# Git 基础

什么是版本库呢？版本库又名仓库，英文名repository，你可以简单理解成一个目录，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”。

## 1.1 创建版本库

### （1）一个空文件夹

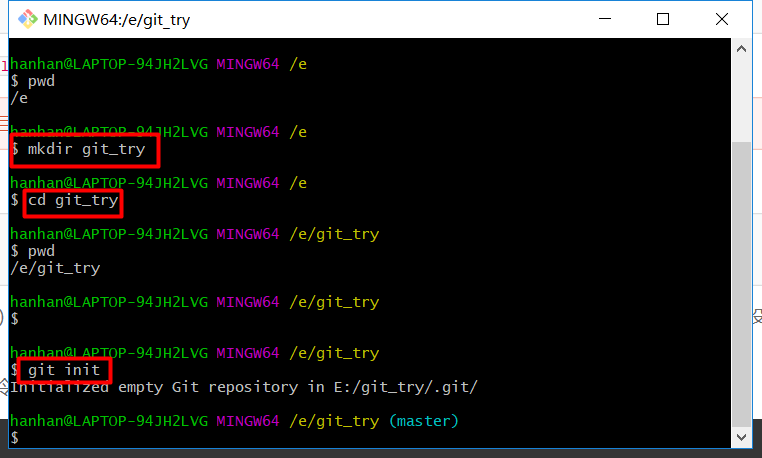
打开git bash，cd 某个空文件夹，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

$ mkdir learngit

$ cd learngit

### （2）git init

通过git init命令把这个目录变成Git可以管理的仓库。如下：



创建成功，在E:/git\_try/.git/下初始化了空的Git仓库（版本库）。

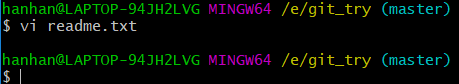
首先这里再明确一下，所有的版本控制系统，其实只能跟踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。版本控制系统可以告诉你每次的改动，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

不幸的是，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。

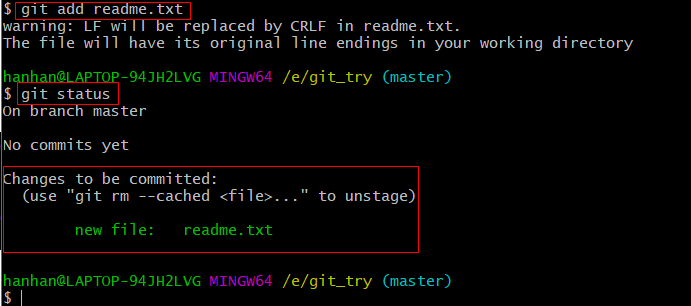
因为文本是有编码的，比如中文有常用的GBK编码，日文有Shift\_JIS编码，如果没有历史遗留问题，强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

### git add,git rm --cached,git rm -f

在本地仓库下新建一个readme.txt文件：

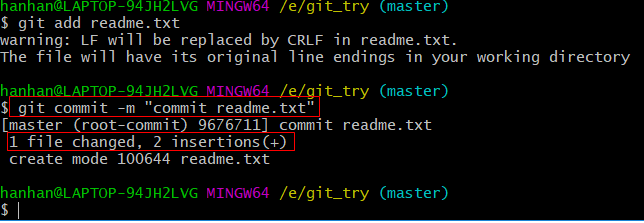


git add readme.txt，将本地仓库中的readme.txt添加到暂存区，并使用git status查看状态，显示新建了一个文件：readme.txt（使用git rm --cached readme.txt从暂存区删除文件，本地仓库的文件还在；使用git rm -f readme.txt会将暂存区和本地的文件都删除掉）。



