Git教程

# 本地文件版本管理

**创建：**git init

备注：首先这里再明确一下，所有的版本控制系统，其实只能跟踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。版本控制系统可以告诉你每次的改动，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

**添加**：git add file\_name

**提交：**git commit –m ‘’注释”

**查看状态**：git status

**查看最近修改**：git diff file\_name

**查看提交日志**：git log

备注：看到的一大串类似1094adb...的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示。

为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

每提交一个新版本，实际上Git就会把它们自动串成一条时间线。如果使用可视化工具查看Git历史，就可以更清楚地看到提交历史的时间线：

**回滚：**git reset –hard HEAD^ #回到上一个版本

git reset –hard 版本号 #返回到指定的版本号

备注：HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。

穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。

要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

**工作区和暂存区：**

把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

备注：Git跟踪并管理的是修改，而非文件。

Git diff

用于比较两次修改的差异

**1.1 比较工作区与暂存区**

git diff 不加参数即默认比较工作区与暂存区

**1.2 比较暂存区与最新本地版本库（本地库中最近一次commit的内容）**

git diff --cached  [<path>...]

**1.3 比较工作区与最新本地版本库**

git diff HEAD [<path>...]  如果HEAD指向的是master分支，那么HEAD还可以换成master

**1.4 比较工作区与指定commit-id的差异**

git diff commit-id  [<path>...]

**1.5 比较暂存区与指定commit-id的差异**

git diff --cached [<commit-id>] [<path>...]

**1.6 比较两个commit-id之间的差异**

git diff [<commit-id>] [<commit-id>]

**1.7 使用git diff打补丁**

git diff > patch //patch的命名是随意的，不加其他参数时作用是当我们希望将我们本仓库工作区的修改拷贝一份到其他机器上使用，但是修改的文件比较多，拷贝量比较大，此时我们可以将修改的代码做成补丁，之后在其他机器上**对应目录下**使用 git apply patch 将补丁打上即可

git diff --cached > patch //是将我们暂存区与版本库的差异做成补丁

git diff --HEAD > patch //是将工作区与版本库的差异做成补丁

git diff Testfile > patch//将单个文件做成一个单独的补丁

**撤销修改：**git checkout -- readme.txt

撤销可以分为两种情况：

一种是对工作区修改的撤销：git checkout – file\_name

第二种是对已经提交到暂存区的修改的撤销：git reset HEAD file\_name; 这个操作把暂存区的修改撤销掉（unstage），重新放回工作区，然后就可以撤销工作区的修改了。

**备注：**命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是readme.txt已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次git commit或git add时的状态。

删除文件：

修改操作的删除：rm file\_name

在Git中，删除也是一个修改操作；在Git中，删除也是一个修改操作；而这时删除了文件，工作区和版本库就不一致了，因此git status会说明哪些文件被删除了。

从版本库删除一个文件后，有两个可能的选择：

1. 确实要从版本库删除，那么使用指令：

git rm file\_name

Git commit –m “注释”

1. 另一种是删错了，因为版本库里还有，所以可以很轻松的恢复到最新版本：

git checkout – test.txt

特别注意：git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

# 远程仓库

**先创建本地库，再创建远程库，然后将本地库关联远程库，并向远程库推送。**

## 添加远程仓库与推送

Git是分布式版本控制系统，同一个Git仓库，可以分布到不同的机器上。

首先在git上创建有对应的远程库；

然后将本地库链接到远程库：

git remote add origin [git@github.com:michaelliao/learngit.git](mailto:git@github.com:michaelliao/learngit.git)；

最后将本地库推送到远程库：

git push -u origin master

**备注**：远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

在第一次推送后，只要本地做了提交，就可以通过以下命令将本地master的最新修改推送至github：git push origin master

SSH警告

当你第一次使用Git的clone或者push命令连接GitHub时，会得到一个警告；这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。

