



搞飞机大坏蛋

关注

总资产2

Mac上 git 的使用教程



搞飞机大坏蛋

关注

IP属地: 广东

2019.05.07 19:32:21 字数 11,595 阅读 10,181

一. 在Mac OS X上安装Git

如果你正在使用Mac做开发，有两种安装Git的方法。

一是安装homebrew，然后通过homebrew安装Git，具体方法请参考homebrew的文档：

<http://brew.sh/>。

第二种方法更简单，也是推荐的方法，就是直接从AppStore安装Xcode，Xcode集成了Git，不过默认没有安装，你需要运行Xcode，选择菜单“Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择“Command Line Tools”，点“Install”就可以完成安装了。

二.创建版本库

初始化一个Git仓库，使用 `git init` 命令。

添加文件到Git仓库，分两步：

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

使用命令 `git commit -m <message>`，完成。

三.时光机穿梭

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：

```
1 | git reset --hard commit_id
```

回退到上一版本：

```
1 | git reset --hard HEAD~1
```

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。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

```
1 | git log --pretty=oneline
```

iOS UIView用户事件响应

(exclusiveTouch，触摸响应，手

阅读 2,313

iOS 几种常用的 crash log 崩溃信息
调试方法

阅读 2,435

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。

2.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，参考版本回退一节，不过前提是没有推送到远程库。

3.删除文件

本地仓库删除了文件或者 `rm <file>`

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`：

```
1 | $ git rm test.txt
2 | rm 'test.txt'
3 |
4 | $ git commit -m "remove test.txt"
5 | [master d46f35e] remove test.txt
6 | 1 file changed, 1 deletion(-)
7 | delete mode 100644 test.txt
```

文件就从版本库中被删除了

二是删错了，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：

```
1 | $ git checkout -- test.txt
```

`git checkout` 其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“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！

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。

四.远程仓库

获得GitHub远程仓库

第1步：创建SSH Key。在用户主目录(user)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：

```
1 | $ ssh-keygen -t rsa -C "xxxxx@mail.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](#)，点击右上角头像，选择[Settings](#)，[SSH and GPG keys](#)页面；

然后，点 [New SSH Key](#)，填上任意Title，在Key文本框里粘贴id_rsa.pub文件的内容；

点 [Add SSH Key](#)，你就应该看到已经添加的Key；

```
1 | 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Git3
2 |
3 | 当然，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6
4 |
5 | 最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感7
6 |
7 | 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7
```

1.添加远程库

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+号找到[New repository](#)按钮，创建一个新的仓库；

在Repository name填入仓库名 [learn git](#)，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 [learn git](#) 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 [learn git](#) 仓库下运行命令：

```
1 | $ git remote add origin git@github.com:xxxxx/learn git.git
2 |
```

请千万注意，把上面的 [xxxxx](#) 替换成你自己的 [GitHub账户名](#)，否则，你在本地关联的就是别人的远程库，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公钥不在别人的账户列表中。

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：

```
1 | $ git push -u origin master
2 | Counting objects: 20, done.
3 | Delta compression using up to 4 threads.
4 | Compressing objects: 100% (15/15), done.
```

```
5 | Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.
6 | Total 20 (delta 5), reused 0 (delta 0)
7 | remote: Resolving deltas: 100% (5/5), done.
8 | To github.com:michaelliao/learngit.git
9 | * [new branch]      master -> master
10 | Branch 'master' set up to track remote branch 'master' from 'origin'.
11 |
```

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：

```
1 | $ 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！

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：

```
1 | The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.
2 | RSA key fingerprint is xx.xx.xx.xx.xx.
3 | 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：

```
1 | 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的指纹信息是否与SSH连接给出的一致。

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，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

2.从远程库克隆

首先，登录GitHub，创建一个新的仓库，名字叫 `gitskills`；

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：

```
1 | $ git clone git@github.com:xxxxx/gitskills.git
2 | Cloning into 'gitskills'...
3 | remote: Counting objects: 3, done.
4 | remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 3
5 | Receiving objects: 100% (3/3), done.
```

注意把Git库的地址换成你自己的，然后进入 `gitskills` 目录看看，已经有README.md文件了：

```
1 | $ cd gitskills
2 | $ ls
3 | README.md
```

```
1 | 如果有多个个人协作开发，那么每个人各自从远程克隆一份就可以了。
2 |
3 | 你也许还注意到，GitHub给出的地址不止一个，还可以用https://github.com/michaelliao/gitskills.git
4 |
5 | 使用https除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部
```

小结

```
1 | 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。
2 |
3 | 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。
```

五.分支管理

```
1 | 分支就是科幻电影里面的平行宇宙，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SV
2 |
3 | 如果两个平行宇宙互不干扰，那对现在的你也没啥影响。不过，在某个时间点，两个平行宇宙合并了，结果，你既学会
4 |
5 | 分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻
6 |
7 | 现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在
8 |
9 | 其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简
10 |
11 | 但Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是
```

1.创建与合并分支

首先，我们创建 `dev` 分支，然后切换到 `dev` 分支：

```
1 | $ git checkout -b dev
2 | 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：

```
1 | $ git branch dev
2 | $ git checkout dev
3 | Switched to branch 'dev'
```

然后，用 `git branch` 命令查看当前分支：

