1. Git的简介

Git是目前世界上最先进的分布式版本控制系统。使用情况：



优点：不但能自动记录每次文件的改动，还可以让同事协作编辑。

Git的开发语言是：C语言

1. 集中式版本控制系统与分布式版本控制系统

**集中式版本控制系统**：版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟。CVS及SVN都是集中式的版本控制系统。

**分布式版本控制系统：**分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

**分布式优点**：和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

**分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。**

1. Git的安装

**在Linux上安装Git**

Debian或Ubuntu Linux(新版)： sudo apt-get install git

Debian或Ubuntu Linux(老版)： sudo apt-get install git-core

其他Linux版本：先从Git官网下载源码，然后解压，依次输入：./config，make，sudo make install这几个命令安装就好了。(https://git-scm.com/downloads)

**在Windows上安装Git**

从Git官网直接下载安装程序，然后按默认选项安装即可。

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！

安装完成后，还需要最后一步设置，在命令行输入：



因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。

git config命令的**--global**参数，用了这个参数，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。

1. 创建版本库

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”。

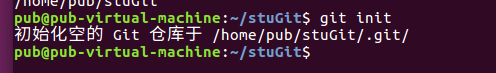
创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：



注：如果你使用Windows系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

第二步，通过git init命令把这个目录变成Git可以管理的仓库：





可以发现当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

1. 把文件添加到版本库

所有的版本控制系统，其实**只能**跟踪**文本文件**的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。

版本控制系统可以告诉你每次的改动，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。

而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的。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。

**使用Windows要特别注意：**

不要使用Windows自带的记事本编辑任何文本文件。原因是Microsoft开发记事本的团队在保存UTF-8编码的文件时在每个文件开头添加了0xefbbbf（十六进制）的字符。

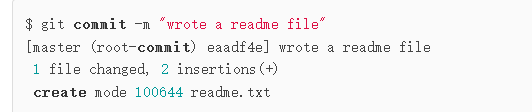
建议使用[Visual Studio Code](https://code.visualstudio.com/)代替记事本。

现在我们编写一个readme.txt文件，内容随便：(touch+gedit+cat)

第一步，用命令git add告诉Git，把文件添加到仓库：

正常情况下，没有任何显示结果

第二步，用命令git commit告诉Git，把文件提交到仓库：



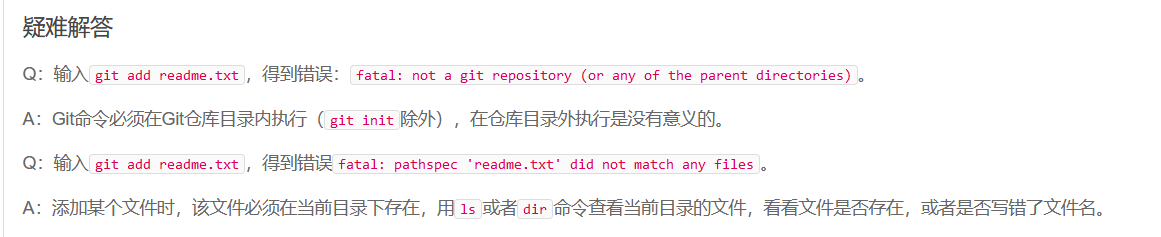
git commit命令：

-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

git commit命令执行成功后会告诉你，1 file changed：1个文件被改动（我们新添加的readme.txt文件）；2 insertions：插入了两行内容（readme.txt有两行内容）。

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不同的文件，比如：





1. 版本回退相关操作
2. 版本回退

版本控制系统肯定有某个命令可以告诉我们历史记录，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，我们可以看到2次提交，最近的一次是append GPL，最早的一次是我创建了第一个文本文件。

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：



Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示

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

**进行版本回退：**

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

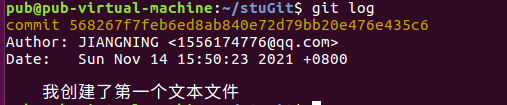
**上一个版本就是HEAD^**，**上上一个版本就是HEAD^^**，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