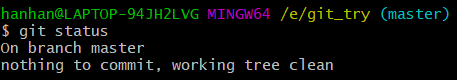
git add file1 file2 file3可以一次性添加多个文件。

### git commit -m “comment”



git commit -m “comment”是将暂存区中的文件提交，提交成功后会显示文件跟踪信息，一个文件被改变，插入了两行。

再输入git status查看状态：



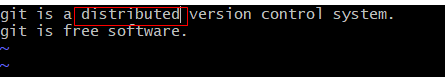
工作目录clean。

git commit是将暂存区中的所有文件一次性提交。

## 1.2 版本回退

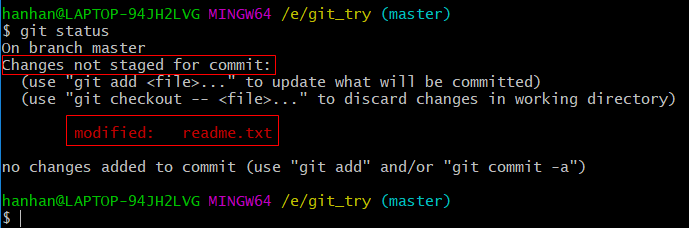
修改readme.txt的内容，添加一个单词：





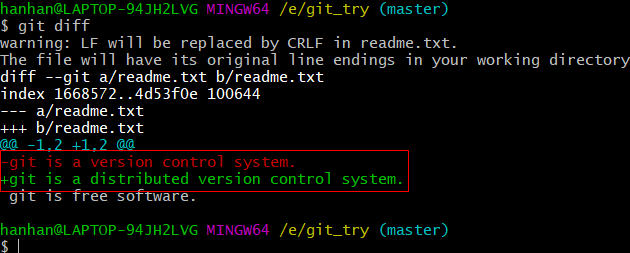
### (1)git status

git status查看仓库的当前状态：



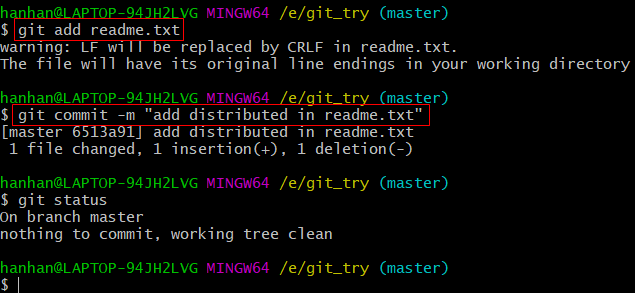
提示修改未暂存（changes not staged），readme.txt文件被修改（modified：readme.txt）。

使用命令git diff查看文件作了哪些修改：



变化内容为：减号开头的红色行被删掉，加号开头的绿色行被添加。

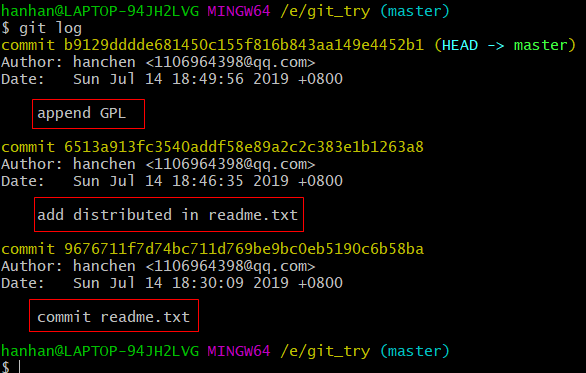
将修改提交到版本库：



像这样，不断对文件进行修改，然后不断提交修改到版本库里，每当文件修改到一定程度的时候，就可以“保存一个“**快照**”，这个快照在Git中被称为commit。一旦文件被改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

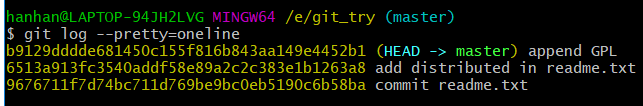
### git log

版本控制系统可以告诉我们文件历次修改的历史记录，在Git中，我们用git log命令查看：



git log命令显示从最近到最远的提交日志，可以看到3次提交，最近的一次是append GPL，上一次是add distributed，最早的一次是commit readme.txt。

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：



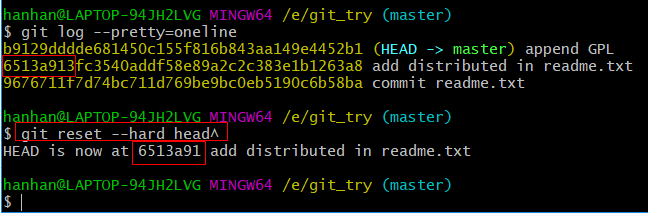
黄色的长串数字是commit id（版本号）

### git reset版本回退

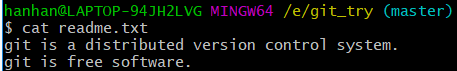
把readme.txt回退到上一个版本，也就是add distributed的那个版本，怎么做呢？