```
1 | $ git branch
2 | * dev
3 | 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：

```
1 | Creating a new branch is quick.
```

然后提交：

```
1 | $ git add readme.txt
2 | $ git commit -m "branch test"
3 | [dev b17d20e] branch test
4 | 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：

```
1 | $ git checkout master
2 | 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：

```
1 | $ git merge dev
2 | Updating d46f35e..b17d20e
3 | Fast-forward
4 | readme.txt | 1 +
5 | 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：

```
1 | $ git branch -d dev
2 | Deleted branch dev (was b17d20e).
```

删除后，查看 `branch`，就只剩下 `master` 分支了：

```
1 | $ git branch
2 | * 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。

小结

Git鼓励大量使用分支：

```
1 | 查看分支: git branch
2 |
3 | 创建分支: git branch <name>
4 |
5 | 切换分支: git checkout <name>
6 |
7 | 创建+切换分支: git checkout -b <name>
8 |
9 | 合并某分支到当前分支: git merge <name>
10 |
11 | 删除分支: git branch -d <name>
```

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：

```
1 | $ git checkout -b feature1
2 | Switched to a new branch 'feature1'
```

修改 `readme.txt` 最后一行，改为：

```
1 | Creating a new branch is quick AND simple.
```

在 `feature1` 分支上提交：

```
1 | $ git add readme.txt
2 | $ git commit -m "AND simple"
3 | [feature1 14096d0] AND simple
4 | 1 file changed, 1 insertion(+), 1 deletion(-)
```

切换到 `master` 分支：

```
1 | $ git checkout master
2 | Switched to branch 'master'
3 | Your branch is ahead of 'origin/master' by 1 commit.
4 | (use "git push" to publish your local commits)
```

Git还会自动提示我们当前 `master` 分支比远程的 `master` 分支要超前1个提交。

在 `master` 分支上把 `readme.txt` 文件的最后一行改为：

```
1 | Creating a new branch is quick & simple.
```

提交：

```
1 | $ git add readme.txt
2 | $ git commit -m "& simple"
3 | [master 5dc6824] & simple
4 | 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；

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：

```
1 | $ git merge feature1
2 | Auto-merging readme.txt
3 | CONFLICT (content): Merge conflict in readme.txt
4 | 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：

```
1 | $ git status
2 | On branch master
3 | Your branch is ahead of 'origin/master' by 2 commits.
4 |   (use "git push" to publish your local commits)
5 | You have unmerged paths.
6 |   (fix conflicts and run "git commit")
7 |   (use "git merge --abort" to abort the merge)
8 | Unmerged paths:
9 |   (use "git add <file>..." to mark resolution)
10 |    both modified:   readme.txt
11 | 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：

```
1 | Git is a distributed version control system.
2 | Git is free software distributed under the GPL.
3 | Git has a mutable index called stage.
4 | Git tracks changes of files.
5 | <<<<<< HEAD
6 | Creating a new branch is quick & simple.
7 | =====
8 | Creating a new branch is quick AND simple.
9 | >>>>>> feature1
```

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

```
1 | Creating a new branch is quick and simple.
```


再提交：

```
1 | $ git add readme.txt
2 | $ git commit -m "conflict fixed"
3 | [master cf810e4] conflict fixed
```

现在，`master` 分支和 `feature1` 分支变成了下图所示：

[图片上传失败...(image-bd478-1557217136484)]

用带参数的 `git log` 也可以看到分支的合并情况：

```
1 | $ git log --graph --pretty=oneline --abbrev-commit
2 | * cf810e4 (HEAD -> master) conflict fixed
3 | | \
4 | | * 14096d0 (feature1) AND simple
5 | | * | 5dc6824 & simple
6 | | | /
7 | | * b17d20e branch test
8 | | * d46f35e (origin/master) remove test.txt
9 | | * b84166e add test.txt
10 | | * 519219b git tracks changes
11 | | * e43a48b understand how stage works
12 | | * 1094adb append GPL
13 | | * e475afc add distributed
14 | | * eaadf4e wrote a readme file
```

最后，删除 `feature1` 分支：

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

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。

```
1 | git log --graph --pretty=oneline --abbrev-commit
```

3.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：

```
1 | $ git checkout -b dev
2 | Switched to a new branch 'dev'
```

修改readme.txt文件，并提交一个新的commit：

```
1 | $ git add readme.txt
2 | $ git commit -m "add merge"
3 | [dev f52c633] add merge
4 | 1 file changed, 1 insertion(+)
```

现在，我们切换回master：

```
1 | $ git checkout master
2 | Switched to branch 'master'
```

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

```
1 | $ git merge --no-ff -m "merge with no-ff" dev
2 | Merge made by the 'recursive' strategy.
3 |  readme.txt | 1 +
4 | 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：

```
1 | $ git log --graph --pretty=oneline --abbrev-commit
2 | * e1e9c68 (HEAD -> master) merge with no-ff
3 | | \
4 | | * f52c633 (dev) add merge
5 | | /
6 | * cf810e4 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。

小结

Git分支十分强大，在团队开发中应该充分应用。

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

5. 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：

```
1 | $ git status
2 | On branch dev
3 | Changes to be committed:
4 |   (use "git reset HEAD <file>..." to unstage)
5 |     new file:   hello.py
6 | Changes not staged for commit:
7 |   (use "git add <file>..." to update what will be committed)
8 |   (use "git checkout -- <file>..." to discard changes in working directory)
9 |     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 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：

```
1 | $ git stash
2 | 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：