使用**git reset**命令：



同时用gedit查看readme.txt发现内容变更

让我们用**git log**再看看现在版本库的状态：



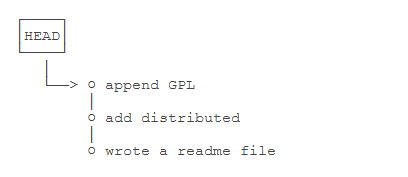
最新的那个版本append GPL已经看不到了。如果想要再次回到append GPL对应的版本：



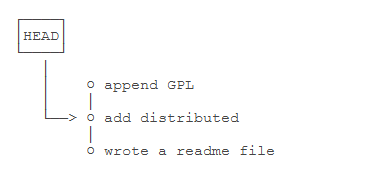
即输入其对应的版本号即可，版本号没必要写全，前几位就可以了，Git会自动去找。

**版本回退的原理：**

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append GPL：



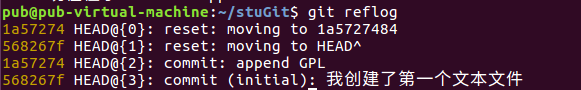
改为指向add distributed：



然后顺便把工作区的文件更新了。所以你让HEAD指向哪个版本号，你就把当前版本定位在哪。

**忘记版本号怎么办？**

**Git提供了一个命令git reflog用来记录你的每一次命令：**

****

在这个日志文件中，所有的操作(commit)都会被记录，而且commit id也会被记录

1. 工作区和暂存区

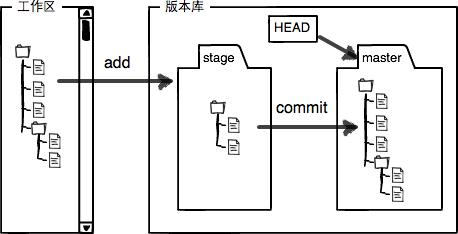
**工作区：**

在电脑里能看到的目录，比如我的stuGit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。



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；

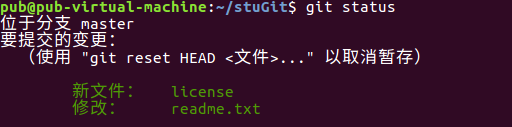
第二步是用**git commit**提交更改，实际上就是把暂存区的所有内容提交到当前分支。

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。

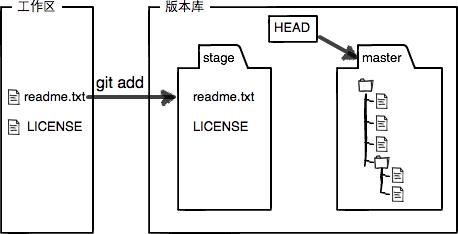
需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

在工作区新增一个LICENSE文本文件（内容随便写）。

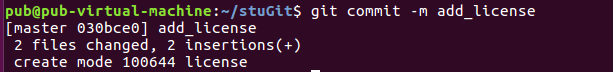
使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：



现在，暂存区的状态就变成这样了：

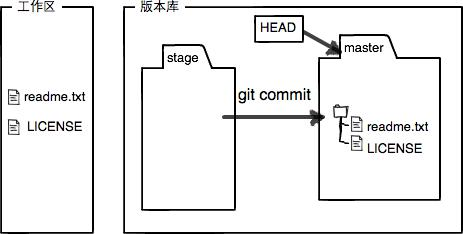


所以，git add命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行git commit就可以**一次性**把暂存区的所有修改提交到分支：



一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：





1. 管理修改

为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

Git commit提交的总是git add到暂存区中的数据，并不会把工作区中的数据提交。

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

1. 撤销修改
   1. 处理掉工作区的错误

**git checkout -- file**可以丢弃**工作区**的修改。让这个文件回到最近一次git commit或git add时的状态，

具体的：

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；

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。

* 1. 处理掉暂存区的错误

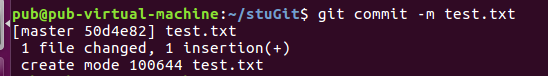
**git reset HEAD <file>**可以把**暂存区**的修改撤销掉（unstage），重新放回工作区。可以撤销掉那些已经完成 git add ，但还未完成 git commit的数据。接下来，再次使用**git checkout – file** 丢弃掉工作区的修改

