1、创建版本库

创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

$ mkdir learngit

$ cd learngit

$ pwd

/Users/michael/learngit

或者

第二步，通过git init命令把这个目录变成Git可以管理的仓库：

$ git init

Initialized empty Git repository in /Users/michael/learngit/.git/

可以发现当前目录下多了一个.git的目录，如果你没有看到.git目录，那是因为这个目录默认是隐藏的

，用ls -ah命令就可以看见。

2、添加文件到Git仓库，分两步：

比如有个文件叫 readme.txt

第一步，使用命令 $ git add readme.txt；

可一次性添加多个（git **add** -u）

$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "表识名"

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

执行上面的命令，没有任何显示，这就对了，说明添加成功。

第二步，用命令git commit告诉Git，把文件提交到仓库：

$ git commit -m "这里随便写方便自己识别"

3、修改文件 查看状态

git status命令可以让我们时刻掌握仓库当前的状态

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")

上面的命令告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

$ git diff 能看看具体修改了什么内容

知道了对readme.txt作了什么修改后，提交修改和提交新文件是一样的两步，第一步是git add：

$ git add readme.txt

同样没有任何输出。在执行第二步git commit之前，我们再运行git status看看当前仓库的状态：

$ git status

git status告诉我们，将要被提交的修改包括readme.txt，下一步，就可以放心地提交了：

$ git commit -m "标识名"

4、版本回退和前进

在Git中，我们用git log命令查看历史记录

$ git log

$ git log --pretty=oneline（可以简化历史记录显示）

$ git reset --hard HEAD^（回退一个版本）

$ git reset --hard HEAD~100（回退100个版本）

$ git reset --hard 3628164（前进到commit为3628164....这个版本，版本号没必要写全，前几位就可

以了）\*\*\*\* 只要上面的命令行窗口还没有被关掉！！！！

5、工作区和暂存区

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

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。

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。

6、撤销修改

$ 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。

用命令git reset HEAD file可以把暂存区的修改撤销掉

7、删除和误删恢复

用rm命令删了：

$ rm test.txt

git status命令会立刻告诉你哪些文件被删除了：

从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

$ git rm test.txt

rm 'test.txt'

$ git commit -m "remove test.txt"

[master d17efd8] remove test.txt

1 file changed, 1 deletion(-)

delete mode 100644 test.txt

现在，文件就从版本库中被删除了。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

$ git checkout -- test.txt

8、添加远程库

创建repository（仓库）点击Repositories 创建一个仓库 仓库名和本地磁盘创建的文件夹名一

致

（生成密匙）

先点击 settings 再点击 SSH and GPG keys 再点击 New SSH key 就会出现title框、key框——前者

是随便填的，后者需要我们生成

怎么生成这个key？回到git bash，输入ssh-keygen -t rsa -C "注册时的邮箱（791747453@qq.com）"

这时会让你输入保存密钥文件的路径，注意现在cd到哪个目录了，如果现在是z盘下面的git文件夹，输

入z: 其实还是会保存到z:\git，也就是当前所在文件夹

还要输入密码，输一个能记住的就行了。然后用记事本打开生成的pub文件，将里面的东西就是需要的

key，粘贴到网页上面，点add key

…or push an existing repository from the command line

按照网页提示在命令行输入：

（1）git remote add origin https://github.com/yyqsxr/hehhehe.git（添加后，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）

如果这一步出现常见报错 fatal: remote origin already exists.

