## linux下安装git

sudo apt-get install git

安装完成后，还需要最后一步设置，在命令行输入：

git config --global user.name "Your Name"

git config --global user.email "email@example.com"

git config命令的--global参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址

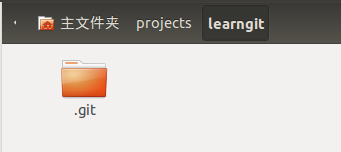
## 创建版本库

版本库（仓库）可以被理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

创建一个名为learngit的仓库

只需要在learngit目录下执行 git init 命令。（在哪个目录下执行git init 则会为这个目录创建一个仓库）。----这里创建的只是一个本地仓库，还没有与远程仓库进行关联呢。

在learngit目录下执行git init 命令后，会在当前目录下生成.git的目录，这个目录是Git来跟踪管理版本库的。(这个.git目录是隐藏的，可以通过ls –ha查看或者ctrl+h在GUI界面下查看)



## 把文件添加到版本库

**第一步：用git add命令将文件readme.txt添加到版本库**

git add readme.txt

**第二步：用命令git commit告诉Git，把文件提交到仓库(这里只是添加到了本地仓库中)**

git **commit** -m "对本次提交的说明"

注意：实际上git add命令只是将文件（文件的修改）添加到了版本库的暂存区，而git commit命令则是将暂存区中的所有文件（文件的修改）提交给分支。所以可以执行多次git add命令后，再执行一次git commit 命令统一提交。--关于暂存区和分支将在后面说明。

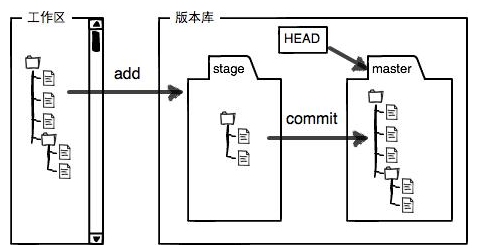
Git跟踪并管理的是修改，而非文件。

## 工作区与版本库

上面我们建立了一个learngit仓库。learngit目录现在就是我们的工作区。

learngit目录下的.git目录就是一个版本库。

Git的版本库里存了很多东西，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。



前面讲了我们把文件往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就是往master分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

## 返回上某一个版本

git reset命令可以使我们回退到某一个版本。

HEAD：在Git中，用HEAD表示当前版本，也就是最新的提交版本。

HEAD^：表示上一个版本

HEAD^^：表示上上个版本

HEAD~n：表示前n个版本(n=1,23,…)

回退到上一个版本：git reset --hard HEAD^

--hard参数的作用将在后面讲。

根据版本号直接回退到某一个版本：git re**set** --hard 版本号

版本号可以通过git log命令看到

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD从指向你回退到的版本：





对版本跳转的总结

* 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。

## 撤销修改

当你不小心在工作区进行了一些错误的修改，该如何撤消这些错误的修改呢？

情况0.

对于一些不涉及到版本的错误直接修改即可。

情况1：

当我们想要废弃工作区的修改时，执行git checkout -- file可以丢弃工作区的修改，使工作区（或工作区中的某些文件）恢复到上一次git add或git commit时的状态。---如果你的修改中既有“正确“的也有”错误“的，这时不管“正确“的还是有”错误“的都将被被撤消。

git checkout -- readme.txt

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修改后还没有被放到暂存区（没有执行 git add），那么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（在执行git add后），又作了修改，那么撤销修改就回到添加到暂存区时的状态。

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

注意：git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到git checkout命令。

情况2.

你对工作区的文件的修改已经添加到版本库的暂存区时（执行了git add），但还没有提交到版本库时（没有git commit）;为了下次git commit时不提交错误的修改，需要将添加到暂存区的错误修改清理掉，同时将工作区退回上一个版本。

git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。

执行git checkout -- file可以丢弃工作区的修改，使工作区（或工作区中的某些文件）恢复到上一次git add或git commit时的状态。

情况3.

如果你对工作区的修改已经提交到了版本库（执行了git commit）。这时需要进行[版本回退](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)操作。

情况4.