* 1. 处理掉版本库的错误

版本回退

1. 删除文件

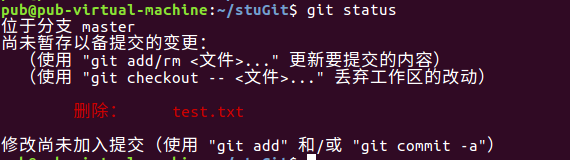
添加一个新文件test.txt到Git并且提交：



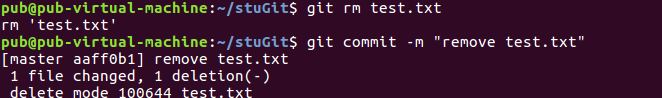
一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：



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(删除操作也是修改)：



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：



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**

如果git rm已生效，但我们反悔了，可以进行版本回退后，在进行git checkout -- test.txt

1. 远程仓库

**GitHub远程仓库的使用：**

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"**

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，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。

为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

在GitHub上免费托管的Git仓库，任何人都可以看到。如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

1. 添加远程库

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。

第一步：在Gitee上创建一个新的仓库

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。

根据GitHub的提示，在本地的stuGit仓库下运行命令：



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：



输入登录时的账号与密码完成推送

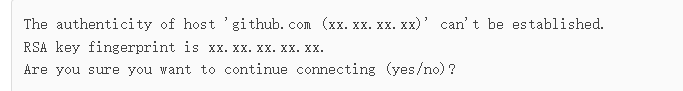
把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。

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。

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

**SSH警告**

第一次使用Git的clone或者push命令连接GitHub时，会得到一个警告：



这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。

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：

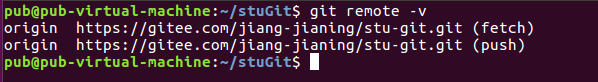
Warning: Permanently added 'github.com' (RSA) to the list of known hosts.

这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入yes前可以对照[GitHub的RSA Key的指纹信息](https://help.github.com/articles/what-are-github-s-ssh-key-fingerprints/" \t "_blank)是否与SSH连接给出的一致。

**删除远程库**

如果添加的时候地址写错了，或者就是想删除远程库，可以用git remote rm <name>命令。使用前，建议先用git remote -v查看远程库信息：



然后，根据名字删除，比如删除origin：**git remote rm origin**

此处的“删除”其实是解除了本地和远程的绑定关系，并不是物理上删除了远程库。远程库本身并没有任何改动。要真正删除远程库，需要登录到GitHub，在后台页面找到删除按钮再删除。

1. 从远程库克隆

登陆GitHub，创建一个新的仓库，名字叫gitskills：

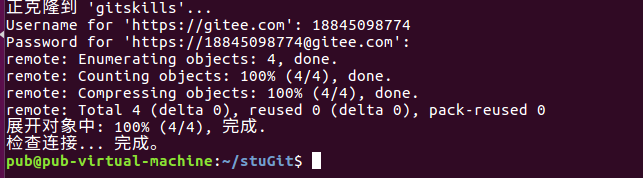


GitHub会自动为我们创建一个README.md文件。创建完毕后，可以看到README.md文件：

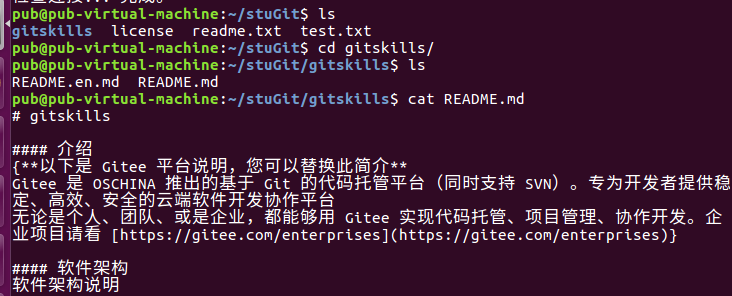


远程库已经准备好了，下一步是用命令git clone克隆一个本地库：



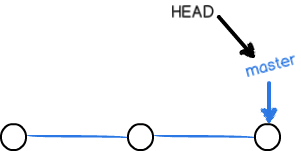


然后进入gitskills目录看看，已经有README.md文件了：



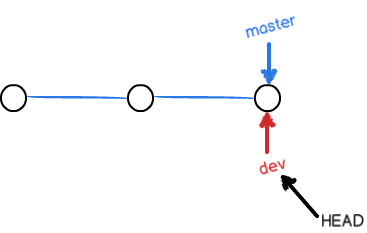
1. 分支管理
2. 创建与合并分支

在版本回退里，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。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：