解决办法如下：

1、先输入$ git remote rm origin

2、再输入$ git remote add origin git@github.com:djqiang/gitdemo.git 就不会报错了！

3、如果输入$ git remote rm origin 还是报错的话，error: Could not remove config section

'remote.origin'. 我们需要修改gitconfig文件的内容

4、找到你的github的安装路径，我的是C:\Users\ASUS\AppData\Local\GitHub

\PortableGit\_ca477551eeb4aea0e4ae9fcd3358bd96720bb5c8\etc

5、找到一个名为gitconfig的文件，打开它把里面的[remote "origin"]那一行删掉就好了！

如果输入$ ssh -T git@github.com

出现错误提示：Permission denied (publickey).因为新生成的key不能加入ssh就会导致连接不上

github。

解决办法如下：

1、先输入$ ssh-agent，再输入$ ssh-add ~/.ssh/id\_key，这样就可以了。

2、如果还是不行的话，输入ssh-add ~/.ssh/id\_key 命令后出现报错Could not open a

connection to your authentication agent.解决方法是key用Git Gui的ssh工具生成，这样生成的时候

key就直接保存在ssh中了，不需要再ssh-add命令加入了，其它的user，token等配置都用命令行来做。

3、最好检查一下在你复制id\_rsa.pub文件的内容时有没有产生多余的空格或空行，有些编辑器会帮

你添加这些的。

如果输入$ git push origin master

提示出错信息：error:failed to push som refs to .......

解决办法如下：

1、先输入$ git pull origin master //先把远程服务器github上面的文件拉下来

2、再输入$ git push origin master

3、如果出现报错 fatal: Couldn't find remote ref master或者fatal: 'origin' does not

appear to be a git repository以及fatal: Could not read from remote repository.

4、则需要重新输入$ git remote add origin git@github.com:djqiang/gitdemo.git

使用git在本地创建一个项目的过程

$ makdir ~/hello-world //创建一个项目hello-world

$ cd ~/hello-world //打开这个项目

$ git init //初始化

$ touch README

$ git add README //更新README文件

$ git commit -m 'first commit' //提交更新，并注释信息“first commit”

$ git remote add origin git@github.com:defnngj/hello-world.git //连接远程github项目

$ git push -u origin master //将本地项目更新到github项目上去

（2）git push -u origin master（提交文件到远程，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。）

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：

The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.

RSA key fingerprint is xx.xx.xx.xx.xx.

Are you sure you want to continue connecting (yes/no)?

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。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

Warning: Permanently added 'github.com' (RSA) to the list of known hosts.

这个警告只会出现一次，后面的操作就不会有任何警告了。

9、从远程库克隆

首先，登陆GitHub，创建一个新的仓库，名字叫gitskills：

我们勾选Initialize this repository with a README，这样GitHub会自动为我们创建一个README.md文

件。创建完毕后，可以看到README.md文件：

现在，远程库已经准备好了，下一步是用命令git clone克隆一个本地库：

$ git clone https://github.com/yyqsxr/yuanku.git

$ cd yuanku

$ ls

README.md

#### 10、创建与合并分支

在[版本回退](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，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，随着你不断提交，master分支的线也越来越长：

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：



你看，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：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

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 devSwitched to a new branch '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 devSwitched to branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行：

Creating a **new** branch is quick.

然后提交：

$ git add readme.txt $ git commit -m "branch test"

[dev fec145a] branch test

1 file changed, 1 insertion(+)

现在，dev分支的工作完成，我们就可以切换回master分支：

$ git checkout masterSwitched to branch 'master'

切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：

现在，我们把dev分支的工作成果合并到master分支上：

$ git merge devUpdating d17efd8..fec145aFast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，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。

当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。

合并完成后，就可以放心地删除dev分支了：

$ git branch -d devDeleted branch dev (was fec145a).

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master

11、解决冲突

创建一个新分支 feature1

$ git checkout -b feature1Switched to a new branch 'feature1'

修改文件readme.txt

提交：

$ git add readme.txt $ git commit -m "& simple"

[master 400b400] & simple

1 file changed, 1 insertion(+), 1 deletion(-)

切换到master分支：

$ git checkout master

再修改readme.txt提交

$ git add readme.txt $ git commit -m "& simple"

[master 400b400] & simple

1 file changed, 1 insertion(+), 1 deletion(-)

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 feature1Auto-merging readme.txtCONFLICT (content): Merge conflict **in** readme.txtAutomatic merge failed; fix conflicts **and** **then** commit the result.

