创建Git仓库：

在一个文件夹(可为空亦可非空)中执行：git init



将文件(可以是图片视频等文件，但Git只能跟踪文本文件的改动)添加到版本库中：

echo Git is version control system. >> Readme.txt

echo Git is free software. >> Readme.txt

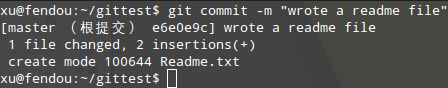
第一步：

$git add Readme.txt



第二步：

$git commit -m “wrote a readme file”



-m参数后面输入的是本次提交的说明。可以不加此参数。强烈建议添加。

为什么Git添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同文件，比如：

$git add file1.txt

$git add file2.txt file3.txt

$git commit -m “add 3 files.”

查看状态：

$git status



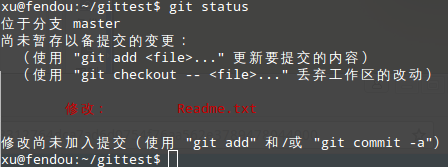
修改Readme.txt文件：

Git is a distributed version control system.

Git is free software.

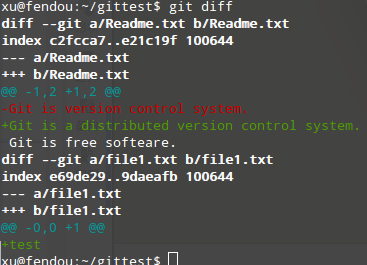
再次查看状态：

$git status



查看文件具体有哪些改动：

$git diff



提交修改后的文件：

$git add Readme.txt file1.txt

查看状态，确保提交无误：

$git status

提交：

$git commit -m “add distributed”

版本回退：

再次修改Readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

然后提交：

$ git add readme.txt

$ git commit -m "append GPL"

现在一共有个三个版本提交了。描述分别为：

1. wrote a feadme file

Git is a version control system.

Git is free software.

1. add distributed

Git is a distributed version control system.

Git is free software.

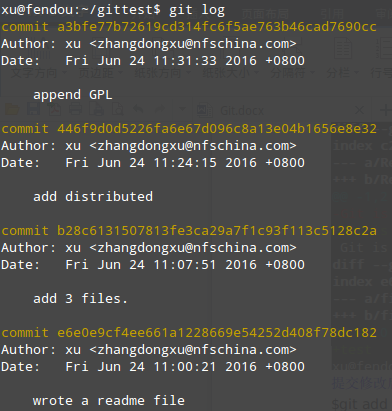
1. add 3 files
2. addend GPL

Git is a distributed version control system.

Git is free software distributed under the GPL.

查看更改历史记录：

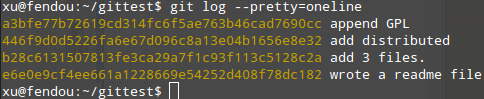
$git log



git log命令显示从最近到最远的提交日志。

更改git log输出信息显示类型：

$git log --pretty=oneline



类似a3befe....等字符串是commit id(版本号)。

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^^，可以用HEAD~n来表示HEAD之前第n个版本。

查看当前HEAD在哪里(我自己写的方法，官方是不是这样还没看到地方)：

$git reset --hard HEAD



回退到上一版本：

$git reset --hard HEAD^



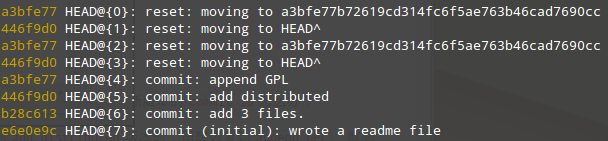
通过commit id回退到某一版本：

$git reset --hard a3bfe77b72619cd314fc6f5ae763b46cad7690cc

将HEAD从新指向commit id为a3bfe77b72619cd314fc6f5ae763b46cad7690cc的提交(即append GPL的地方)

查看操作记录(当回退版本之后找不到commit id时，可以用这个命令查看)：

$git reflog



Git工作区：

就是在哪个目录中执行的git init，则那个目录即为工作区。

版本库：

工作区有一个隐藏目录.git，这个不算是工作区，而是Git的版本库。

Git的版本库里存放了很多东西，其中最重要的就是称为stage(或者叫index)的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