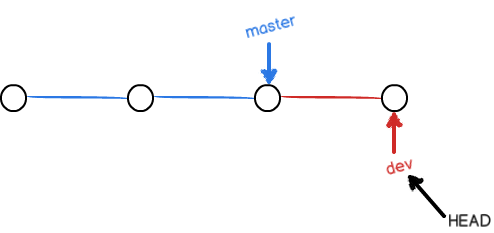
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

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：

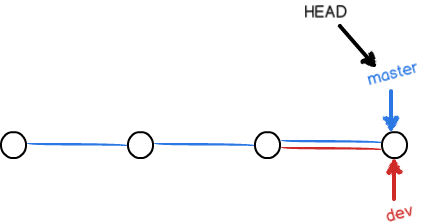


Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

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：

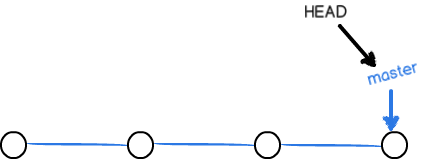


假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



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：



**实战演练**

首先，我们创建dev分支，然后切换到dev分支：

|  |
| --- |
| $ git checkout -b dev  Switched 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 dev  Switched 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。 |

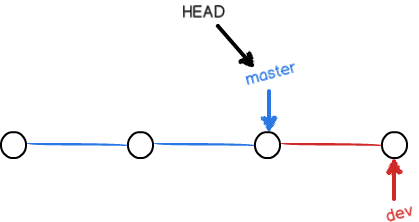
然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行，然后提交

|  |
| --- |
| $ git add readme.txt  $ git commit -m "branch test"  [dev b17d20e] 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 master  Switched 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 dev  Updating d46f35e..b17d20e  Fast-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 dev  Deleted branch dev (was b17d20e). |

删除后，查看branch，就只剩下master分支了：

|  |
| --- |
| $ git branch  \* master |

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。

我们注意到切换分支使用git checkout <branch>，而前面讲过的撤销修改则是git checkout -- <file>，同一个命令，有两种作用，确实有点令人迷惑。实际上，切换分支这个动作，用switch更科学。因此，最新版本的Git提供了新的git switch命令来切换分支：

创建并切换到新的dev分支，可以使用：

|  |
| --- |
| $ git switch -c dev |

直接切换到已有的master分支，可以使用：

|  |
| --- |
| $ git switch master |

**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>或者git switch <name>  创建+切换分支：git checkout -b <name>或者git switch -c <name>  合并某分支到当前分支：git merge <name>  删除分支：git branch -d <name> |

1. 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。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。

准备新的feature1分支，继续我们的新分支开发：

|  |
| --- |
| $ git switch -c feature1  Switched to a new branch 'feature1' |

修改readme.txt最后一行，改为：

Creating a new branch is quick **AND simple.**

在feature1分支上提交：

|  |
| --- |
| $ git add readme.txt  $ git commit -m "AND simple"  [feature1 14096d0] AND simple  1 file changed, 1 insertion(+), 1 deletion(-) |

切换到master分支

|  |
| --- |
| $ git switch master  Switched to branch 'master'  Your branch is ahead of 'origin/master' by 1 commit.  (use "git push" to publish your local commits) |

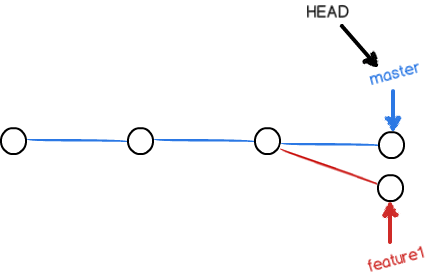
在master分支上把readme.txt文件的最后一行改为：

|  |
| --- |
| Creating a new branch is quick **& simple.** |

提交：

|  |
| --- |
| $ git add readme.txt  $ git commit -m "& simple"  [master 5dc6824] & simple  1 file changed, 1 insertion(+), 1 deletion(-) |

