如果窗口还在可直接获得；

如果时间间隔长：

git reflog 查看命令历史，找到commit id

git stash 把所有未提交的修改保存起来

git stash save “message” 给每次修改加上信息

git stash pop/git stash apply 恢复最新缓存的工作目录（第一个），并删除缓存堆栈中的那一个stash删除(pop), apply则只恢复不删除

git stash list 查看现有所有修改

git stash drop 移除最新的stash，后面也可以跟指定stash的名字

关于Reset/checkout/revert区别

|  |  |  |
| --- | --- | --- |
|  | 文件层面 | 提交层面 |
| git reset | 将文件从缓存区移除 | 在私有分支上舍弃一些没提交的修改 |
| git checkout | 舍弃工作目录中的更改 | 切换分支或者查看旧版本 |
| git revert | 用一个新提交替换旧提交 | 在公共分支上回滚更改 |

**管理修改**

注：每次修改如果不用git add则不会加入到commit中。

**回退修改**

git checkout **--** file 丢弃工作区的修改

注：存在两种情况，一种是文件自修改后还没有被放到暂存区，撤销修改就回到和版本库一模一样的状态；另一种是文件已经添加到暂存区后，又作了修改，撤销修改就回到添加到暂存区后的状态。总之让文件回到最后一次add或commit的状态。

git reset <file> 把暂存区的修改撤销掉

**删除文件**

rm file 仅删除工作区文件

git rm --cached file 仅删除暂存区文件

git f file 工作区和暂存区均删除

git rm file 删除版本库中的文件

若误删

git checkout 用版本库中的文件替换工作区文件

注：只能恢复文件到最新版本，会丢失最后一次提交后修改的内容。

**远程仓库**

1. **创建SSH Key。**在主目录下，用查看有没有.ssh目录，如果有，用ls查看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Git Bash，创建SSH Key；

创建SSH key具体步骤:

1、git config --global user.name'用户名' 配置用户名

git config --global user.email'邮箱' 配置邮箱

2、cd ~/.ssh/    如果没有对应的文件夹，则执行  mkdir  ./.ssh

3、ssh-keygen -t rsa -C " "

4、cat 文件名.pub 获取SSH key

5、github添加SSH key

1. 登陆GitHub，打开“Account settings”，“SSH Keys”页面，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容。

**添加远程仓库**

1. 登陆GitHub，在右上角找到“Create a new repo”，创建一个新的仓库；
2. 在Repository name填入仓库名，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库；
3. 在本地的仓库下运行命令