把文件往Git版本库里添加的时候，是分两步执行的：

第一步：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：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就是王master分支上提交更改。

git add命令实际上就是把要提交的所有修改放到暂存区(stage)，然后执行git commit就可以一次性把暂存区的所有修改提交到分支。

管理修改：

Git跟踪并管理的是修改，而非文件。

实例理解：

修改readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

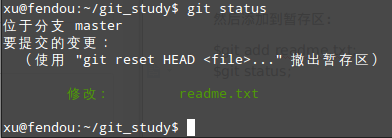
Git has a mutable index called stage.

Git tracks changes.

然后添加到暂存区：

$git add readme.txt;

$git status;



再次修改readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

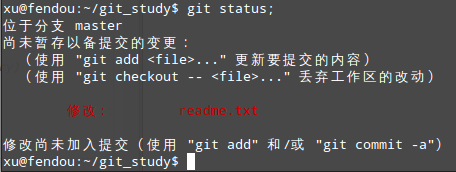
Git tracks changes of files.

提交：

$git commit -m “git tracks changes”

再次查看状态：

$git stauts



如此，可以看到第二次的修改没有被提交。

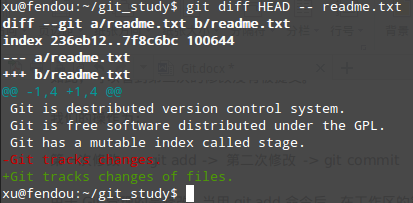
我们的操作为：

第一次修改 -> git add -> 第二次修改 -> git commit

由于Git管理的是修改，当用git add命令后，在工作区的第一次修改就被放入暂存区，准备提交，但是在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改提交了，第二次的修改并不会被提交。

可以查看工作区和版本库里面最新版本的区别：

$git diff HEAD -- readme.txt



可见，第二次修改的确没有被提交。

每次修改，如果不add到暂存区，那就不会加入到commit中。

撤销修改：

修改readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

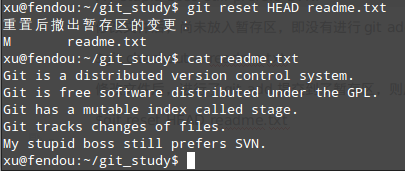
My stupid boss still prefers SVN.

修改文件后，尚未放入暂存区，即没有进行git add操作，可以直接修改文件或者执行：

$git checkout -- readme.txt

修改文件后，进行了git add提交到了暂存区，则应该使用：

$git reset HEAD readme.txt



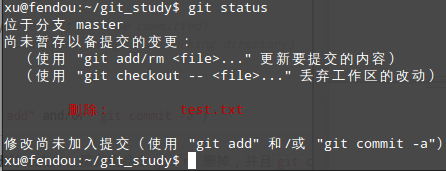
删除文件：

首先添加并提交一个测试文件test.txt，然后在本地工作区中删掉该文件：

$rm test.txt

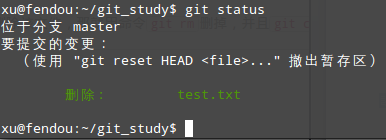
此时查看状态：

$git status



如果确实要从版本库中删除该文件，那么应该使用git rm命令并重新进行提交。

$git rm test.txt



$git commit -m “remove test.txt”

如果是错误删除本地工作区的内容，则可以从版本仓库中很轻松的把误删除的文件恢复到最新版本：

$git checkout -- test.txt

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，都可以还原。

远程仓库：

添加远程仓库：

$git remote add origin [git@github.com:xu910121/git.git](mailto:git@github.com:xu910121/git.git)