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：

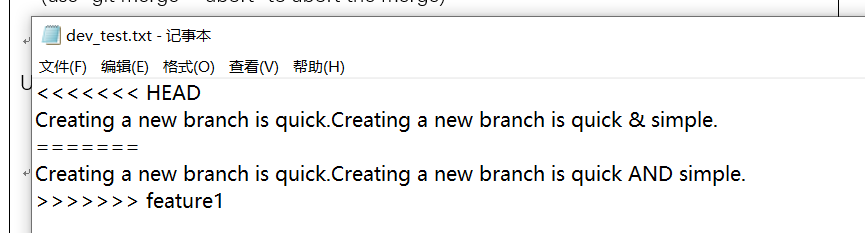
|  |
| --- |
| $ git merge feature1  Auto-merging readme.txt  CONFLICT (content): **Merge conflict in readme.txt**  Automatic merge failed; fix conflicts and then commit the result. |

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.  (use "git push" to publish your local commits)  You have unmerged paths.  (fix conflicts and run "git commit")  (use "git merge --abort" to abort the merge)  Unmerged paths:  (use "git add <file>..." to mark resolution)  both modified: readme.txt  no changes added to commit (use "git add" and/or "git commit -a") |

我们可以直接查看readme.txt的内容：

|  |
| --- |
| 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 |



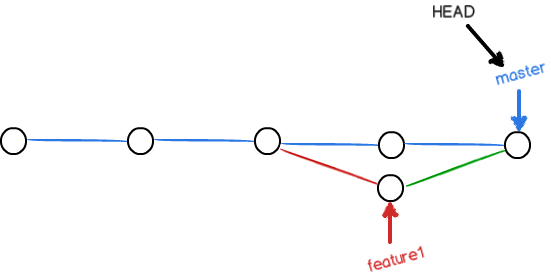
Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：

|  |
| --- |
| Creating a new branch is quick and simple. |

再提交：

|  |
| --- |
| $ git add readme.txt  $ git commit -m "conflict fixed"  [master cf810e4] 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：

（用git log --graph命令可以看到分支合并图。）

|  |
| --- |
| $ git log --graph --pretty=oneline --abbrev-commit  \* cf810e4 (HEAD -> master) conflict fixed  |\  | \* 14096d0 (feature1) AND simple  \* | 5dc6824 & simple  |/  \* b17d20e branch test  \* d46f35e (origin/master) remove test.txt  \* b84166e add test.txt  \* 519219b git tracks changes  \* e43a48b understand how stage works  \* 1094adb append GPL  \* e475afc add distributed  \* eaadf4e wrote a readme file |

最后，删除feature1分支：

|  |
| --- |
| $ git branch -d feature1  Deleted branch feature1 (was 14096d0). |

1. 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。

下面我们实战一下--no-ff方式的git merge：

首先，仍然创建并切换dev分支：

|  |
| --- |
| $ git switch -c dev  Switched 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 f52c633] add merge  1 file changed, 1 insertion(+) |

现在，我们切换回master：

|  |
| --- |
| $ git switch master  Switched to branch 'master' |

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

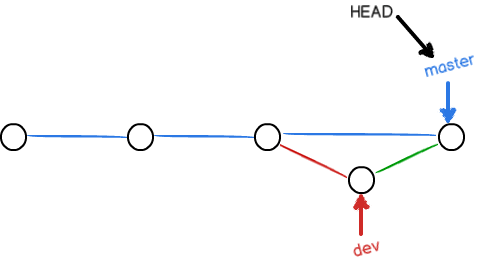
|  |
| --- |
| $ git merge --no-ff -m "merge with no-ff" dev  Merge made by the 'recursive' strategy.  readme.txt | 1 +  1 file changed, 1 insertion(+) |

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  \* e1e9c68 (HEAD -> master) merge with no-ff  |\  | \* f52c633 (dev) add merge  |/  \* cf810e4 conflict fixed  ... |

可以看到，不使用Fast forward模式，merge后就像这样：(dev没有与master指向同一处)