如果修改推送到了远程版本库，那么就真的“错”了。

小结：

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](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)一节，不过前提是没有推送到远程库。

## 删除文件

1. 要删除的文件在工作区中，但它不在版本库中（没有对它执行过git add，git commit命令）。直接用rm 命令删除即可。
2. 要删除的文件在版本库中。则需要进行三步：

rm filename将文件删除。

这时Git知道你删除了文件，因此，工作区和版本库就不一致了。git status命令会立刻告诉你哪些文件被删除了

用git rm命令从版本库中删除该文件

git commit

小结

命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失**最近一次提交后你修改的内容**。

## 关联本地仓库与远程仓库(github)

### 情况1：本地同步到远端

你已经在本地创建了一个Git仓库后，又想在远程端创建一个Git仓库，并且让这两个仓库进行远程同步，这样，远程端上的仓库既可以作为备份，又可以让其他人通过该仓库来协作。

**步骤1：在远程端创建一个Git仓库**。

假设建立的仓库名为learngit。

**步骤2：在本地执行git remote add命令**：

git remote add origin [git@github.com:jiaGuYuan/learngit.git（jiaGuYuan](mailto:git@github.com:jiaGuYuan/learngit.git（jiaGuYuan)是我的GitHub用户名，learngit是我在GitHub上的一个仓库）

格式 git remote add 远程库名字 远程主机名@远程主机IP:路径/git仓库名

**步骤3：把本地库的所有内容推送到远程库上**

git push -u origin master

格式：git push -u远程库名字 分支名

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。

小结

要关联一个远程库，使用命令git remote add origin git@server-name:path/repo-name.git；

关联后，使用命令git push -u origin master第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令git push origin master推送最新修改；

### 情况2：远程克隆到本地

在远程端有一个Git仓库，你希望将这个远程的仓库克隆到本地（从远程库克隆）。

git clone命令克隆一个本地库。

git clone git@github.com:jiaGuYuan/learngit.git

小结

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

## 创建与合并分支

概念：

在Git里，这个分支叫主分支，即master分支。HEAD指向的就是当前（正在操作的）分支。

在Git中一开始的时候，只有一个master分支（即主分支），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：



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



Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，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



假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：



创建dev分支，然后切换到dev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 创建分支

git checkout dev 切换分支

git命令针对的是当前正在操作的分支。所以要在某个分支上进行操作需要先切换到对应的分支才行。

把dev分支的工作成果合并到master分支上：（首先要保证当前的处于master分支，然后才能合并到master分支上）

git checkout master 切换到master分支

git merge dev 将dev分支合并到当前工作分支

删除dev分支了：

git branch -d dev

小结

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>

## 解决冲突

如果master分支和feature1分支都修改了某个文件（比如readme.txt），并且都有新的提交，变成了这样：



这种情况下，Git无法执行“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有冲突，我们试试看：

git merge feature1

提示错误信息：

Auto-merging readme.txt

CONFLICT (content): Merge conflict **in** readme.txt 在readme.txt文件中存在合并冲突

Automatic merge failed; fix conflicts **and** **then** commit the result.

这时打开readme.txt文件，Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容。这时必须手动解决冲突后再提交。

修改冲突后master分支和feature1分支变成了下图所示：



## 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。

--no-ff方式的git merge：以非Fast forward模式合并分支。

将dev分支以非Fast forward模式合并到当前分支， --no-ff参数，表示禁用Fast forward：

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

因为本次合并要创建一个新的commit，所以加上-m参数，把commit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

\* 7825a50 merge **with** no-ff

|\

| \* 6224937 “显示在dev分支中的提交说明”

|/

\* 59bc1cb conflict fixed

...

可以看到，不使用Fast forward模式，merge后就像这样：



## 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：

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。

所以，团队合作的分支看起来就像这样：



**小结**

Git分支十分强大，在团队开发中应该充分应用。

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。

### Bug分支

在Git中，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。

当你接到修复一个代号101的bug的任务时，你想创建一个分支issue-101来修复它，但是，当前正在dev上进行的工作还没有提交：并不是你不想提交，而是工作只进行到一半，还没法提交。

Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作.

首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：

git checkout master 切换到要修复bug的分支

git checkout -b issue-101 在要修复bug的分支上创建一个执行修复操作的issue-101分支，并切换到这个分支

修复完成后提交，切换到master分支，并完成合并，最后删除issue-101分支。

恢复工作现场：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；

另一种方式是用git stash pop，恢复的同时把stash内容也删了：

你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：

### Feature分支

git stash apply stash@{0}

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过git branch -D <name>强行删除

## 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。

要查看远程库的信息，用git remote

用git remote -v显示更详细的信息：显示可以抓取和推送的origin的地址。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

如果要推送其他分支，比如dev，就改成：

git push origin dev

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。

当你的小伙伴从远程库clone时git clone [git@github.com:jiaGuYuan/learngit.git](mailto:git@github.com:jiaGuYuan/learngit.git)。 默认情况下，你的小伙伴只能看到本地的master分支。