果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：

$ git status*# On branch master# Your branch is ahead of 'origin/master' by 2 commits.## Unmerged paths:# (use "git add/rm <file>..." as appropriate to mark resolution)## both modified: readme.txt#*

no changes added to commit (use "git add" **and**/**or** "git commit -a")

点开readme.txt会看到冲突部分

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.<<<<<<< HEADCreating 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，我们修改如下后保存：

$ git add readme.txt $ git commit -m "conflict fixed"

[master 59bc1cb] 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

\* 59bc1cb conflict fixed

|\

| \* 75a857c **AND** simple

\* | 400b400 & simple

|/

\* fec145a branch test

...

最后，删除feature1分支：

$ git branch -d feature1Deleted branch feature1 (was 75a857c).

1. 分支管理策略

首先，仍然创建并切换dev分支：

$ git checkout -b devSwitched to a new branch 'dev'

修改readme.txt文件，并提交一个新的commit：

$ git add readme.txt

$ git **commit** -m "add merge"

[dev 6224937] **add** merge

1 file changed, 1 insertion(+)

现在，我们切换回master：

$ git checkout masterSwitched to branch 'master'

准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：

$ git merge --no-ff -m "merge with no-ff" devMerge made by the 'recursive' strategy.

readme.txt | 1 +

1 file changed, 1 insertion(+)

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 add merge

|/

\* 59bc1cb conflict fixed

...

### 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。

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。

13、bug分支

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，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。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交：

$ git status*# On branch dev# Changes to be committed:# (use "git reset HEAD <file>..." to unstage)## new file: hello.py## Changes not staged for commit:# (use "git add <file>..." to update what will be committed)# (use "git checkout -- <file>..." to discard changes in working directory)## modified: readme.txt#*

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？

幸好，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：

$ git stashSaved working directory **and** index state WIP on dev: 6224937 add mergeHEAD is now at 6224937 add merge

现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

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 masterSwitched to branch 'master'Your branch is ahead of 'origin/master' by 6 commits.$ git checkout -b issue-101Switched to a new branch 'issue-101'

现在修复bug，需要把“Git is free software ...”改为“Git is a free software ...”，然后提交：

$ git add readme.txt $ git commit -m "fix bug 101"

[issue-101 cc17032] fix bug 101

1 file changed, 1 insertion(+), 1 deletion(-)

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

$ git checkout masterSwitched to branch 'master'Your branch is ahead of 'origin/master' by 2 commits.$ git merge --no-ff -m "merged bug fix 101" issue-101Merge made by the 'recursive' strategy.

readme.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)$ git branch -d issue-101Deleted branch issue-101 (was cc17032).

太棒了，原计划两个小时的bug修复只花了5分钟！现在，是时候接着回到dev分支干活了！

$ git checkout devSwitched to branch 'dev'$ git status*# On branch dev*

nothing to commit (working directory clean)

工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

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：

$ git stash pop*# On branch dev# Changes to be committed:# (use "git reset HEAD <file>..." to unstage)## new file: hello.py## Changes not staged for commit:# (use "git add <file>..." to update what will be committed)# (use "git checkout -- <file>..." to discard changes in working directory)## modified: readme.txt#*Dropped refs/stash@{0} (f624f8e5f082f2df2bed8a4e09c12fd2943bdd40)

再用git stash list查看，就看不到任何stash内容了：

$ git stash list

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：

$ git stash apply stash@{0}

1. 新功能分支

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。

现在，你终于接到了一个新任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船。

于是准备开发：

$ git checkout -b feature-vulcanSwitched to a new branch 'feature-vulcan'

5分钟后，开发完毕：

$ git add vulcan.py$ git status*# On branch feature-vulcan# Changes to be committed:# (use "git reset HEAD <file>..." to unstage)## new file: vulcan.py#*$ git commit -m "add feature vulcan"

[feature-vulcan 756d4af] add feature vulcan

1 file changed, 2 insertions(+)

create mode 100644 vulcan.py

切回dev，准备合并：

$ git checkout dev

一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。

但是，

就在此时，接到上级命令，因经费不足，新功能必须取消！

虽然白干了，但是这个分支还是必须就地销毁：

$ git branch -d feature-vulcanerror: The branch 'feature-vulcan' is **not** fully merged.If you are sure you want to delete it, run 'git branch -D feature-vulcan'.