## 克隆

知道远程库地址后可以通过如下指令实现克隆：

git clone git@github.com:michaelliao/gitskills.git

备注：GitHub给出的地址不止一个，还可以用https://github.com/michaelliao/gitskills.git这样的地址。实际上，Git支持多种协议，默认的git://使用ssh，但也可以使用https等其他协议。

使用https除了速度慢以外，还有个最大的麻烦是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。

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。

Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

# 分支管理

## 基本分支操作

创建分支，然后切换到dev分支

git checkout –b dev

备注： git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

git branch dev #创建分支

git checkout dev #切换到dev分支

git branch #用于查看当前的分支

合并分支

git merge dev

合并了之后就可以删除dev分支了：git branch –d dev

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>

备注：git merge命令用于合并指定分支到当前分支。合并分支时，要将当前分支切换到需要保留的分支上，merge后的分支名是需要丢掉的分支名

合并方式分为多种：

Fast-forward: 合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

建议：因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

## 解决冲突

用带参数的git log也可以看到分支的合并情况：git log --graph --pretty=oneline --abbrev-commit

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

用git log --graph命令可以看到分支合并图。

## 分支管理策略

通常，合并分支时，如果可能，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。

如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

**开发实践策略：**

在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，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；

那在哪干活呢？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分支上合并就可以了。

--no-ff方式的git merge 就是：在非Fast forward模式下merge：

git merge --no-ff -m "merge with no-ff" dev

**查看分支历史：**通过下面的指令可以查看分支历史，

git log --graph --pretty=oneline --abbrev-commit

## bug分支

**注意：当要切换到另一个分支时，当前分支的内容要么被提交到暂存库和版本库，要么使用git stash 储藏现场；总之必须要保存，否则无法切换分支。**

当在一个分支的工作进行到一半，还无法加入到暂存库和版本库时，出现突发状况，在某一个分支上有一个bug需要修复，那么需要保护当前分支上的修改，但又不能提交到暂存区和版本库，因此可以通过：git stash来储藏工作现场，等以后恢复现场。

首先储藏现场：

git stash #用于保存无法add和commit的当前状态（现场）

然后切换到右bug的分支：

git checkout master

创建bug分支：

git checkout –b issue-101

修改bug，并提交：

合并bug分支：

切换回原来的工作分支：

使用git stash pop 恢复现场

## Feature分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。

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。

创建一个新的功能分支：

git checkout –b feature-vulcan

添加功能，修改，提交：

修改后返回原来的分支：

然后可以合并将新功能添加要原来的分支；

如果没有合并，且不要新功能分支，由于机密还必须销毁，那么可以：

git branch -d feature-vulcan

以上指令会销毁失败。Git友情提醒，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用大写的-D参数。。

现在我们强行删除：git branch -D feature-vulcan

## 多人协作

**推送分支**

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

要查看远程库的信息，用git remote；用git remote -v显示更详细的信息。

推送分支：git push origin branch\_name

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

master分支是主分支，因此要时刻与远程同步；

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；

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；

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。

总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

**抓取分支**

多人协作时，大家都会往master和dev分支上推送各自的修改。

当从远程库clone时，默认情况下，只能看到本地的master分支。

现在，要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是用以下命令创建本地dev分支：

git checkout -b dev origin/dev

现在，就可以在dev上继续修改，然后，时不时地把dev分支push到远程

当伙伴修改了dev，你也修改了dev，小伙伴比你早推送dev，那么你再推送dev就会推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送。

直接git pull 也会失败，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：

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

**特别注意：多人协作的工作模式通常是这样**：

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

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；

如果合并有冲突，则解决冲突，并在本地提交；

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！

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>。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

**小结**

查看远程库信息，使用git remote -v；

本地新建的分支如果不推送到远程，对其他人就是不可见的；

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；

从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

## Rebase

在git pull后，也就是将本地的分支的修改和远程对应分支的修改合并

使用git rebase 就可以调整本地的提交历史线。

rebase操作可以把本地未push的分叉提交历史整理成直线；

rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比。