```
1 | $ git checkout master
2 | Switched to branch 'master'
3 | Your branch is ahead of 'origin/master' by 6 commits.
4 |   (use "git push" to publish your local commits)
5 |
6 | $ git checkout -b issue-101
7 | 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：

```
1 | $ git add readme.txt
2 | $ git commit -m "fix bug 101"
3 | [issue-101 4c805e2] fix bug 101
4 | 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：

```
1 | $ git checkout master
2 | Switched to branch 'master'
3 | Your branch is ahead of 'origin/master' by 6 commits.
4 |   (use "git push" to publish your local commits)
```

```
5 |
6 | $ git merge --no-ff -m "merged bug fix 101" issue-101
7 | Merge made by the 'recursive' strategy.
8 |  readme.txt | 2 +-
9 |  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！

```
1 | $ git checkout dev
2 | Switched to branch 'dev'
3 |
4 | $ git status
5 | On branch dev
6 | 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：

```
1 | $ git stash list
2 | 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：

```
1 | $ git stash pop
2 | On branch dev
3 | Changes to be committed:
```

简书

发现 关注 会员 IT技术 消息

Aa beta 

```
7 | (use "git add <file>..." to update what will be committed)
8 | (use "git checkout -- <file>..." to discard changes in working directory)
9 | modified:   readme.txt
10 | Dropped refs/stash@{0} (5d677e2ee266f39ea296182fb2354265b91b3b2a)
```

再用 `git stash list` 查看，就看不到任何stash内容了：

```
1 |  stash list
1赞
```

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

```
1 | $ git stash apply stash@{0}
```

小结

[更多好文](#)

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。

热门故事

- 桂林志异：龙王起水
- 离婚后，妈宝男前夫后悔了
- 救了他两次的神仙让他今天三更去死
- 演金丝雀太入戏，他还真以为我爱上他了
- 为了活命，我对病娇反派弟弟表白，他竟当真要做我夫君
- “有个坐过牢的富豪老公是种什么体验？”“要不然你来试试？”
- 前世渣男把我迷晕还叫我别怕，重生后我杀疯了
- 妹妹过失杀人，警察来时，我捡起了那把滴血的刀
- 我被校霸堵在巷口，却发现他是我谈了三个月的网恋对象
- 我首富之女的身份居然被人偷了

6.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：

```
1 | $ git checkout -b feature-vulcan
2 | Switched to a new branch 'feature-vulcan'
```

5分钟后，开发完毕：

```
1 | $ git add vulcan.py
2 | $ git status
3 | On branch feature-vulcan
4 | Changes to be committed:
5 |   (use "git reset HEAD <file>..." to unstage)
6 |    new file:   vulcan.py
7 | $ git commit -m "add feature vulcan"
8 | [feature-vulcan 287773e] add feature vulcan
9 | 1 file changed, 2 insertions(+)
10 | create mode 100644 vulcan.py
```

写下你的评论...



评论0



赞1



```
1 | $ 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，但是这个包含机密资料的分支还是必须就地销毁：

```
1 | $ git branch -d feature-vulcan
2 | error: The branch 'feature-vulcan' is not fully merged.
3 | 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，需要使用大写的-D参数。。

现在我们强行删除：

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

终于删除成功！

小结

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。

7.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`：

```
1 | $ git remote
2 | origin
```

或者，用 `git remote -v` 显示更详细的信息：

```
1 | $ git remote -v
2 | origin  git@github.com:michaelliao/learngit.git (fetch)
3 | 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：

```
1 | $ git push origin master
```

如果要推送其他分支，比如 `dev`，就改成：

```
1 | $ 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：

```
1 | $ git clone git@github.com:michaelliao/learngit.git
2 | Cloning into 'learngit'...
3 | remote: Counting objects: 40, done.
4 | remote: Compressing objects: 100% (21/21), done.
5 | remote: Total 40 (delta 14), reused 40 (delta 14), pack-reused 0
6 | Receiving objects: 100% (40/40), done.
7 | 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：

```
1 | $ git branch
2 | * 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，于是他用了这个命令创建本地 **dev** 分支：

```
1 | $ 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：

```
1 | $ git add env.txt
2 |
3 | $ git commit -m "add env"
4 | [dev 7a5e5dd] add env
5 | 1 file changed, 1 insertion(+)
6 | create mode 100644 env.txt
7 |
8 | $ git push origin dev
9 | Counting objects: 3, done.
10 | Delta compression using up to 4 threads.
11 | Compressing objects: 100% (2/2), done.
12 | Writing objects: 100% (3/3), 308 bytes | 308.00 KiB/s, done.
13 | Total 3 (delta 0), reused 0 (delta 0)
14 | To github.com:michaelliao/learngit.git
15 | 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：

```
1 | $ cat env.txt
2 | env
3 |
4 | $ git add env.txt
5 |
6 | $ git commit -m "add new env"
7 | [dev 7bd91f1] add new env
8 | 1 file changed, 1 insertion(+)
9 | create mode 100644 env.txt
10 |
11 | $ git push origin dev
12 | To github.com:michaelliao/learngit.git
13 | ! [rejected] dev -> dev (non-fast-forward)
14 | error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'
```

```
15 | hint: Updates were rejected because the tip of your current branch is behind
16 | hint: its remote counterpart. Integrate the remote changes (e.g.
17 | hint: 'git pull ...') before pushing again.
18 | hint: See the 'Note about fast-forwards' in 'git push --help' for details.
19 |
```