你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：git checkout -b dev origin/dev。

如果你和你的小伙伴都将修改推送到dev分支，那么将发生推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突。要解决这个问题需要先用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。

指定本地dev分支与远程origin/dev分支的链接：git branch --set-upstream dev origin/dev

### 多人协作的工作模式通常是这样：

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！

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 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。

## 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（tag），这样，就唯一确定了打标签时刻的版本。tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

#### 创建标签

在Git中打标签非常简单，首先，切换到需要打标签的分支上。然后，敲命令git tag <tagname>就可以打一个新标签。

默认标签是打在最新提交的commit上的。

如果要对不是最新提交的commit打标签，git tag <tagname> commit id

commit id可以通过git log查看到

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

git tag -a <tagname> -m "标签说明" commit id

**小结**

* 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。

## 操作标签

如果标签打错了，也可以删除：git tag -d <tagname>

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。

如果要推送某个标签到远程，使用命令git push origin <tagname>

一次性推送全部尚未推送到远程的本地标签：git push origin --tags

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 <tagname>

然后，从远程删除。删除命令也是push，但是格式如下：

git push origin :refs/tags/v0.9<tagname>

**小结**

* 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。

### 查看信息

git

直接输入git将提示git的使用信息。

git status

查看仓库的状态，它会告诉我们什么文件被修改了，什么文件修改了但是还没有提交，什么文件是刚添加进来的等等。它还会提示你接下来可以进行一些什么操作。

git diff

查看difference,它能告诉我们我们在某个文件上所做的修改。

git diff readme.txt ---将显示我们在readme.txt上所做的修改

git log

查看历史记录，它能告诉我们从最近到最远的提交日志。

--pretty=oneline参数：简化显示的内容。

git log命令显示的内容有commit id（版本号）、作者、日期、及该版本的修改。

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

git reflog

git reflog命令能够查看我们执行的每一次命令。

git branch

查看分支信息

git stash list

查看工作现场的信息

git remote

查看远程库的信息

-v显示更详细的信息

git tag

查看所有标签

git show <tagname>

查看标签信息

信息：

Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把一个分支指向另一个分支的当前提交，所以合并速度非常快。

## 远程仓库

实际情况往往是这样，找一台电脑充当服务器的角色，每天24小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

GitHub:

注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看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Shell（Windows下打开Git Bash），创建SSH Key：

ssh-keygen -t rsa -C "youremail@example.com"

id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人.

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，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：



点“Add Key”，你就应该看到已经添加的Key：



### Git的一些配置

user.name和user.email的配置

git config --global user.name "Your Name"

git config --global user.email "email@example.com"

color配置

让Git显示颜色，会让命令输出看起来更醒目：

$ git config --global color.ui true

### 忽略特殊文件

有些时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件啦，等等，每次git status都会显示Untracked files ...，有强迫症的童鞋心里肯定不爽。

在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

最后就是把.gitignore也提交到Git。

忽略文件的原则是：

1. 忽略操作系统自动生成的文件，比如缩略图等；
2. 忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
3. 忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

不需要从头写.gitignore文件，GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>。

有些时候，你想添加一个文件到Git，但发现添加不了，原因是这个文件被.gitignore忽略了：

git add App.class **添加**App.**class到版本库**

**添加失败，输出提示信息如下：**

The following paths are ignored by one of your .gitignore files:

App.class

Use -f if you really want to add them.

-f强制添加到Git

git add –f App.class

或者你发现，可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用git check-ignore命令检查：

git check-ignore -v App.class

.gitignore:3:\*.class App.class

Git会告诉我们，.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

**小结**

* 忽略某些文件时，需要编写.gitignore；
* .gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理

### 配置别名

有没有经常敲错命令？比如git status？status这个单词真心不好记。

如果敲git st就表示git status那就简单多了，当然这种偷懒的办法我们是极力赞成的。

我们只需要敲一行命令，告诉Git，以后st就表示status：

git config --global alias.st status

当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch：

git config --global **alias**.co checkout

git config --global **alias**.ci commit

git config --global **alias**.br branch

以后提交就可以简写成：

git ci -m "bala bala bala..."