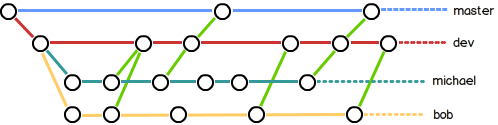


合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

**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：



1. 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：(没有完成git commit)

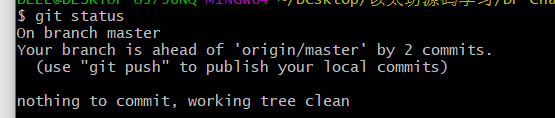
|  |
| --- |
| $ git status  On branch dev  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: hello.py  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  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），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，假设需要把“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 switch 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-101  Merge made by the 'recursive' strategy.  readme.txt | 2 +-  1 file changed, 1 insertion(+), 1 deletion(-) |

现在，是时候接着回到dev分支干活了！

|  |
| --- |
| $ git switch dev  Switched 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} |

**注：**在master分支上修复了bug后，我们要想一想，dev分支是早期从master分支分出来的，所以，这个bug其实在当前dev分支上也存在。

那怎么在dev分支上修复同样的bug？重复操作一次，提交不就行了？但这里有一种更加简单的方法：

同样的bug，要在dev上修复，我们只需要把4c805e2 fix bug 101这个提交所做的修改“复制”到dev分支。注意：我们只想复制4c805e2 fix bug 101这个提交所做的修改，**并不是把整个master分支merge过来。**

为了方便操作，Git专门提供了一个cherry-pick命令，让我们能复制一个特定的提交到当前分支：

|  |
| --- |
| $ git branch  \* dev  master  $ git cherry-pick 4c805e2  [master 1d4b803] fix bug 101  1 file changed, 1 insertion(+), 1 deletion(-) |

Git自动给dev分支做了一次提交，注意这次提交的commit是1d4b803，它并不同于master的4c805e2，因为这两个commit只是改动相同，但确实是两个不同的commit。用git cherry-pick，我们就不需要在dev分支上手动再把修bug的过程重复一遍。

**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；在master分支上修复的bug，想要合并到当前dev分支，可以用git cherry-pick <commit>命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

1. Feature分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

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 switch -c feature-vulcan  Switched 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 287773e] add feature vulcan  1 file changed, 2 insertions(+)  create mode 100644 vulcan.py |

切回dev，准备合并：

|  |
| --- |
| $ git switch 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，但是这个包含机密资料的分支还是必须就地销毁：

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

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

现在我们强行删除：

|  |
| --- |
| $ git branch -D feature-vulcan  Deleted branch feature-vulcan (was 287773e). |

**小结：**

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。

1. 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：

|  |
| --- |
| $ 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。

* 1. **推送分支 push**

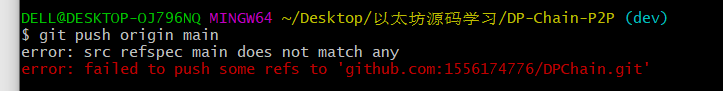
推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上(被推送的分支必须在远程主机上是存在的)：

|  |
| --- |
| $ git push origin master |

如果要推送其他分支，比如dev，就改成：

|  |
| --- |
| $ git push origin dev |

现在我们推送一个本地没有，存在于远程主机上的main分支：



如果推送一个本地有，远程主机没有的dev分支，远程主机会新创建一个dev分支

* 1. **抓取分支 fetch**

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 |

可以通过以下命令查看联系的远程origin以及本地的分支信息：

|  |
| --- |
| $ git branch -a  \* main (本地存在的分支信息)  remotes/origin/HEAD -> origin/main (main是当前指定的主分支)  remotes/origin/dev  remotes/origin/main  remotes/origin/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(首先在本地创建dev分支，然后将远程origin的dev分支clone下来)**，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 |

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

|  |
| --- |
| $ git branch --set-upstream-to=origin/dev dev  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，但是合并时出现了冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后，提交，再push：

|  |
| --- |
| 修改前的env.txt：  <<<<<<< HEAD  git pull test 2222  =======  git pull test xxxx  >>>>>>> dc51ca502a7641b9bf17da08ceef723426824fe0  修改后：  git pull test final |

|  |
| --- |
| $ 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：

* 首先，可以试图用git push origin <branch-name>推送自己的修改；
* 如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；
* 如果合并有冲突，则解决冲突，并在本地提交；
* 没有冲突或者解决掉冲突后，再用git push origin <branch-name>推送就能成功！