添加后，远程仓库的名字就是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。

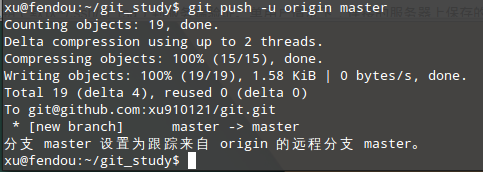
将本地库的所有内容推送到远程库上：

$git push -u origin 本地分支:远程分支。

远程分支可以没有，自动创建。

**删除远程分支：**

$git push origin :远程分支。即不填写本地分支进行push



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-u参数，Git不但会把本地的master分支内容推送到远程新的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。

从现在起，只要本地作了提交，就可以通过命令：

$git push origin master

把本地master分支的最新修改推送至Github。

从远程仓库克隆：

git clone

分支管理：

分支的作用：

假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

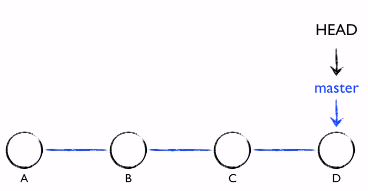
创建与合并分支：

在版本回退里，已经说过，每次提交，Git都把他们穿成一条时间线，这条时间线就是一个分支。在Git里，默认有一条分支，这个分支叫做主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



每次提交，master分支都会向前移动一步，这样，随着不断提交，master分支的线也就越来越长：



当我们创建新的分支，例如dev时，Git新建了一个指针叫做dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：



Git创建分支很快，因为除了增加一个dev指针，改变一下HEAD指向，工作区的文件都没有任何变化。



不过从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



如果我们在dev上的工作完成了，就可以把dev合并到master上，最简单的方法就是直接把master指向dev的当前提交，这样就完成了合并：



合并完成后，甚至可以删除dev分支。删除dev分支就是把dev指针给删除掉，删除掉后，就剩下一条master分支：



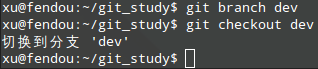
创建分支：

$git checkout -b dev

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：

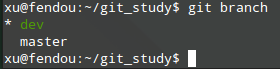
$git branch dev

$git checkout dev



查看当前分支：

$git branch



当前分支即为dev，可以在dev分支上正常提交，比如对readme.txt做一个修改，加上一行：

Creating a new branch is quick.

进行提交：

$git add readme.txt

$git commit -m “branch test”

切换分支：

现在，dev分支的工作完成，我们就可以切换回master分支：

$git checkout master

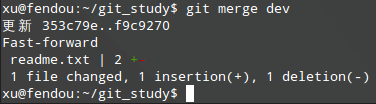
切换回master分支后，查看readme.txt可以看到刚才添加的内容不见了。因为那个提交实在dev分支上，而master分支此刻的提交点并没有变：



合并分支：

现在，我们可以把dev分支上的工作成果合并到master分支上：

$git merge dev



git merge命令用于合并指定的分支到当前分支！（所以合并时需要注意当前分支是哪个，可以使用git branch查看下再进行合并。）

Fast-forward信息表示这次合并是“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。但并不是每次合并都能Fast-forward。

删除分支：

$git branch -d dev



删除远程的分支：

git push origin(远程名) :远程分支名

冲突的解决：

准备新的feature1分支，继续我们的新分支开发：

$git branch feature1

$git checkout feature1

修改readme.txt最后一行为：

Create a new branch is quick AND simple.

在feature1分支进行提交：

$git add readme.txt

$git commit -m “AND simple”

切换到master分支

$git checkout master

在master分支上把readme.txt文件最后一行改为：

Create a new branch is quick & simple.

进行提交：

$git add readme.txt

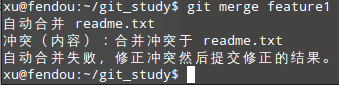
$git commit -m “&simple”

现在master分支和feature1分支各自都有新的提交，如下图：

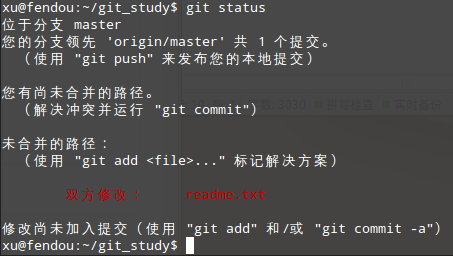


这种情况下，Git无法“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会冲突，我们试试看：

$git merge feature1



$git status



此时readme.txt内容已经变为：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

Creating a new branch is quick AND simple.