--global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

在[撤销修改](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001374831943254ee90db11b13d4ba9a73b9047f4fb968d000)一节中，我们知道，命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：

$ git config --global **alias**.unstage 'reset HEAD'

当你敲入命令：

git unstage test.py

实际上Git执行的是：

git re**set** HEAD test.py

配置一个git last，让其显示最后一次提交信息：

git config --global **alias**.last 'log -1'

这样，用git last就能显示最近一次的提交：

git last

commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2

Merge: bd6ae48 291bea8

Author: Michael Liao <askxuefeng@gmail.com>

Date: Thu Aug 22 22:49:22 2013 +0800

merge & fix hello.py

甚至还有人丧心病狂地把lg配置成了：

git config --global **alias**.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

来看看git lg的效果：



**配置文件**

配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中。而当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中。

### 搭建Git服务器

在[远程仓库](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001374385852170d9c7adf13c30429b9660d0eb689dd43a000)一节中，我们讲了远程仓库实际上和本地仓库没啥不同，纯粹为了7x24小时开机并交换大家的修改。

GitHub就是一个免费托管开源代码的远程仓库。但是对于某些视源代码如生命的商业公司来说，既不想公开源代码，又舍不得给GitHub交保护费，那就只能自己搭建一台Git服务器作为私有仓库使用。

搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的apt命令就可以完成安装。

假设你已经有sudo权限的用户账号，下面，正式开始安装。

第一步，安装git：

$ su**do** apt-**get** install git

第二步，创建一个git用户，用来运行git服务：

$ su**do** adduser git

第三步，创建证书登录：

收集所有需要登录的用户的公钥，就是他们自己的id\_rsa.pub文件，把所有公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

第四步，初始化Git仓库：

先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入命令：

$ su**do** git init --bare sample.git

Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把owner改为git：

$ su**do** chown -R git:git sample.git

第五步，禁用shell登录：

出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑/etc/passwd文件完成。找到类似下面的一行：

git:x:1001:1001:,,,:/home/git:/bin/bash

改为：

git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

第六步，克隆远程仓库：

现在，可以通过git clone命令克隆远程仓库了，在各自的电脑上运行：

$ git clone git@server:/srv/sample.git

Cloning into 'sample'...

warning: You appear to have cloned an empty repository.

剩下的推送就简单了。

管理公钥

如果团队很小，把每个人的公钥收集起来放到服务器的/home/git/.ssh/authorized\_keys文件里就是可行的。如果团队有几百号人，就没法这么玩了，这时，可以用[Gitosis](https://github.com/res0nat0r/gitosis)来管理公钥。

这里我们不介绍怎么玩[Gitosis](https://github.com/res0nat0r/gitosis)了，几百号人的团队基本都在500强了，相信找个高水平的Linux管理员问题不大。

管理权限

有很多不但视源代码如生命，而且视员工为窃贼的公司，会在版本控制系统里设置一套完善的权限控制，每个人是否有读写权限会精确到每个分支甚至每个目录下。因为Git是为Linux源代码托管而开发的，所以Git也继承了开源社区的精神，不支持权限控制。不过，因为Git支持钩子（hook），所以，可以在服务器端编写一系列脚本来控制提交等操作，达到权限控制的目的。[Gitolite](https://github.com/sitaramc/gitolite)就是这个工具。

这里我们也不介绍[Gitolite](https://github.com/sitaramc/gitolite)了，不要把有限的生命浪费到权限斗争中。

小结

* 搭建Git服务器非常简单，通常10分钟即可完成；
* 要方便管理公钥，用[Gitosis](https://github.com/sitaramc/gitolite)；
* 要像SVN那样变态地控制权限，用[Gitolite](https://github.com/sitaramc/gitolite)。

## git常用命令

把文件添加到版本库

第一步：用git add命令将文件readme.txt添加到版本库

git add readme.txt

第二步：用命令git commit告诉Git，把文件提交到仓库

git commit -m "对本次提交的说明"

回退到上一个版本：git reset --hard HEAD^

--hard参数的作用将在后面讲。

根据版本号直接回退到某一个版本：git reset --hard 版本号

版本号可以通过git log命令看到

git checkout -- file可以丢弃工作区的修改，使工作区（或工作区中的某些文件）恢复到上一次git add或git commit时的状态。

注意：git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到git checkout命令。

git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。

执行git checkout -- file可以丢弃工作区的修改，使工作区（或工作区中的某些文件）恢复到上一次git add或git commit时的状态。

git rm命令从版本库中删除该文件

关联本地仓库与远程仓库

git remote add 远程库名字 远程主机名@远程主机IP:路径/git仓库名

把本地库的所有内容推送到远程库上

git push -u origin master

格式：git push -u远程库名字 分支名

将这个远程的仓库克隆到本地（从远程库克隆）。

git clone命令克隆一个本地库。

git clone git@github.com:jiaGuYuan/learngit.git

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>

Git分支十分强大，在团队开发中应该充分应用。

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。

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。

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。