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：

```
1 | $ git pull
2 | There is no tracking information for the current branch.
3 | Please specify which branch you want to merge with.
4 | See git-pull(1) for details.
5 |   git pull <remote> <branch>
6 | If you wish to set tracking information for this branch you can do so with:
7 |   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：

```
1 | $ git branch --set-upstream-to=origin/dev dev
2 | Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

再pull：

```
1 | $ git pull
2 | Auto-merging env.txt
3 | CONFLICT (add/add): Merge conflict in env.txt
4 | Automatic merge failed; fix conflicts and then commit the result.
5 |
```

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完全一样。解决后，提交，再push：

```
1 | $ git commit -m "fix env conflict"
2 | [dev 57c53ab] fix env conflict
3 |
4 | $ git push origin dev
5 | Counting objects: 6, done.
6 | Delta compression using up to 4 threads.
7 | Compressing objects: 100% (4/4), done.
8 | Writing objects: 100% (6/6), 621 bytes | 621.00 KiB/s, done.
9 | Total 6 (delta 0), reused 0 (delta 0)
10 | To github.com:michaelliao/learn-git.git
11 |    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。

8.Rebase

多人在同一个分支上协作时，很容易出现冲突。即使没有冲突，后push的童鞋不得不先pull，在本地合并，然后才能push成功。

每次合并再push后，分支变成了这样：

```

1 | $ git log --graph --pretty=oneline --abbrev-commit
2 | * d1be385 (HEAD -> master, origin/master) init hello
3 | * e5e69f1 Merge branch 'dev'
4 | | \
5 | | * 57c53ab (origin/dev, dev) fix env conflict
6 | | | \
7 | | | * 7a5e5dd add env
8 | | * | 7bd91f1 add new env
9 | | | /
10 | * | 12a631b merged bug fix 101
11 | | \
12 | | * | 4c805e2 fix bug 101
13 | | / /
14 | * | e1e9c68 merge with no-ff
15 | | \
16 | | | /
17 | | * f52c633 add merge
18 | | /
19 | * cf810e4 conflict fixed

```

总之看上去很乱，有强迫症的童鞋会问：为什么Git的提交历史不能是一条干净的直线？

其实是可以做到的！

Git有一种称为 `rebase` 的操作，有人把它翻译成“变基”。

先不要随意展开想象。我们还是从实际问题出发，看看怎么把分叉的提交变成直线。

在和远程分支同步后，我们对 `hello.py` 这个文件做了两次提交。用 `git log` 命令看看：

```
1 | $ git log --graph --pretty=oneline --abbrev-commit
2 | * 582d922 (HEAD -> master) add author
3 | * 8875536 add comment
4 | * d1be385 (origin/master) init hello
5 | * e5e69f1 Merge branch 'dev'
6 | | \
7 | | * 57c53ab (origin/dev, dev) fix env conflict
8 | | | \
9 | | | * 7a5e5dd add env
10 | | * | 7bd91f1 add new env
11 | ...
```

注意到Git用 `(HEAD -> master)` 和 `(origin/master)` 标识出当前分支的HEAD和远程origin的位置，分别是 `582d922 add author` 和 `d1be385 init hello`，本地分支比远程分支快两个提交。

现在我们尝试推送本地分支：

```
1 | $ git push origin master
2 | To github.com:michaelliao/learngit.git
3 | ! [rejected]        master -> master (fetch first)
4 | error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'
5 | hint: Updates were rejected because the remote contains work that you do
6 | hint: not have locally. This is usually caused by another repository pushing
7 | hint: to the same ref. You may want to first integrate the remote changes
8 | hint: (e.g., 'git pull ...') before pushing again.
9 | hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

很不幸，失败了，这说明有人先于我们推送了远程分支。按照经验，先pull一下：

```
1 | $ git pull
2 | remote: Counting objects: 3, done.
3 | remote: Compressing objects: 100% (1/1), done.
4 | remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
5 | Unpacking objects: 100% (3/3), done.
6 | From github.com:michaelliao/learngit
7 |   d1be385..f005ed4  master    -> origin/master
8 |   * [new tag]         v1.0     -> v1.0
9 | Auto-merging hello.py
10 | Merge made by the 'recursive' strategy.
11 |   hello.py | 1 +
12 |   1 file changed, 1 insertion(+)
```

再用 `git status` 看看状态：

```
1 | $ git status
2 | On branch master
3 | Your branch is ahead of 'origin/master' by 3 commits.
4 |   (use "git push" to publish your local commits)
5 |
6 | nothing to commit, working tree clean
```

加上刚才合并的提交，现在我们本地分支比远程分支超前3个提交。

用 `git log` 看看：

```
1 | $ git log --graph --pretty=oneline --abbrev-commit
2 | * e0ea545 (HEAD -> master) Merge branch 'master' of github.com:michaelliao/learngit
3 | | \
4 | | * f005ed4 (origin/master) set exit=1
5 | * | 582d922 add author
6 | * | 8875536 add comment
7 | | /
8 | * d1be385 init hello
9 | ...
```