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，需要使用命令git branch -D feature-vulcan。

现在我们强行删除：

$ git branch -D feature-vulcanDeleted branch feature-vulcan (was 756d4af).

终于删除成功！

1. 多人开发

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：

$ git remote

origin

或者，用git remote -v显示更详细的信息：

$ git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

上面显示了可以抓取和推送的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。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

$ git clone git@github.com:michaelliao/learngit.gitCloning into 'learngit'...remote: Counting objects: 46, done.remote: Compressing objects: 100% (26/26), done.remote: Total 46 (delta 16), reused 45 (delta 15)Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.Resolving deltas: 100% (16/16), done.

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：

$ git branch

\* master

现在，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

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

$ git **commit** -m "add /usr/bin/env"

[dev 291bea8] **add** /usr/bin/env

1 file changed, 1 insertion(+)

$ git push origin dev

Counting objects: 5, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 349 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)**To** git@github.com:michaelliao/learngit.git

fc38031..291bea8 dev -> dev

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

$ git add hello.py

$ git **commit** -m "add coding: utf-8"

[dev bd6ae48] **add** coding: utf-8

1 file changed, 1 insertion(+)

$ git push origin dev**To** git@github.com:michaelliao/learngit.git

! [rejected] dev -> dev (non-fast-forward)

error: failed **to** push **some** refs **to** 'git@github.com:michaelliao/learngit.git'

hint: Updates were rejected because the tip **of** your **current** branch **is** behind

hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

hint: before pushing again.

hint: See the 'Note about fast-forwards' **in** 'git push --help' **for** details.

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：

$ git pull

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From github.com:michaelliao/learngit

fc38031..291bea8 dev -> origin/dev

There is no tracking information **for** the current branch.

Please specify which branch you want to merge **with**.

See git-pull(1) **for** details

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream dev origin/<branch>

git pull也失败了，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 dev origin/dev*

Branch dev **set** up **to** track remote branch dev **from** origin.

再pull：

$ git pullAuto-merging hello.pyCONFLICT (content): Merge conflict **in** hello.pyAutomatic merge failed; fix conflicts **and** **then** commit the result.

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的[解决冲突](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001375840202368c74be33fbd884e71b570f2cc3c0d1dcf000" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)完全一样。解决后，提交，再push：

$ git **commit** -m "merge & fix hello.py"

[dev adca45d] merge & fix hello.py

$ git push origin dev

Counting objects: 10, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (6/6), 747 bytes, done.

Total 6 (delta 0), reused 0 (delta 0)**To** git@github.com:michaelliao/learngit.git

291bea8..adca45d dev -> dev

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：

$ git remote

origin

或者，用git remote -v显示更详细的信息：

$ git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

上面显示了可以抓取和推送的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。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

$ git clone git@github.com:michaelliao/learngit.gitCloning into 'learngit'...remote: Counting objects: 46, done.remote: Compressing objects: 100% (26/26), done.remote: Total 46 (delta 16), reused 45 (delta 15)Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.Resolving deltas: 100% (16/16), done.