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，也就是最新的提交b9129dddde...，上一个版本就是HEAD^，上上一个版本就是HEAD^^，往上100个版本写成HEAD~100。

现在，我们要把当前版本append GPL回退到上一个版本add distributed，就可以使用git reset命令：

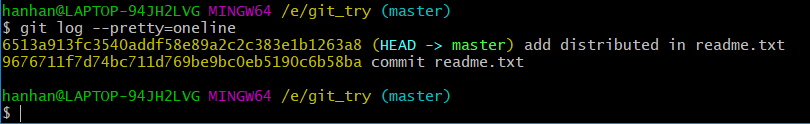


Readme.txt已经被回退到了上一个版本，查看其内容：

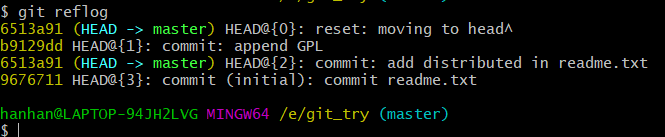


内容已经被还原。

版本回退之后再使用git log查看版本库的状态：



发现最新的add GPL版本已经不见了，如果想要再回到GPL那个版本，必须要知道其commit id，此时可以使用git reflog来查看输入的每一个git命令：



可以看到最近的命令是在append GPL的commit id是，所以，要回到这个版本，只需要使用reset指令：



又回到了最初的起点~

## 1.3 工作区和暂存区

工作区（Working Directory）

就是你在电脑里能看到的目录，比如本地的git\_try文件夹就是一个工作区;

版本库（Repository）

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

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。



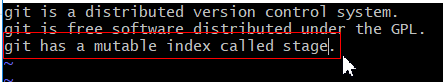
git add实际上就是把文件修改添加到暂存区；

git commit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。

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。

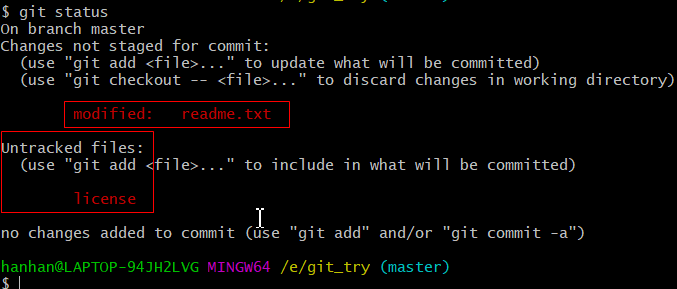
现在向readme.txt中添加一行：



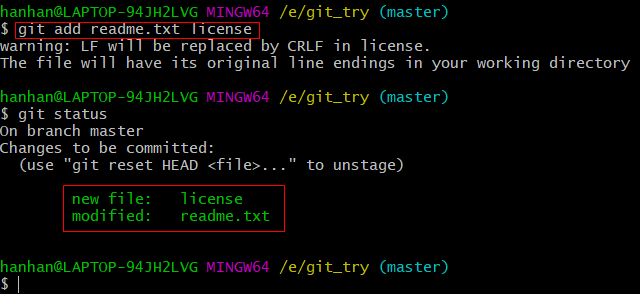
再新增一个license文件（内容随便）：



使用git status查看当前状态：



显示readme.txt文件被修改，license未被添加过，所以是untracked，将这两个文件add到暂存区:

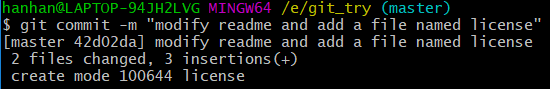


(注意，git diff是查看本地仓库中的文件做了哪些修改，一旦被修改的文件被add到了暂存区，就看不到diff内容了。）

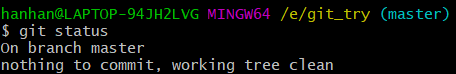
此时，暂存区的状态就变成了下图：



所以，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：



再使用git status查看版本库的状态：



现在版本库变成了这样，暂存区就没有任何内容了：

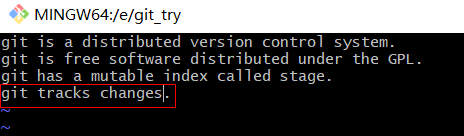


## 1.4 管理修改

Git跟踪并管理的是修改，而非文件。

什么叫管理修改？如下:

修改readme.txt的内容,新增一行：



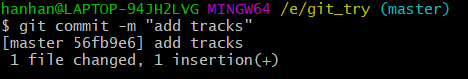
将文件add到暂存区：



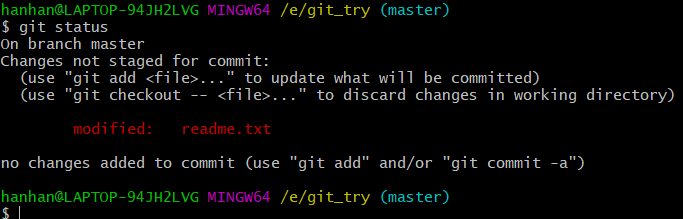
再修改readme.txt的内容:



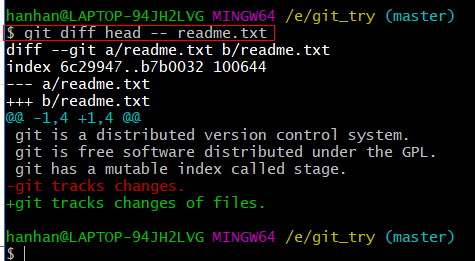
再提交:



再git status查看状态：



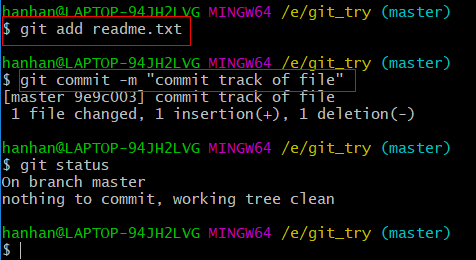
显示第二次的修改没有被提交，因为Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。提交后，用git diff HEAD -- readme.txt命令可以查看工作区和版本库里面最新版本的区别：



（注意：“--”前后都有空格）

可以看到第二次修改确实没有被提交。

再add、commit完成第二次修改的提交：



## 1.5 撤销修改git chech out -- filename

git checkout -- file可以丢弃工作区的修改：

$ 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：

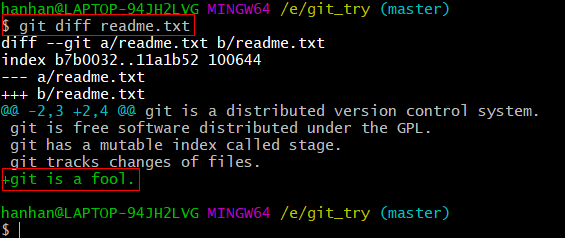
一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

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。

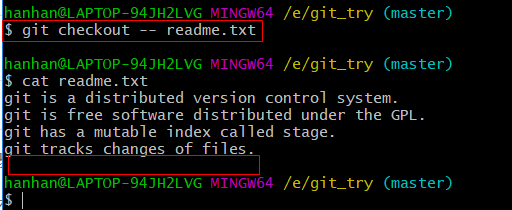
总之，就是让这个文件回到最近一次git commit或git add时的状态。

例如:

在readme.txt中新增一行：



使用git checkout -- readme.txt撤销本地的修改:



