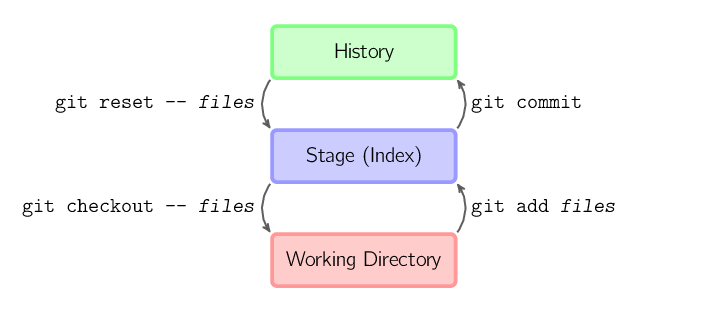
Git学习笔记

## 一、Git是什么

* Git是一个**分布式**的版本控制系统，一般来讲，各个Git仓库没有主次之分。分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作；
* 大多数的操作可以在本地完成，事后方便时，再推送到中心服务器的仓库中；
* 采用**“直接记录快照，而非差异比较”**的版本控制策略，内部只关心**文件数据的整体是否发生改变，而不是文件内容的具体差异**（Git内部被实现为一种微型的文件系统）；
* Git工作时就是在**工作目录（工作区、work tree）、暂存区（索引、index file）、本地仓库**三者之间管理文件的变化情况，Git会监视工作目录中的文件变化(增加新文件，删除文件，修改文件等)，需要我们自己手动将变化的文件添加(git add)到暂存区中（这就是文件快照），然后再提交(git commit)到本地仓库中；



* 上述过程，涉及Git内部的三种对象：commit对象、tree对象和blob对象，blob对象会对应的文件快照中那些变化的文件内容，tree对象记录了文件快照中各个目录和文件的结构关系，从概念上讲，tree对象和blob对象组成了文件快照，commit对象则记录了这次要提交到本地仓库的文件快照，同时也会指向上次的commit对象，它也是Git内部进行版本控制的重点（Git内部会记录各个commit对象，并用HEAD来指示当前分支中最新的commit对象），很多重要的功能，如分支、版本回溯、Git仓库内部状态等都是在commit对象基础上实现的；上述的每一个对象都对应一个独一无二的ID，该ID是一个由40个字符组成的哈希码，由SHA-1算法计算而来；Git能够通过ID的前几个字符就识别出对应的对象；
* Git的分支功能很强大，很灵活，切换速度非常快，并且实现成本很低，这也是Git比其他版本控制软件要受欢迎的原因之一。