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：

$ git branch

\* master

现在，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

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

$ git **commit** -m "add /usr/bin/env"

[dev 291bea8] **add** /usr/bin/env

1 file changed, 1 insertion(+)

$ git push origin dev

Counting objects: 5, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 349 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)**To** git@github.com:michaelliao/learngit.git

fc38031..291bea8 dev -> dev

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

$ git add hello.py

$ git **commit** -m "add coding: utf-8"

[dev bd6ae48] **add** coding: utf-8

1 file changed, 1 insertion(+)

$ git push origin dev**To** git@github.com:michaelliao/learngit.git

! [rejected] dev -> dev (non-fast-forward)

error: failed **to** push **some** refs **to** 'git@github.com:michaelliao/learngit.git'

hint: Updates were rejected because the tip **of** your **current** branch **is** behind

hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

hint: before pushing again.

hint: See the 'Note about fast-forwards' **in** 'git push --help' **for** details.

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：

$ git pull

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From github.com:michaelliao/learngit

fc38031..291bea8 dev -> origin/dev

There is no tracking information **for** the current branch.

Please specify which branch you want to merge **with**.

See git-pull(1) **for** details

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream dev origin/<branch>

git pull也失败了，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 dev origin/dev*

Branch dev **set** up **to** track remote branch dev **from** origin.

再pull：

$ git pullAuto-merging hello.pyCONFLICT (content): Merge conflict **in** hello.pyAutomatic merge failed; fix conflicts **and** **then** commit the result.

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的[解决冲突](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001375840202368c74be33fbd884e71b570f2cc3c0d1dcf000" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)完全一样。解决后，提交，再push：

$ git **commit** -m "merge & fix hello.py"

[dev adca45d] merge & fix hello.py

$ git push origin dev

Counting objects: 10, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (6/6), 747 bytes, done.

Total 6 (delta 0), reused 0 (delta 0)**To** git@github.com:michaelliao/learngit.git

291bea8..adca45d dev -> dev

因此，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 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。

1. 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：

$ git branch

\* dev

master$ git checkout masterSwitched to branch 'master'

然后，敲命令git tag <name>就可以打一个新标签：

$ git tag v1.0

可以用命令git tag查看所有标签：

$ git tag

v1.0

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了：

$ git log --pretty=oneline --abbrev-commit6a5819e merged bug fix 101

cc17032 fix bug 1017825a50 merge with no-ff6224937 add merge59bc1cb conflict fixed400b400 & simple75a857c AND simple

fec145a branch test

d17efd8 remove test.txt

...

比方说要对add merge这次提交打标签，它对应的commit id是6224937，敲入命令：

$ git tag v0.9 6224937

再用命令git tag查看标签：

$ git tag

v0.9

v1.0

注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息：

$ git **show** v0.9**commit** 622493706ab447b6bb37e4e2a2f276a20fed2ab4

Author: Michael Liao <askxuefeng@gmail.com>**Date**: Thu Aug 22 11:22:08 2013 +0800

**add** merge

...

可以看到，v0.9确实打在add merge这次提交上。

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 v0.1 -m "version 0.1 released" 3628164

用命令git show <tagname>可以看到说明文字：

$ git show v0.1

tag v0.1

Tagger: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 26 07:28:11 2013 +0800

version 0.1 released

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 15:11:49 2013 +0800

append GPL

### 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。

1. 标签操作

如果标签打错了，也可以删除：

$ git tag -d v0.1Deleted tag 'v0.1' (was e078af9)

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>：

$ git push origin v1.0Total 0 (delta 0), reused 0 (delta 0)To git@github.com:michaelliao/learngit.git

\* [new tag] v1.0 -> 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

Counting objects: 1, done.