>>>>>>> feature1

Git用<<<<<,=====,>>>>>>标记出不同分支的内容，我们将readme.txt最后一行修改为如下后保存：

Creating a new branch is quick and simple.

进行提交

$git add readme.txt

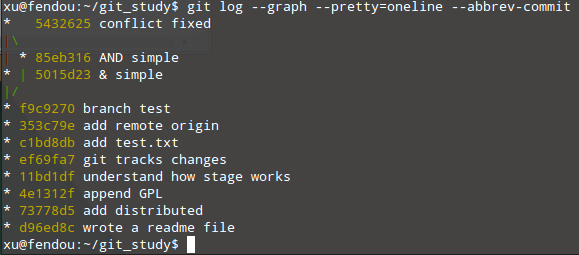
$git commit -m “conflict fixed”

现在，master分支和feature1分支变成了下图所示：



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



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。

禁用Fast forward模式：

在合并时添加--no-ff参数。

实例：

创建并切换dev分支：

$git branch -b dev

修改readme.txt文件，在最后一行添加如下：

Git can megre branches.

并提交：

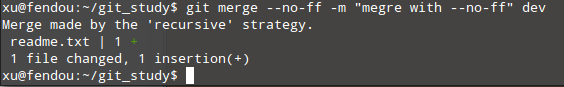
$git add readme.txt

$git commit -m “add merge”

切换回master分支：

$git checkout master

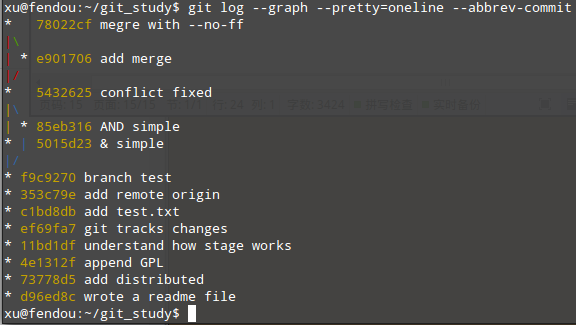
使用参数--no-ff来合并dev分支：



因为本次合并并要创建一个新的commit，所以加上-m参数，把commit的描述写进去。

合并后，我们用git log看看分支历史：

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



不使用Fast forward模式，megre后就像下图所示：



分支策略：

在实际开发中，我们应该按照几个基本原理进行分支管理：

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；

那么在哪干活呢？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；

每个人都在dev分之上干活，每个人都有自己的分支，是不是地往dev分支上合并就可以了。

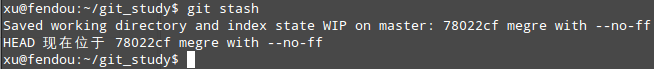
所以，团队合作的分支看起来就像这样；



保存工作现场：

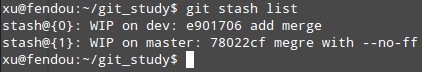
git stash命令可以将当前工作现场“储藏”起来，等以后恢复现场后继续工作。

$git stash



查看保存的工作现场：

$git stash list



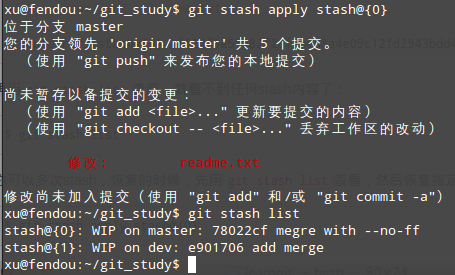
恢复工作现场：

从git stash list中可以看到保存的工作现场。

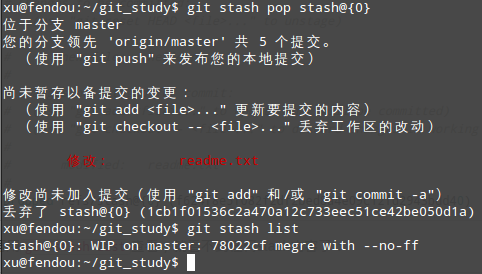
方法一：用git stash apply恢复，但是恢复后，stash内容并不会删除，需要使用git stash drop来进行删除。