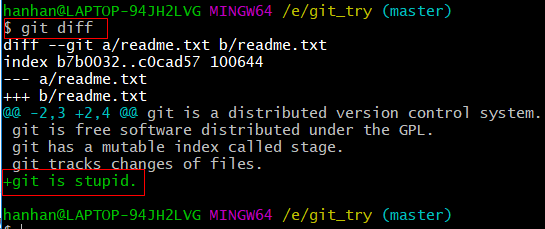
文件内容复原了。

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。

再看另外一种情况：

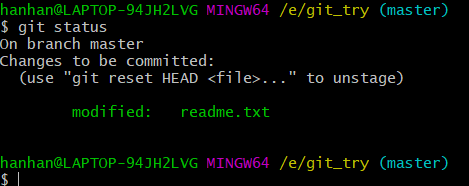
在readme.txt中新增一行：



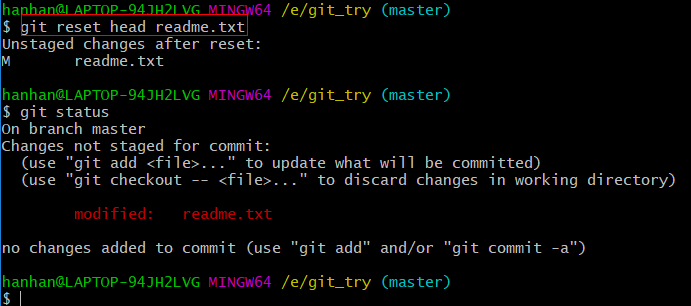
add到暂存区：



git status查看当前状态：

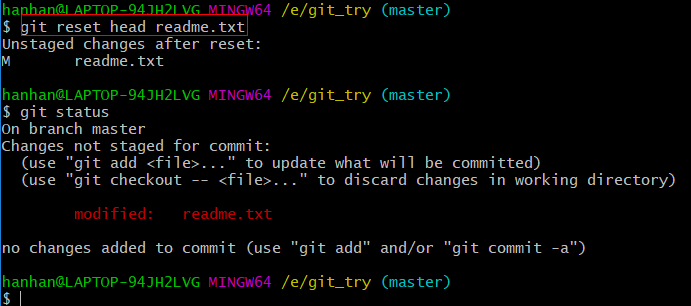


git告诉我们，用命令git reset HEAD <file>可以把暂存区的修改撤销掉（unstage），重新放回工作区，git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本：

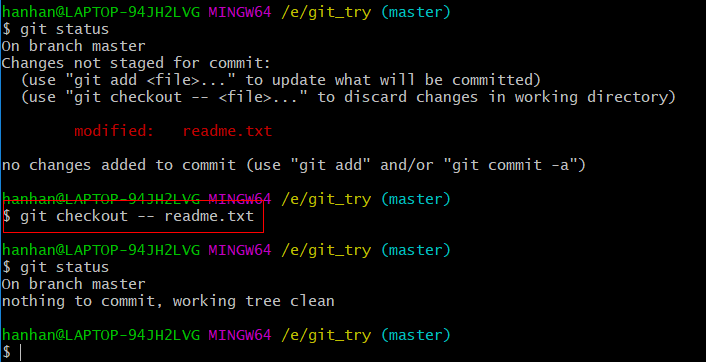


git reset head readme.txt就是将readme.txt这个文件已经提交到暂存区的修改丢弃掉。

再用git status查看一下，现在暂存区是干净的，工作区有修改：



再使用checkout将工作区的修改丢弃掉：



**撤销修改小结**

**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。**

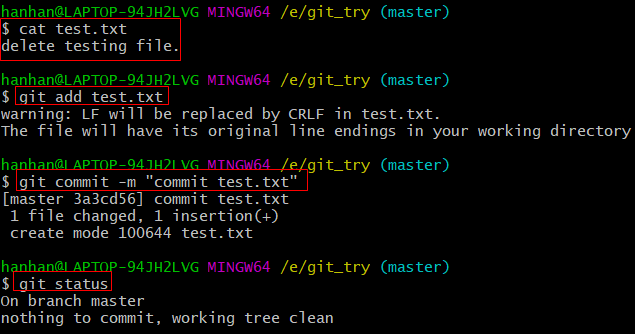
**场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD <file>，就回到了场景1，第二步按场景1操作。**

**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**](https://www.liaoxuefeng.com/wiki/896043488029600/897013573512192)**一节，不过前提是没有推送到远程库。**