Writing objects: 100% (1/1), 554 bytes, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [**new** tag] v0.2 -> v0.2

\* [**new** tag] v0.9 -> v0.9

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 v0.9Deleted tag 'v0.9' (was 6224937)

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/v0.9To git@github.com:michaelliao/learngit.git

- [deleted] v0.9

1. 码云的使用

使用GitHub时，国内的用户经常遇到的问题是访问速度太慢，有时候还会出现无法连接的情况（原因你懂的）。

如果我们希望体验Git飞一般的速度，可以使用国内的Git托管服务——[码云](https://gitee.com/" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)（[gitee.com](https://gitee.com/" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)）。

和GitHub相比，码云也提供免费的Git仓库。此外，还集成了代码质量检测、项目演示等功能。对于团队协作开发，码云还提供了项目管理、代码托管、文档管理的服务，5人以下小团队免费。

 码云的免费版本也提供私有库功能，只是有5人的成员上限。

使用码云和使用GitHub类似，我们在码云上注册账号并登录后，需要先上传自己的SSH公钥。选择右上角用户头像 -> 菜单“修改资料”，然后选择“SSH公钥”，填写一个便于识别的标题，然后把用户主目录下的.ssh/id\_rsa.pub文件的内容粘贴进去：

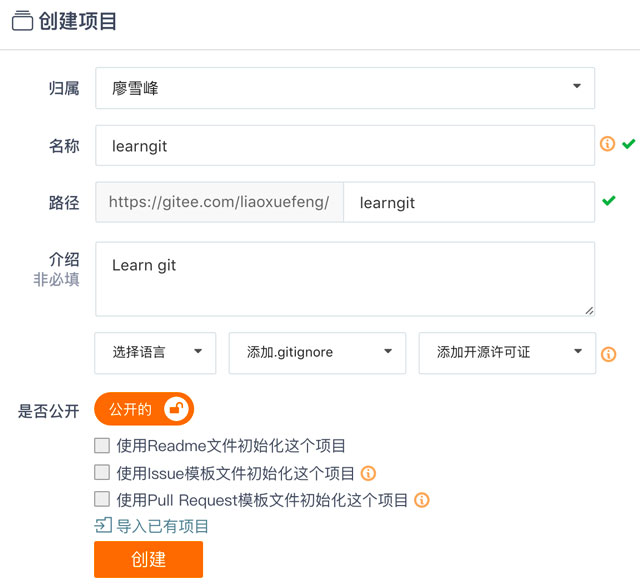


点击“确定”即可完成并看到刚才添加的Key：



如果我们已经有了一个本地的git仓库（例如，一个名为learngit的本地库），如何把它关联到码云的远程库上呢？

首先，我们在码云上创建一个新的项目，选择右上角用户头像 -> 菜单“控制面板”，然后点击“创建项目”：



项目名称最好与本地库保持一致：

然后，我们在本地库上使用命令git remote add把它和码云的远程库关联：

git remote add origin git@gitee.com:liaoxuefeng/learngit.git

之后，就可以正常地用git push和git pull推送了！

如果在使用命令git remote add时报错：

git remote add origin git@gitee.com:liaoxuefeng/learngit.gitfatal: remote origin already exists.

这说明本地库已经关联了一个名叫origin的远程库，此时，可以先用git remote -v查看远程库信息：

git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

可以看到，本地库已经关联了origin的远程库，并且，该远程库指向GitHub。

我们可以删除已有的GitHub远程库：

git remote rm origin

再关联码云的远程库（注意路径中需要填写正确的用户名）：

git remote add origin git@gitee.com:liaoxuefeng/learngit.git

此时，我们再查看远程库信息：

git remote -v

origin git@gitee.com:liaoxuefeng/learngit.git (fetch)

origin git@gitee.com:liaoxuefeng/learngit.git (push)

现在可以看到，origin已经被关联到码云的远程库了。通过git push命令就可以把本地库推送到Gitee上。

有的小伙伴又要问了，一个本地库能不能既关联GitHub，又关联码云呢？

答案是肯定的，因为git本身是分布式版本控制系统，可以同步到另外一个远程库，当然也可以同步到另外两个远程库。

使用多个远程库时，我们要注意，git给远程库起的默认名称是origin，如果有多个远程库，我们需要用不同的名称来标识不同的远程库。

仍然以learngit本地库为例，我们先删除已关联的名为origin的远程库：

git remote rm origin

然后，先关联GitHub的远程库：

git remote add github git@github.com:michaelliao/learngit.git

注意，远程库的名称叫github，不叫origin了。

接着，再关联码云的远程库：

git remote add gitee git@gitee.com:liaoxuefeng/learngit.git

同样注意，远程库的名称叫gitee，不叫origin。

现在，我们用git remote -v查看远程库信息，可以看到两个远程库：

git remote -v

gitee git@gitee.com:liaoxuefeng/learngit.git (fetch)