方法二：用git stash pop，恢复的同时把stash内容也删除了。

$git stash apply stash@{0}

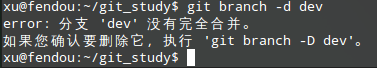


$git stash pop stash@{0}



强行删除分支：

如果dev（dev分支必须已经进行提交过内容，如果仅仅是创建了一个dev分支而没有任何内容的提交，则直接删除即可。）分支未被合并，执行git branch -d dev进行dev分支的删除时，将会无法删除，Git将会提示没有合并，如果删除，将会丢失修改。



如果确认该分支无需合并而且确实要删除，则执行：

$git branch -D dev

即可将未合并的分支删除。



多人协作：

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。

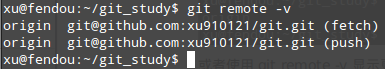
查看远程仓库信息：

$git remote



或者使用git remote -v显示更详细的信息：

$git remote -v



上面显示了可以抓取和推送的origin地址。如果没有推送权限，就看不到push的地址。

推送分支：

推送分支，就是把该分支上的所有本地提交推送到远程仓库。推送时要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

$git push origin master

如果推送到其他分支，比如dev上，就改成

$git pust origin dev

抓取分支（克隆分支）：

$git clone [git@github.com:xu910121/git.git](mailto:git@github.com:xu910121/git.git)

当从远程库中clone时默认情况下只能看到本地的master分支。如果要在dev分支上开发，就必须创建远程origin的dev分支到本地：

$git branch dev origin/dev



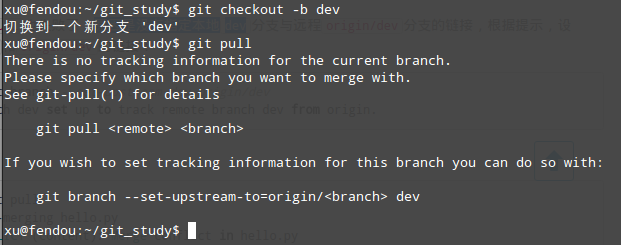
当有多人对远程同一文件进行提交修改时，将会出现提交冲突，



这时应该先用git pull把最新提交从origin/dev抓下来，解决冲突，再提交。

$git pull

如果git pull也失败了，是因为没有指定本地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



然后再进行pull操作：

$git pull



标签管理：

发布一个版本时，我们通常先在版本库中打一个标签，这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

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

创建标签：Git创建标签默认是打在最新提交的commit上的。

在Git中打标签非常容易，首先切换到需要打标签的分支上，然后创建标签：

$git branch

$git checkout master

$git tag v1.0 -m “tag test” //(v1.0表示标签的名称，可以自己指定 -m参数可以不要)

在某次提交上创建标签：

$git tag tag\_name commit\_id

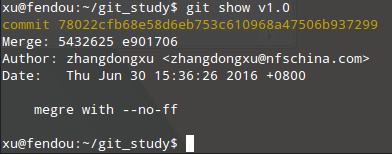
查看所有标签：

$git tag

标签不是按照时间排序的，而是按字母顺序排的。

查看某一个标签的具体信息：

$git show tag\_name



标签签名：

使用-s参数用私钥签名一个标签：

$git tag -s tag-name -m “此处可以填写标签的描述” commit\_id //这样只能为新创建的标签签名

操作标签：

删除标签：

如果标签打错了，可以进行删除操作：

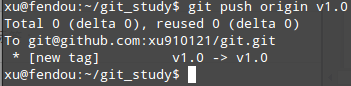
$git tag -d v1.0

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

远程推送标签：

将本地标签推送到远程：

$git push origin v1.0



或者一次性推送全部尚未推送到远程的本地标签：

$git push origin --tags



删除推送到远程的标签：

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

$git tag -d v1.0

然后，从远程删除。删除命令也是push，格式如下：

$git push origin :refs/tags/v1.0