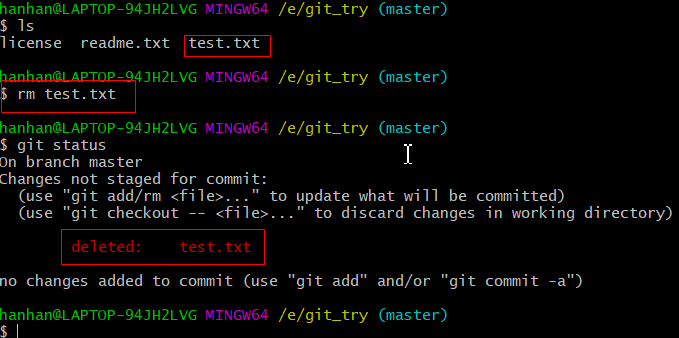
## 1.6 删除

在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：

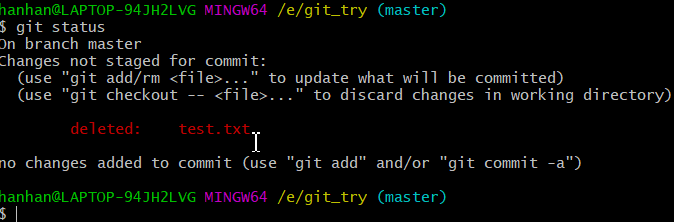




先直接把本地用rm命令把文件删了：

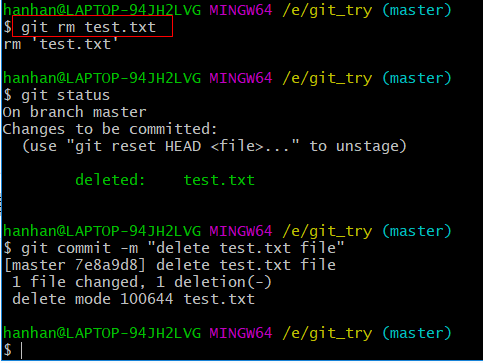


这个时候，使用git status命令查看当前状态：



此时本地仓库和版本库不一致了，所以提示说test.txt被删除了。

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：



现在，文件就从版本库中被删除了。

小提示：先手动删除文件，然后使用git rm <file>和git add<file>效果是一样的。

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”。

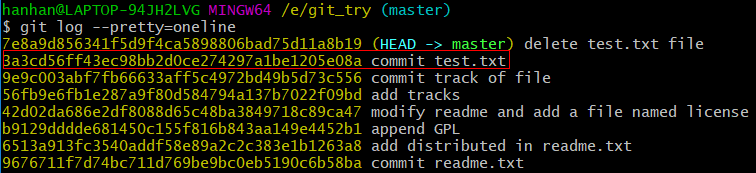
注意：从来没有被添加到版本库就被删除的文件，是无法恢复的！

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：

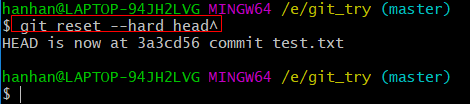
$ git checkout -- test.txt



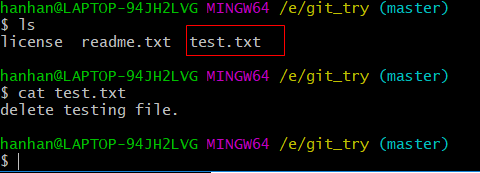
因为当前版本库中的test.txt文件已经被git rm命令删除掉了，所以找不到（did not match）。所以要使用git reset命令回退到上一个版本：



可以使用git reset --hard head^ 或者git reset --hard 3a3cd回退到上一个版本：



查看工作区的文件：



**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。**

# 操作远程库

现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub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，真是一举多得。

首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：



在Repository name填入learngit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的learngit仓库下运行命令：

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

请千万注意，把上面的michaelliao替换成你自己的GitHub账户名，否则，你在本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

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。

下一步，就可以把本地库的所有内容推送到远程库上：

$ git push -u origin master

Counting objects: 20, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.

Total 20 (delta 5), reused 0 (delta 0)

remote: Resolving deltas: 100% (5/5), done.

To github.com:michaelliao/learngit.git

\* [new branch] master -> master

Branch 'master' set up to track remote branch 'master' from 'origin'.

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。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：



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，现在，你就拥有了真正的分布式版本库！

### 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；

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，

**注意：**

Merge是合并commit，所以修改冲突之后，还要提交该文件。

# 分支管理

## 创建与合并分支

在[版本回退](https://www.liaoxuefeng.com/wiki/896043488029600/897013573512192)里已经知道，每次提交，Git都把commit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，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：