git remote add origin git@github.com:自己的账户名/仓库名.git

git push 把本地库的内容推送到远程（把当前分支master推送到远程）

git push origin master 把本地master分支的最新修改推送至GitHub

git remote 查看远程库信息

git remote -v 显示更详细的信息

注：SSH警告输入yes即可；若担心有人冒充github服务器则在yes之前对照GitHub的RSA Key的指纹信息，看是否与SSH连接给出的一致。

git remote remove origin 取消本地库与远程仓库的关联

**克隆（从零开发）**

最好先创建远程库再从远程库克隆。

git clone git@github.com:自己的账户名/仓库名.git 克隆一个本地库

注：1、GitHub给出的地址不止一个，默认的git://使用ssh，但也可以使用https等其他协议。

2、但使用https速度慢，每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。

**创建与合并分支**

git checkout －b 分支名 创建并切换到分支

git checkout 分支名 切换到分支

git switch －c 分支名 创建切换到分支

git switch 分支名 切换到分支

git branch 显示所有分支，当前分支前会有一个\*

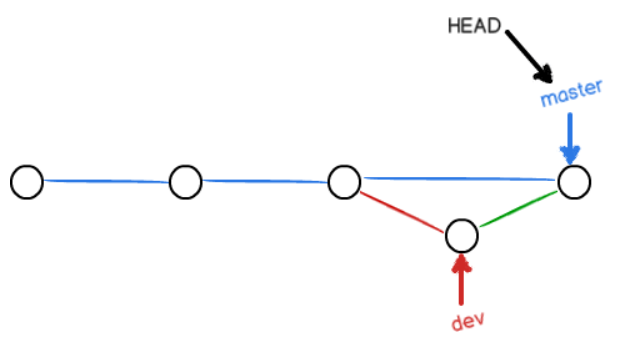
git merge 分支名 合并指定分支到当前分支

git branch －d 分支名 删除分支

**分支管理策略**

通常合并分支时Git会用Fast forward模式，但这种模式下，删除分支后会丢掉分支信息。

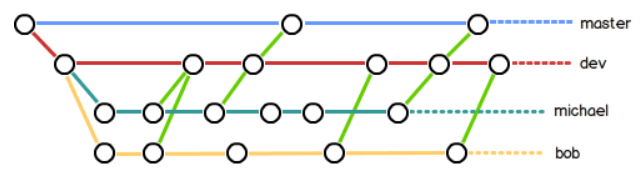
--no-ff 参数 禁用fast forward



禁用fast forward之后的分支

**分支策略**

即小队成员在分支上完成工作后提交到master上共同构建项目



·bug分支

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。（灵活运用git stash，见版本回退）

git cherry-pick commit id 复制一个特定的提交到当前分支

·feature分支

开发一个新feature，最好新建一个分支

git branch -D 分支名 强制删除分支

注：1、master分支是主分支，因此要时刻与远程同步；

2、dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；

3、bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；

4、feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

多人协作的工作流程：

1、首先，可以试图用git push origin <branch-name>推送自己的修改；

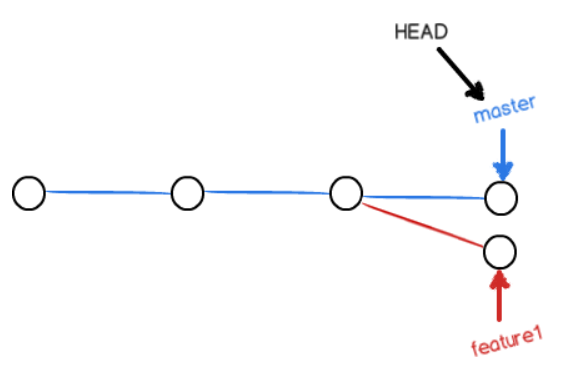
2、如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；

3、如果合并有冲突，则解决冲突，并在本地提交；

4、没有冲突或者解决掉冲突后，再用git push origin <branch-name>推送就能成功！

5、如果git pull提示no tracking information，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream-to <branch-name> origin/<branch-name>。

**解决冲突**

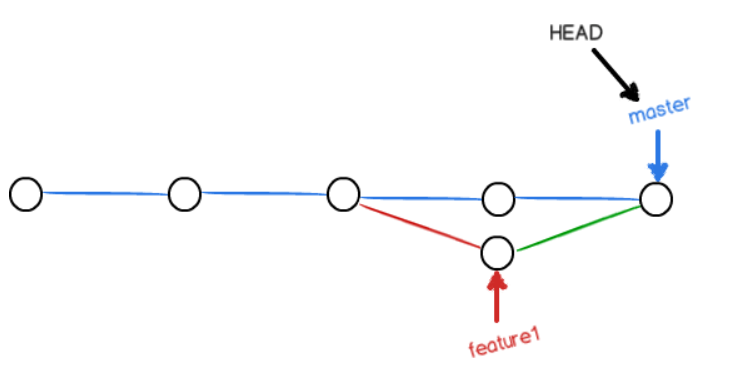


注：上图情况Git无法实现快速合并，只能**试图**把各自的修改合并；

feature1分支与master冲突

查看文件内容，Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容；

手动修改后提交。



feature1分支与master合并

git log --graph 看到分支合并图

·直接编辑冲突文件

git branch --set-upstream-to=origin/分支名 设置分支与远程库分支的链接

git pull 把最新的提交从远程库上抓取下来手动解决冲突

git rebase 分叉的提交历史“整理”成一条直线，看上去更直观。缺点是本地的分叉提交已经被修改过了。（自动rebase，其他有关rebase的操作见附B）

·利用图形界面工具解决冲突

**标签管理**

Git的标签（tag）是版本库的快照(即指向某个commit的指针)

跟分支的区别：分支可以移动，标签不能移动

·创建标签

切换到需要打标签的分支上

git tag 标签名（如:v1.3）(commit id) 给该分支(的commit)打上标签

git tag 查看所有标签

·操作标签

git tag -d 标签名 删除该标签

git push origin 标签名 把标签推送到远程

git push origin --tags 一次性推送所有标签

删除远程标签：

先删除本地标签 git tag -d 标签名

再删除远程标签 git push origin :refs/tags/标签名

**追踪关系**

建立test仓库 并建立追踪关系

git branch --track test origin/master 修改追踪关系

git checkout test 先切换到test

git branch --set-upstream-to origin/master 修改追踪仓库（一定要先切换）

建立追踪关系之后，本地分支名称和远程一样时，使用git push时不用带上远程名称，git pull也不用带上远程分支名

附A

·代码提交的完整流程

git add -A

git commit -m “message”

git pull --rebase (或者git fetch + git rebase）

解决冲突

git add 冲突文件

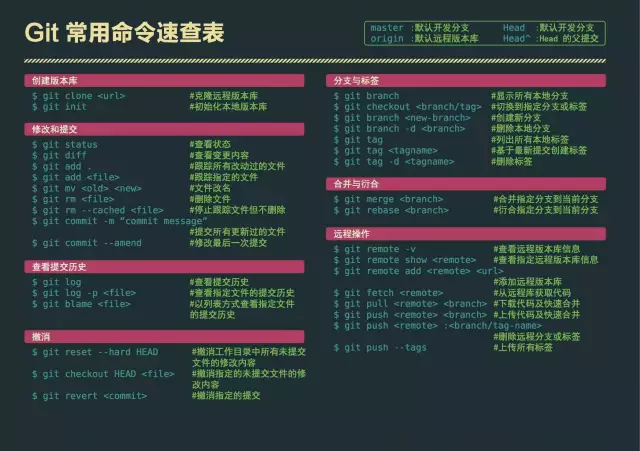
git rebase –continue

git push

其中，3、4、5点，如果没遇到冲突就不用进行，直接push上去。

当遇到冲突时，git会提示patch failed，并要我们解决问题了再执行git rebase --continue

·cit常用命令速查表



附B

**·交互式rebase**(用于将feature分支提交到master之前清理混乱的历史)

git checkout feature

git rebase -i master

进入文本编辑器，它会显示所有将被移动的提交

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

**·rebase的黄金法则**（绝不要在公共的分支上使用)

**·rebase的本地清理功能**

git checkout feature

git rebase -i HEAD~3(或者第四个commit-id)

附C