## 二、基本操作

**1. 项目创建/克隆**

通过git init命令把当前目录变成Git可以管理的仓库；

要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆。

**2. 工作提交**

用命令git add告诉Git，把文件添加到仓库

用命令git commit告诉Git，把文件提交到仓库

git commit –a合并上述两个命令

**3. 查看区别**

git status命令可以让我们时刻掌握仓库当前的状态

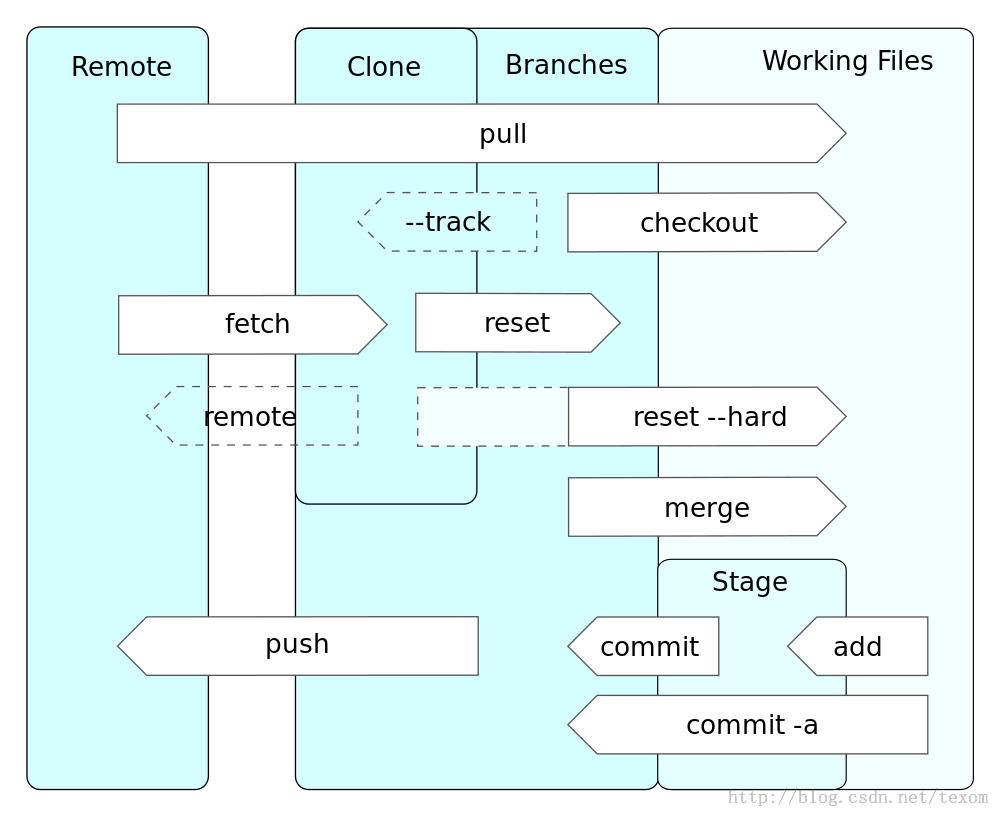
git diff 显示已写入缓存与已修改但尚未写入缓存的改动的区别

git diff --cached  查看已缓存的改动

git diff HEAD查看已缓存的与未缓存的所有改动

git diff --stat 显示摘要而非整个diff

上面指令后面加上-filename，可以显示某个文件的改变。



**4. 撤销操作**

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout --file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

**5. 回退操作**

首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上10个版本写10个^比较容易数不过来，所以写成HEAD~10

git reset --hard HEAD^

git reflog用来记录你的每一次命令，查看commit记录可以帮助定位想要回退的位置。

**6. 分支操作**

Git鼓励大量使用分支：

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建+切换分支：git checkout -b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

**7. 标签管理**

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

命令git tag <name>用于新建一个标签，默认为HEAD，也可以指定一个commit id；

git tag -a <tagname> -m "blablabla..."可以指定标签信息；

git tag -s <tagname> -m "blablabla..."可以用PGP签名标签；

命令git tag可以查看所有标签。

命令git push origin <tagname>可以推送一个本地标签；

命令git push origin --tags可以推送全部未推送过的本地标签；

命令git tag -d <tagname>可以删除一个本地标签；

命令git push origin :refs/tags/<tagname>可以删除一个远程标签。

**8. 远程库**

查看远程库信息，使用git remote -v；

从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交；

在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致；

建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name；

git fetch 会使你与另一仓库同步，提取你本地所没有的数据

从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

## 三、分支策略

在实际开发中，我们应该按照几个基本原则进行分支管理：

* master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；
* 干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并。

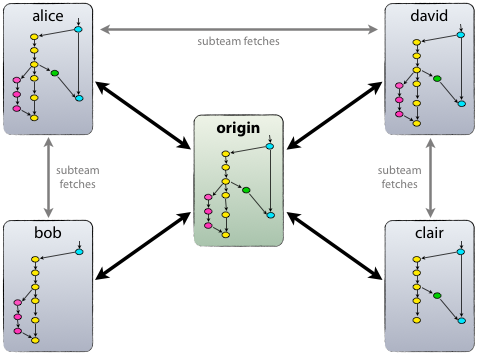
所以，团队合作的分支看起来就像这样：



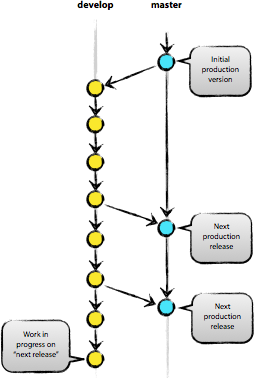
* 添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以每添加一个新功能，最好新建一个feature分支，在上面开发，完成后合并，最后删除该feature分支。
* 修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场。

## ****分支管理艺术****

**1.** Git在技术层面上，绝对是一个无中心的分布式版本控制系统，但在管理层面上,建议保持一个中心版本库。



2. 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)”。



**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，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

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. 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）三个细体字分支（feature/release/hotfixes）：



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

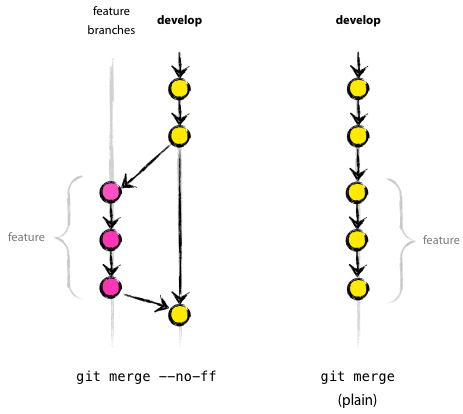
11.“Feature branches”最终也归于“develop”分支，实现方式是：

git checkout develop

git merge --no-ff myfeature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



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

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

git checkout develop

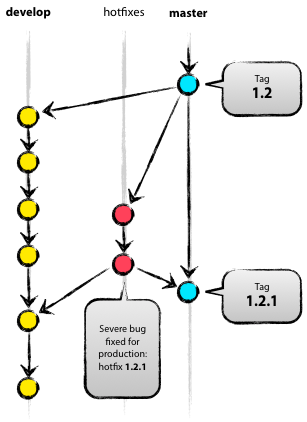
git merge --no-ff release-1.2

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。



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"

git commit -m "Fixed severe production problem"

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