所以，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：



下面开始实战。

首先，我们创建dev分支，然后切换到dev分支：

使用命令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

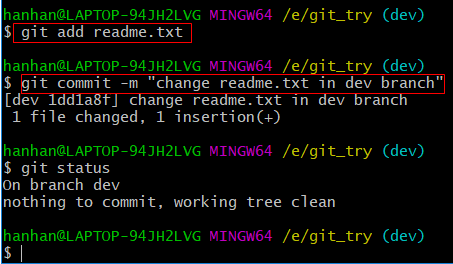


git branch命令会列出所有分支，当前分支前面会标一个\*号。

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：



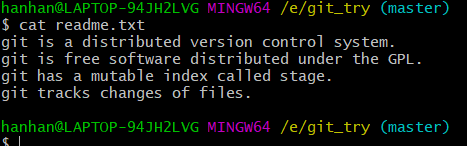
然后提交：



现在，dev分支的工作完成，我们就可以切换回master分支：



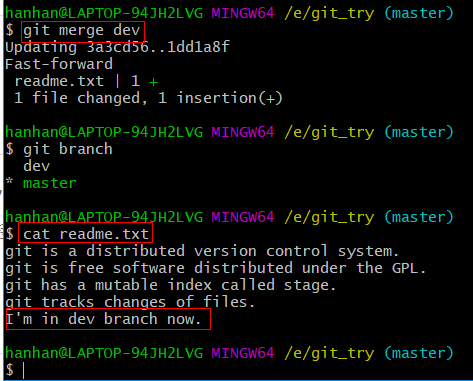
切换回master分支后，再查看一个readme.txt文件



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，如下图：



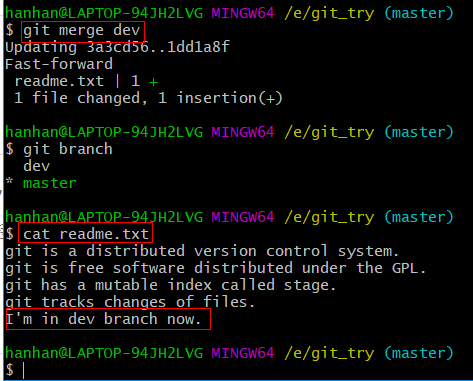
现在，我们把dev分支的工作成果合并到master分支上：



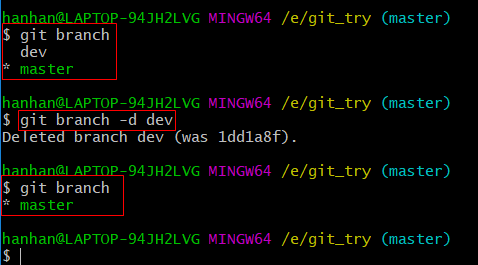
注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

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。

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：



合并完成后，就可以放心地删除dev分支了：



可以看到，删除后，查看branch，就只剩下master分支了。

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。

**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>

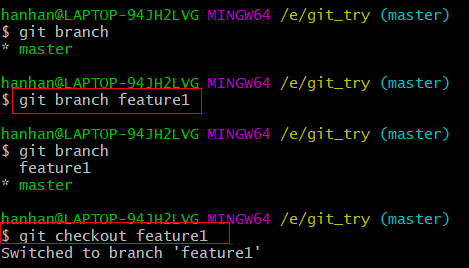
合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

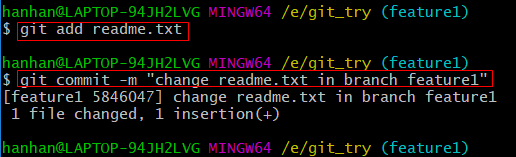
## 3.2 解决冲突

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的feature1分支，继续我们的新分支开发：

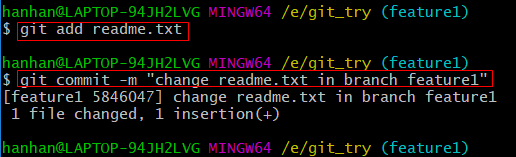


修改readme.txt最后一行，在feature1分支上提交：

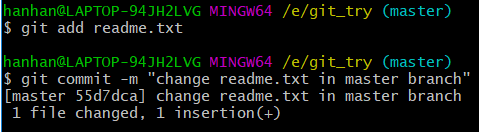


切换到master分支：

$ git checkout master



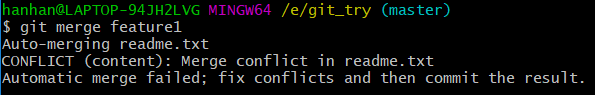
在master分支上把readme.txt文件的末尾添加一行，再提交：



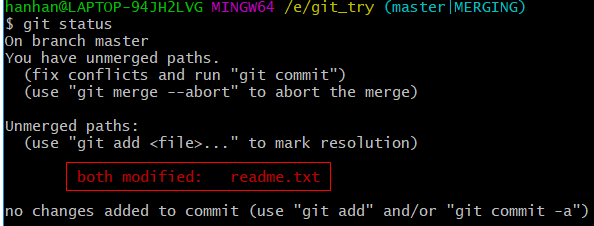
现在，master分支和feature1分支各自都分别有新的提交，变成了这样：



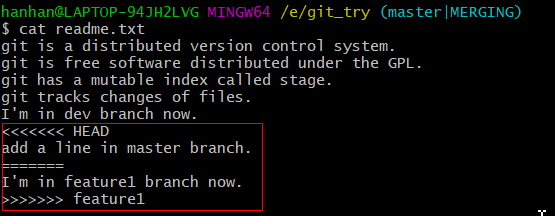
这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：