对强迫症童鞋来说，现在事情有点不对头，提交历史分叉了。如果现在把本地分支push到远程，有没有问题？

有！

什么问题？

不好看！

有没有解决方法？

有！

这个时候，rebase就派上了用场。我们输入命令 `git rebase` 试试：

```
1 | $ git rebase
2 | First, rewinding head to replay your work on top of it...
3 | Applying: add comment
4 | Using index info to reconstruct a base tree...
5 | M   hello.py
6 | Falling back to patching base and 3-way merge...
7 | Auto-merging hello.py
8 | Applying: add author
9 | Using index info to reconstruct a base tree...
10 | M   hello.py
11 | Falling back to patching base and 3-way merge...
12 | Auto-merging hello.py
```

输出了一大堆操作，到底是啥效果？再用 `git log` 看看：

```
1 | $ git log --graph --pretty=oneline --abbrev-commit
2 | * 7e61ed4 (HEAD -> master) add author
3 | * 3611cfe add comment
4 | * f005ed4 (origin/master) set exit=1
5 | * d1be385 init hello
6 | ...
```

原本分叉的提交现在变成一条直线了！这种神奇的操作是怎么实现的？其实原理非常简单。我们注意观察，发现Git把我们本地的提交“挪动”了位置，放到了 `f005ed4 (origin/master) set exit=1` 之后，这样，整个提交历史就成了一条直线。rebase操作前后，最终的提交内容是一致的，但是，我们本地的commit修改内容已经变化了，它们的修改不再基于 `d1be385 init hello`，而是基于 `f005ed4 (origin/master) set exit=1`，但最后的提交 `7e61ed4` 内容是一致的。

这就是rebase操作的特点：把分叉的提交历史“整理”成一条直线，看上去更直观。缺点是本地的分叉提交已经被修改过了。

最后，通过push操作把本地分支推送到远程：

```
1 Mac:~/learngit michael$ git push origin master
2 Counting objects: 6, done.
3 Delta compression using up to 4 threads.
4 Compressing objects: 100% (5/5), done.
5 Writing objects: 100% (6/6), 576 bytes | 576.00 KiB/s, done.
6 Total 6 (delta 2), reused 0 (delta 0)
7 remote: Resolving deltas: 100% (2/2), completed with 1 local object.
8 To github.com:michaelliao/learngit.git
9    f005ed4..7e61ed4  master -> master
```

再用 `git log` 看看效果：

```
1 $ git log --graph --pretty=oneline --abbrev-commit
2 * 7e61ed4 (HEAD -> master, origin/master) add author
3 * 3611cfe add comment
4 * f005ed4 set exit=1
5 * d1be385 init hello
6 ...
```

远程分支的提交历史也是一条直线。

小结

- rebase操作可以把本地未push的分叉提交历史整理成直线；
- rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比。

六.标签管理

```
1 发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么
2
3 Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标
4
5 Git有commit，为什么还要引入tag？
6
7 “请把上周一的那个版本打包发布，commit号是6a5819e...”
8
9 “一串乱七八糟的数字不好找！”
10
11 如果换一个办法：
12
13 “请把上周一的那个版本打包发布，版本号是v1.2”
14
15 “好的，按照tag v1.2查找commit就行！”
16
17 所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。
```

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：

```
1 $ git branch
2 * dev
3   master
4 $ git checkout master
5 Switched to branch 'master'
```

然后，敲命令`git tag <name>`就可以打一个新标签：

```
1 | $ git tag v1.0
```

可以用命令`git tag`查看所有标签：

```
1 | $ git tag
2 | 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：

```
1 | $ git log --pretty=oneline --abbrev-commit
2 | 12a631b (HEAD -> master, tag: v1.0, origin/master) merged bug fix 101
3 | 4c805e2 fix bug 101
4 | e1e9c68 merge with no-ff
5 | f52c633 add merge
6 | cf810e4 conflict fixed
7 | 5dc6824 & simple
8 | 14096d0 AND simple
9 | b17d20e branch test
10 | d46f35e remove test.txt
11 | b84166e add test.txt
12 | 519219b git tracks changes
13 | e43a48b understand how stage works
14 | 1094adb append GPL
15 | e475afc add distributed
16 | ead4f4e 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：

```
1 | $ git tag v0.9 f52c633
```

再用命令`git tag`查看标签：

```
1 | $ git tag
2 | v0.9
3 | 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：

```
1 | $ git show v0.9
2 | commit f52c63349bc3c1593499807e5c8e972b82c8f286 (tag: v0.9)
3 | Author: Michael Liao <askxuefeng@gmail.com>
4 | Date: Fri May 18 21:56:54 2018 +0800
5 |
6 |     add merge
7 |
8 | diff --git a/readme.txt b/readme.txt
9 | ...
```

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：

```
1 | $ 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：

```
1 | $ git show v0.1
2 | tag v0.1
3 | Tagger: Michael Liao <askxuefeng@gmail.com>
4 | Date:   Fri May 18 22:48:43 2018 +0800
5 |
6 | version 0.1 released
7 |
8 | commit 1094adb7b9b3807259d8cb349e7df1d4d6477073 (tag: v0.1)
9 | Author: Michael Liao <askxuefeng@gmail.com>
10 | Date:   Fri May 18 21:06:15 2018 +0800
11 |
12 |     append GPL
13 |
14 | diff --git a/readme.txt b/readme.txt
15 | ...
```