gitee git@gitee.com:liaoxuefeng/learngit.git (push)

github git@github.com:michaelliao/learngit.git (fetch)

github git@github.com:michaelliao/learngit.git (push)

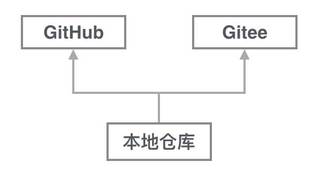
如果要推送到GitHub，使用命令：

git push github master

如果要推送到码云，使用命令：

git push gitee master

这样一来，我们的本地库就可以同时与多个远程库互相同步：



码云也同样提供了Pull request功能，可以让其他小伙伴参与到开源项目中来。你可以通过Fork我的仓库：[https://gitee.com/liaoxuefeng/learngit](https://gitee.com/liaoxuefeng/learngit" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)，创建一个your-gitee-id.txt的文本文件， 写点自己学习Git的心得，然后推送一个pull request给我，这个仓库会在码云和GitHub做双向同步。

1. 配置别名（使用简写设置命令）

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

好了，现在敲git st看看效果。

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](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001374831943254ee90db11b13d4ba9a73b9047f4fb968d000" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)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：



为什么不早点告诉我？别激动，咱不是为了多记几个英文单词嘛！

### 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：

$ cat .git/config

[core]

repositoryformatversion = 0

filemode = **true**

bare = **false**

logallrefupdates = **true**

ignorecase = **true**

precomposeunicode = **true**

[remote "origin"]

url = git@github.com:michaelliao/learngit.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

[**alias**]

last = log -1

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

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：

$ cat .gitconfig

[**alias**]

co = checkout

ci = commit

br = branch

st = status

[user]

name = Your Name

email = your@email.com

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

1. 搭建本地git服务器

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.gitCloning 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" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)来管理公钥。

这里我们不介绍怎么玩[Gitosis](https://github.com/res0nat0r/gitosis" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)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" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)就是这个工具。

这里我们也不介绍[Gitolite](https://github.com/sitaramc/gitolite" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)了，不要把有限的生命浪费到权限斗争中。

### **小结：**

1>创建版本库

添加文件到Git仓库，分两步：

第一步，使用命令git add <file>，注意，可反复多次使用，添加多个文件；

第二步，使用命令git commit，完成。

2>查看工作区状态

要随时掌握工作区的状态，使用git status命令。

如果git status告诉你有文件被修改过，用git diff可以查看修改内容。

1. 版本回退

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。

1. 工作区暂存区

暂存区是Git非常重要的概念，弄明白了暂存区，就弄明白了Git的很多操作到底干了什么

1. 管理修改

Git是如何跟踪修改的，每次修改，如果不add到暂存区，那就不会加入到commit中。

1. 撤销修改

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

1. 删除文件

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****

1. ****添加远程仓库****

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；

#### ****9>****从远程库克隆

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。

#### ****10>****创建与合并分支

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>

#### ****11>****解决冲突

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。

用git log --graph命令可以看到分支合并图。

#### ****12>****分支管理策略

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。

#### ****13>****Bug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。

1. ****新增加功能分支处理****

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。

#### ****15>****多人协作

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。

#### ****16>****创建标签(版本号)

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。

1. ****操作标签****

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。

18>搭建git本地服务器

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" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)；

要像SVN那样变态地控制权限，用[Gitolite](https://github.com/sitaramc/gitolite" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)。