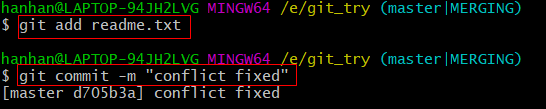
果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：



我们可以直接查看readme.txt的内容：



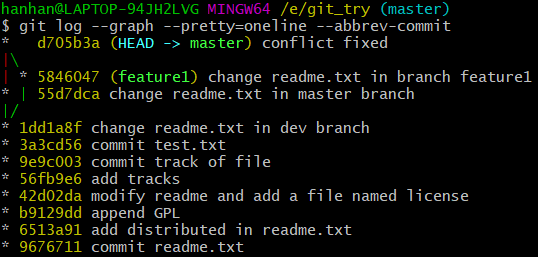
Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改后保存，再提交：



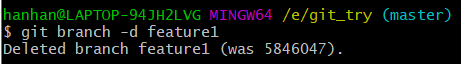
现在，master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支的合并情况：



最后，删除feature1分支：



工作完成。

**小结**

**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。**

## 3.3 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 stash

Saved working directory and index state WIP on dev: f52c633 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，也就是说git stash暂存的是暂存区中的内容，工作区未add的修改是保存不下来的），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 master

Switched to branch 'master'Your branch is ahead of 'origin/master' by 6 commits.

(use "git push" to publish your local commits)

$ git checkout -b issue-101

Switched 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 4c805e2] 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 master

Switched to branch 'master'Your branch is ahead of 'origin/master' by 6 commits.

(use "git push" to publish your local 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(-)

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 tree 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: f52c633 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} (5d677e2ee266f39ea296182fb2354265b91b3b2a)

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}

Git stash只能把对文件的修改保存下来，新增文件是保存不下来的。

注意：

git stash list中，因为是栈，所以后进去的排在前面，所以编号0是最新进栈的修改；当drop掉第0个的时候，栈内的元素编号-1.

**小结**

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。

**注意：**

Stash pop的时候，实际上是对工作区的文件进行一个merge，可能会有冲突，所以，必须先修改冲突（修改、add、commit），再stash pop或stash apply。

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.git

Cloning into 'learngit'...

remote: Counting objects: 40, done.

remote: Compressing objects: 100% (21/21), done.

remote: Total 40 (delta 14), reused 40 (delta 14), pack-reused 0

Receiving objects: 100% (40/40), done.

Resolving deltas: 100% (14/14), 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 add env.txt

$ git commit -m "add env"

[dev 7a5e5dd] add env

1 file changed, 1 insertion(+)

create mode 100644 env.txt

$ git push origin dev

Counting objects: 3, done.

Delta compression using up to 4 threads.

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

Writing objects: 100% (3/3), 308 bytes | 308.00 KiB/s, done.

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

To github.com:michaelliao/learngit.git

f52c633..7a5e5dd 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：

$ cat env.txt

env

$ git add env.txt

$ git commit -m "add new env"

[dev 7bd91f1] add new env

1 file changed, 1 insertion(+)

create mode 100644 env.txt

$ git push origin dev

To 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. Integrate the remote changes (e.g.

hint: 'git pull ...') 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

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-to=origin/<branch> dev

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-to=origin/dev dev

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

再pull：

$ git pull

Auto-merging env.txt

CONFLICT (add/add): Merge conflict in env.txt

Automatic 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/896043488029600/900004111093344)完全一样。解决后，提交，再push：

$ git commit -m "fix env conflict"

[dev 57c53ab] fix env conflict

$ git push origin dev

Counting objects: 6, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (6/6), 621 bytes | 621.00 KiB/s, done.

Total 6 (delta 0), reused 0 (delta 0)

To github.com:michaelliao/learngit.git

7a5e5dd..57c53ab dev -> dev

因此，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-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。