注意：标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签。

小结

命令 `git tag <tagname>` 用于新建一个标签，默认为HEAD，也可以指定一个commit id；

命令 `git tag -a <tagname> -m "blablabla..."` 可以指定标签信息；

命令 `git tag` 可以查看所有标签。

2.操作标签

如果标签打错了，也可以删除：

```
1 | $ git tag -d v0.1
2 | 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>：

```
1 | $ git push origin v1.0
2 | Total 0 (delta 0), reused 0 (delta 0)
3 | To github.com:michaelliao/learn-git.git
4 | * [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：

```
1 | $ git push origin --tags
2 | Total 0 (delta 0), reused 0 (delta 0)
3 | To github.com:michaelliao/learn-git.git
4 | * [new tag]          v0.9 -> v0.9
```

```
1 | $ git tag -d v0.9
2 | Deleted tag 'v0.9' (was f52c633)
```

```
1 $ git push origin :refs/tags/v0.9
2 To github.com:michaelliao/learngit.git
3 - [deleted]          v0.9
```

小结

命令 `git push origin :refs/tags/<tagname>` 可以删除一个远程标签。

七.使用GitHub

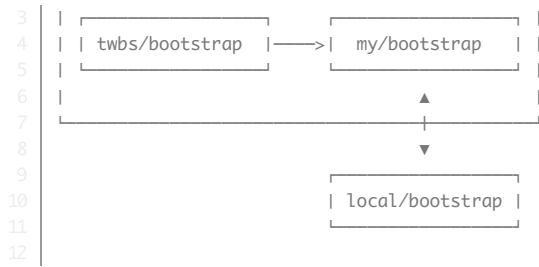
如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页<https://github.com/twbs/bootstrap>，点“Fork”就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：

```
1 | 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 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。

如果你没能力修改bootstrap，但又想要试一把pull request，那就Fork一下我的仓库：

<https://github.com/ZWP233/learnngit>，创建一个 `your-github-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，我会视心情而定是否接受。

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。

八.使用码云

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 \(gitee.com\)](https://gitee.com)。

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learnngit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：

```
1 | git remote add origin git@gitee.com:liaoxxuefeng/learnngit.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：

```
1 | git remote add origin git@gitee.com:liaoxxuefeng/learnngit.git
2 | fatal: 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：

```
1 | git remote -v
2 | origin git@github.com:michaelliao/learnngit.git (fetch)
3 | origin git@github.com:michaelliao/learnngit.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：

```
1 | 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）：

```
1 | git remote add origin git@gitee.com:liaoxxuefeng/learnngit.git
```

此时，我们再查看远程库信息：

```
1 | git remote -v
2 | origin git@gitee.com:liaoxxuefeng/learnngit.git (fetch)
3 | origin git@gitee.com:liaoxxuefeng/learnngit.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 `learnngit` 本地库为例，我们先删除已关联的名为 `origin` 的远程库：

```
1 | git remote rm origin
```

然后，先关联GitHub的远程库：

```
1 | 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：

```
1 | git remote add gitee git@gitee.com:liaoxxuefeng/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：

```
1 | git remote -v
2 | gitee git@gitee.com:liaoxxuefeng/learngit.git (fetch)
3 | gitee git@gitee.com:liaoxxuefeng/learngit.git (push)
4 | github git@github.com:michaelliao/learngit.git (fetch)
5 | github git@github.com:michaelliao/learngit.git (push)
```

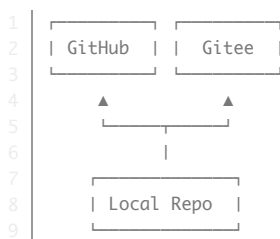
如果要推送到GitHub，使用命令：

```
1 | git push github master
```

如果要推送到码云，使用命令：

```
1 | 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/ZWP233/learngit>，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。

九.自定义Git

在安装Git一节中，我们已经配置了 `user.name` 和 `user.email`，实际上，Git还有很多可配置项。

比如，让Git显示颜色，会让命令输出看起来更醒目：

```
1 | $ git config --global color.ui true
```

这样，Git会适当地显示不同的颜色，比如 `git status` 命令；

文件名就会标上颜色。

我们在后面还会介绍如何更好地配置Git，以便让你的工作更高效。

1.忽略特殊文件

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

好在Git考虑到了大家的感受，这个问题解决起来也很简单，在Git工作区的根目录下创建一个特殊的 `.gitignore` 文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

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

忽略文件的原则是：

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

举个例子：

假设你在Windows下进行Python开发，Windows会自动在有图片的目录下生成隐藏的缩略图文件，如果有自定义目录，目录下就会有 `Desktop.ini` 文件，因此你需要忽略Windows自动生成的垃圾文件：

```
1 | # Windows:
2 | Thumbs.db
3 | ehthumbs.db
4 | Desktop.ini
```

然后，继续忽略Python编译产生的 `.pyc`、`.pyo`、`dist` 等文件或目录：

```
1 | # Python:
2 | *.py[cod]
3 | *.so
4 | *.egg
5 | *.egg-info
6 | dist
7 | build
```

加上你自己定义的文件，最终得到一个完整的 `.gitignore` 文件，内容如下：