如果git pull提示no tracking information，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream-to <branch-name> origin/<branch-name>。

1. Rebase
2. 标签管理
3. 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 master  Switched 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-commit  12a631b (HEAD -> master, tag: v1.0, origin/master) merged bug fix 101  4c805e2 fix bug 101  e1e9c68 merge with no-ff  f52c633 add merge  cf810e4 conflict fixed  5dc6824 & simple  14096d0 AND simple  b17d20e branch test  d46f35e remove test.txt  b84166e add test.txt  519219b git tracks changes  e43a48b understand how stage works  1094adb append GPL  e475afc add distributed  eaadf4e wrote a readme file |

比方说要对add merge这次提交打标签，它对应的commit id是f52c633，敲入命令：

|  |
| --- |
| $ git tag v0.9 f52c633 |

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 f52c63349bc3c1593499807e5c8e972b82c8f286 (tag: v0.9)  Author: Michael Liao <askxuefeng@gmail.com>  Date: Fri May 18 21:56:54 2018 +0800  add merge  diff --git a/readme.txt b/readme.txt  ... |

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" 1094adb |

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: Fri May 18 22:48:43 2018 +0800  **version 0.1 released**  commit 1094adb7b9b3807259d8cb349e7df1d4d6477073 (tag: v0.1)  Author: Michael Liao <askxuefeng@gmail.com>  Date: Fri May 18 21:06:15 2018 +0800  append GPL  diff --git a/readme.txt b/readme.txt  ... |

1. 操作标签

如果标签打错了，也可以删除：

|  |
| --- |
| $ git tag -d v0.1  Deleted tag 'v0.1' (was f15b0dd) |

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 v1.0  Total 0 (delta 0), reused 0 (delta 0)  To 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  Total 0 (delta 0), reused 0 (delta 0)  To github.com:michaelliao/learngit.git  \* [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 v0.9  Deleted tag 'v0.9' (was f52c633) |

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  To github.com:michaelliao/learngit.git  - [deleted] v0.9 |

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。 |

1. 使用Github

GitHub是一个开源协作社区，通过GitHub，既可以让别人参与你的开源项目，也可以参与别人的开源项目。

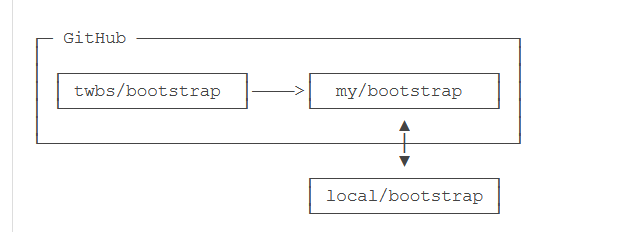
在GitHub出现以前，开源项目开源容易，但让广大人民群众参与进来比较困难，因为要参与，就要提交代码，而给每个想提交代码的群众都开一个账号那是不现实的，因此，群众也仅限于报个bug，即使能改掉bug，也只能把diff文件用邮件发过去，很不方便。

如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页https://github.com/twbs/bootstrap，**点“Fork”就在自己的账号下克隆了一个bootstrap仓库**，然后，**从自己的账号下clone：**

|  |
| --- |
| git clone git@github.com:michaelliao/bootstrap.git |

**一定要从自己的账号下clone仓库，这样你才能推送修改。如果从bootstrap的作者的仓库地址git@github.com:twbs/bootstrap.git克隆，因为没有权限，你将不能推送修改。**

Bootstrap的官方仓库twbs/bootstrap、你在GitHub上克隆的仓库my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：



如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，**往自己的仓库推送**。

如果你希望bootstrap的官方库能接受你的修改，你就可以在GitHub上**发起一个pull request**。当然，对方是否接受你的pull request就不一定了。

**小结：**

|  |
| --- |
| 在GitHub上，可以任意Fork开源仓库；  自己拥有Fork后的仓库的读写权限；  可以推送pull request给官方仓库来贡献代码。 |