```
1 | # Windows:
2 | Thumbs.db
3 | ehthumbs.db
4 | Desktop.ini
5 |
6 | # Python:
7 | *.py[cod]
8 | *.so
9 | *.egg
10 | *.egg-info
11 | dist
12 | build
13 |
14 | # My configurations:
15 | db.ini
16 | deploy_key_rsa
```

最后一步就是把 `.gitignore` 也提交到Git，就完成了！当然检验 `.gitignore` 的标准是 `git status` 命令是不是说 `working directory clean`。

使用Windows的童鞋注意了，如果你在资源管理器里新建一个 `.gitignore` 文件，它会非常弱智地提示你必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为 `.gitignore` 了。

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

```
1 | $ git add App.class
2 | The following paths are ignored by one of your .gitignore files:
3 | App.class
4 | Use -f if you really want to add them.
```

如果你确实想添加该文件，可以用 `-f` 强制添加到Git：

```
1 | $ git add -f App.class
```

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

```
1 | $ git check-ignore -v App.class
2 | .gitignore:3:*.class App.class
```

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

小结

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

2.配置别名

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

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

我们只需要敲一行命令，告诉Git，以后 `st` 就表示 `status`：

```
1 | $ 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`：

```
1 | $ git config --global alias.co checkout
2 | $ git config --global alias.ci commit
3 | $ git config --global alias.br branch
```

以后提交就可以简写成：

```
1 | $ 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` 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：

```
1 | $ git config --global alias.unstage 'reset HEAD'
```

当你敲入命令：

```
1 | $ git unstage test.py
```

实际上Git执行的是：

```
1 | $ git reset HEAD test.py
```

配置一个 `git last`，让其显示最后一次提交信息：

```
1 | $ git config --global alias.last 'log -1'
```

这样，用 `git last` 就能显示最近一次的提交：

```
1 | $ git last
2 | commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2
3 | Merge: bd6ae48 291bea8
4 | Author: Michael Liao <askxuefeng@gmail.com>
5 | Date: Thu Aug 22 22:49:22 2013 +0800
6 | merge & fix hello.py
```

甚至还有人丧心病狂地把 `lg` 配置成了：

```
1 | git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%CC
```

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：

```
1 | $ cat .git/config
2 | [core]
3 |     repositoryformatversion = 0
4 |     filemode = true
5 |     bare = false
6 |     logallrefupdates = true
7 |     ignorecase = true
8 |     precomposeunicode = true
9 | [remote "origin"]
10 |     url = git@github.com:michaelliao/learngit.git
11 |     fetch = +refs/heads/*:refs/remotes/origin/*
12 | [branch "master"]
13 |     remote = origin
14 |     merge = refs/heads/master
15 | [alias]
16 |     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：

```
1 | $ cat .gitconfig
2 | [alias]
3 |     co = checkout
4 |     ci = commit
5 |     br = branch
6 |     st = status
7 | [user]
8 |     name = Your Name
9 |     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。

小结

给Git配置好别名，就可以输入命令时偷个懒。我们鼓励偷懒。

3.搭建Git服务器

在远程仓库一节中，我们讲了远程仓库实际上和本地仓库没啥不同，纯粹为了7x24小时开机并交换大家的修改。

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

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

假设你已经有 `sudo` 权限的用户账号，下面，正式开始安装。

第一步，安装 `git`：

```
1 | $ sudo apt-get install git
```

第二步，创建一个 `git` 用户，用来运行 `git` 服务：

```
1 | $ sudo 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：

```
1 | $ sudo 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`：

```
1 | $ sudo 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：

```
1 | git:x:1001:1001:,,,:/home/git:/bin/bash
```

改为：

```
1 | 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：

```
1 | $ git clone git@server:/srv/sample.git
2 | Cloning into 'sample'...
3 | 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来管理公钥。

这里我们不介绍怎么玩Gitosis了，几百号人的团队基本都在500强了，相信找个高水平的Linux管理员问题不大。

管理权限

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

这里我们也不介绍Gitolite了，不要把有限的生命浪费到权限斗争中。

小结

- 搭建Git服务器非常简单，通常10分钟即可完成；
- 要方便管理公钥，用Gitosis；
- 要像SVN那样变态地控制权限，用Gitolite。

git

最后编辑于：2019.05.08 15:41:51

©著作权归作者所有,转载或内容合作请联系作者




更多精彩内容，就在简书APP



"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下



搞飞机大坏蛋


咸鱼如果没有梦想, 那还是闲鱼吗?

总资产2 共写了1.9W字 获得16个赞 共2个粉丝

关注


人面猴

序言：七十年代末，一起剥皮案震惊了整个滨河市，随后出现的几起案子，更是在滨河造成了极大的恐慌，老刑警刘岩，带你破解...

 沈念sama 阅读 148,637 评论 1 赞 318


死咒

序言：滨河连续发生了三起死亡事件，死亡现场离奇诡异，居然都是意外死亡，警方通过查阅死者的电脑和手机，发现死者居然都...

 沈念sama 阅读 63,443 评论 1 赞 266


救了他两次的神仙让他今天三更去死

文/潘晓璐 我一进店门，熙熙楼的掌柜王于贵愁眉苦脸地迎上来，“玉大人，你说我怎么就摊上这事。” “怎么了？”我有些...

 开封第一讲书人 阅读 99,164 评论 0 赞 218


道士缉凶录：失踪的卖姜人

文/不坏的土叔 我叫张陵，是天一观的道长。 经常有香客问我，道长，这世上最难降的妖魔是什么？ 我笑而不...

 开封第一讲书人 阅读 42,075 评论 0 赞 188


港岛之恋（遗憾婚礼）

正文 为了忘掉前任，我火速办了婚礼，结果婚礼上，老公的妹妹穿的比我还像新娘。我一直安慰自己，他们只是感情好，可当我...

 茶点故事 阅读 50,080 评论 1 赞 266


恶毒庶女顶嫁案：这布局不是一般人想出来的

文/花漫 我一把揭开白布。她就那样静静地躺着，像睡着了一般。火红的嫁衣衬着肌肤如雪。梳的纹丝不乱的头发上，一...

 开封第一讲书人 阅读 39,365 评论 1 赞 184


城市分裂传说

那天，我揣着相机与录音，去河边找鬼。笑死，一个胖子当着我的面吹牛，可吹牛的内容都是我干的。我是一名探鬼主播，决...

 沈念sama 阅读 30,901 评论 2 赞 283

双鸳鸯连环套：你想象不到人心有多黑

文/苍兰香墨 我猛地睁开眼，长吁一口气：“原来是场噩梦啊.....” “哼！你这毒妇竟也来了？” 一声冷哼从身侧响起，我...

 开封第一讲书人 阅读 29,649 评论 0 赞 176

万荣杀人案实录

序言：老挝万荣一对情侣失踪，失踪者是张志新（化名）和其女友刘颖，没想到半个月后，有当地人在树林里发现了一具尸体，经...

 沈念sama 阅读 33,122 评论 0 赞 223

护林员之死

正文 独居荒郊野岭守林人离奇死亡，尸身上长有42处带血的脓包..... 初始之章·张勋 以下内容为张勋视角 年9月15日...

 茶点故事 阅读 29,734 评论 2 赞 225

白月光启示录

正文 我和宋清朗相恋三年，在试婚纱的时候发现自己被绿了。 大学时的朋友给我发了我未婚夫和他白月光在一起吃饭的照片。...

 茶点故事 阅读 31,093 评论 1 赞 236


活死人

序言：一个原本活蹦乱跳的男人离奇死亡，死状恐怖，灵堂内的尸体忽然破棺而出，到底是诈尸还是另有隐情，我是刑警宁泽，带...

 沈念sama 阅读 27,548 评论 2 赞 222

日本核电站爆炸内幕

正文 年R本政府宣布，位于F岛的核电站，受9级特大地震影响，放射性物质发生泄漏。R本人自食恶果不足惜，却给世界环境...

 茶点故事 阅读 32,028 评论 3 赞 216

男人毒药：我在死后第九天来索命

文/蒙蒙 一、第九天 我趴在偏房一处隐蔽的房顶上张望。院中可真热闹，春花似锦、人声如沸。这庄子的主人今日做“春日...

 开封第一讲书人 阅读 25,765 评论 0 赞 9

一桩弑父案，背后竟有这般阴谋

文/苍兰香墨 我抬头看了看天上的太阳。三九已至，却和暖如春，着一层夹袄步出监牢的瞬间，已是汗流浹背。一阵脚步声响...

 开封第一讲书人 阅读 26,291 评论 0 赞 178

情欲美人皮

我被黑心中介骗来泰国打工， 没想到刚下飞机就差点儿被人妖公主榨干..... 1. 我叫王不留，地道东北人。 一个月前我还...

 沈念sama 阅读 34,162 评论 2 赞 239

代替公主和亲

正文 我出身青楼，却偏偏与公主长得像，于是被迫代替她去往敌国和亲。 传闻我的和亲对象是个残疾皇子，可洞房花烛夜当晚...

 茶点故事 阅读 34,293 评论 2 赞 242



写下你的评论...

全部评论 0 只看作者

按时间倒序 按时间正序

被以下专题收入，发现更多相似内容

+ 收入我的专题 iOS开发

推荐阅读

[更多精彩内容 >](#)

Git命令学习笔记

1. 安装 Github 查看是否安装git: `$ git config --global user.name "...`

Albert_Sun 阅读 13,362 评论 9 赞 163

Git使用教程

(预警：因为详细，所以行文有些长，新手边看边操作效果出乎你的预料) 一：Git是什么？ Git是目前世界上最先进的...

axiaochao 阅读 1,899 评论 1 赞 8

Git使用

声明：这篇文章来源于廖雪峰老师的官方网站，我仅仅是作为学习之用 Git简介 Git是什么？ Git是目前世界上最先...

横渡 阅读 3,916 评论 3 赞 27

git详细使用教程入门到精通(史上最全的git教程)

还是老规矩，这篇看完后，还是学不会git版本控制的，你来砍我 是兄弟就来砍我吧!!! Git是分布式版本控制系统，...

Zteen 阅读 3,042 评论 0 赞 6

Git学习总结——简单易懂的教程

安装Git Git的下载地址：Git官网下载地址 Git本地仓库和命令 配置用户 下载完Git后，右键会有一个Gi...

TokyoZ 阅读 4,426 评论 1 赞 7