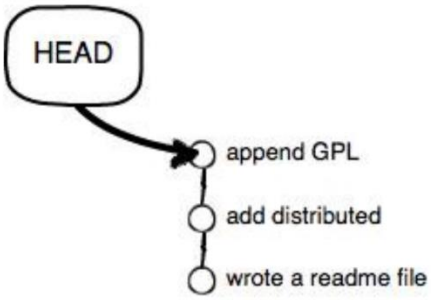
1. git add readme.txt (添加到仓库)
2. git commit -m “create a readme file” (提交到仓库)
3. git status (随时检查仓库状态)
4. type readme.txt (查看文件内容)
5. git diff readme.txt (显示修改内容)

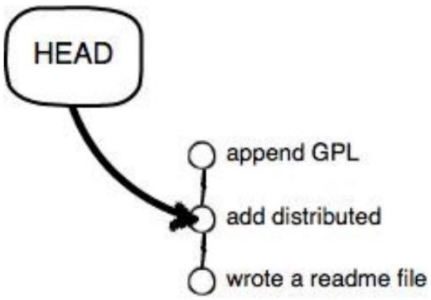
   - 要随时掌握工作区的状态，使用`git status`命令。
   - 如果`git status`告诉你有文件被修改过，用`git diff`可以查看修改内容。

6. git log (显示从最近到最远的提交日志)
   git log --pretty=oneline (以行显示)
7. git reset --hard HEAD~1 (退回去一个版本，HEAD~n：退回去 n 个版本。HEAD 指向当前版本)
   若想重新回到新版本，需要 git reset --hard 1094a(版本号)

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。

8. git reflog 记录每一次命令，可用于查找版本号

```
D:\mygit>git reflog
a38a25c (HEAD -> master) HEAD@{0}: reset: moving to HEAD~1
1bb040a HEAD@{1}: commit: change file1
a38a25c (HEAD -> master) HEAD@{2}: commit: add distributed
fedd123 HEAD@{3}: commit: add file1
8783c29 HEAD@{4}: commit (initial): wrote a readme file
```

9. `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

https://www.liaoxuefeng.com/wiki/001373951630592960dd18361248578c67b8067c8c017b000/0013745374151782eb658c5a5ca454eaa451661275886c6000

10. git diff VS git diff HEAD:
    - 当暂存区中没有文件时，git diff 比较的是，工作区中的文件与上次提交到版本库中的文件。
    - 当暂存区中有文件时，git diff 则比较的是，当前工作区中的文件与暂存区中的文件。

    而 git diff HEAD -- file，比较的是工作区中的文件与版本库中文件的差异。HEAD 指向的是版本库中的当前版本，而 file 指的是当前工作区中的文件。

    补充：git diff 命令比较的是工作目录中当前文件与暂存区快照之间的差异，也就是修改之后还没有暂存起来的变化内容。

    Example:

    1) 修改 readme.txt，在最后一行加上 third change.
    2) 执行 git diff readme.txt: (比较工作区和上次提交的文件)

```
D:\mygit>git diff readme.txt
diff --git a/readme.txt b/readme.txt
index 0c52ba2..bd5fa19 100644
--- a/readme.txt
+++ b/readme.txt
@@ -3,4 +3,5 @@ Git is a free software.
 Git has a mutable index called stage.
 Git tracks changes of files.
 first change.
-second change.
\ No newline at end of file
+second change.
+third change.
\ No newline at end of file
```

    3) 执行 git add readme.txt，再执行 git diff readme.txt(比较工作区和暂存区文件)

```
D:\mygit>git add readme.txt

D:\mygit>git diff readme.txt
```

    4) 执行 git diff HEAD -- readme.txt (比较工作区和版本库文件的差异)

```
D:\mygit>git diff HEAD -- readme.txt
diff --git a/readme.txt b/readme.txt
index 0c52ba2..bd5fa19 100644
--- a/readme.txt
+++ b/readme.txt
@@ -3,4 +3,5 @@ Git is a free software.
 Git has a mutable index called stage.
 Git tracks changes of files.
 first change.
-second change.
\ No newline at end of file
+second change.
+third change.
\ No newline at end of file
```

11. 若修改了文件，撤销修改:

    git **checkout** -- readme.txt

    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。

12. 使用 git reset HEAD readme.txt 可以把暂存区的修改退回到工作区，HEAD 表示最新版本，然后再将工作区的修改撤销 (git checkout -- readme.txt):

```
D:\mygit>git add readme.txt

D:\mygit>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   readme.txt


D:\mygit>git reset HEAD readme.txt
Unstaged changes after reset:
M       readme.txt

D:\mygit>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   readme.txt

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

D:\mygit>git checkout -- readme.txt

D:\mygit>git status
On branch master
nothing to commit, working tree clean
```

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。

13. del test.txt (删除文件)
从版本库中删除文件，两步：
① git rm test.txt (删除) ②git commit -m "remove test.txt" (提交删除)

```
D:\mygit>git rm test.txt
rm 'test.txt'

D:\mygit>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    test.txt


D:\mygit>git commit -m "remove test.txt"
[master 3329920] remove test.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 test.txt
```

删除 untracked file: git clean -fdx

14. 如果误删，版本库还有，可以恢复：
    git checkout -- test.txt
    若 add 后 commit 了，版本库中已经没有了，用 checkout 也无法恢复，只能使用 git reset --hard 1094a(版本号)恢复到最新版本，但会丢失最后一次提交后你修改的内容。
15. 添加远程库：
    git remote add origin https://github.com/EuniceF/learngit.git
    git push -u origin master
    origin:远程库

    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。现在远程的 learngit 就和本地的 mygit 内容一模一样。

```
D:\mygit>git remote add origin https://github.com/EuniceF/learngit.git

D:\mygit>git push -u origin master
Counting objects: 26, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (26/26), 2.14 KiB | 168.00 KiB/s, done.
Total 26 (delta 8), reused 0 (delta 0)
remote: Resolving deltas: 100% (8/8), done.
To https://github.com/EuniceF/learngit.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

    从现在起只要本地做了**提交**，就可以通过命令:(要先 **add**，再 **commit**,再输命令)
    git push origin master　把本地 master 分支的最新修改推送至 GitHub。

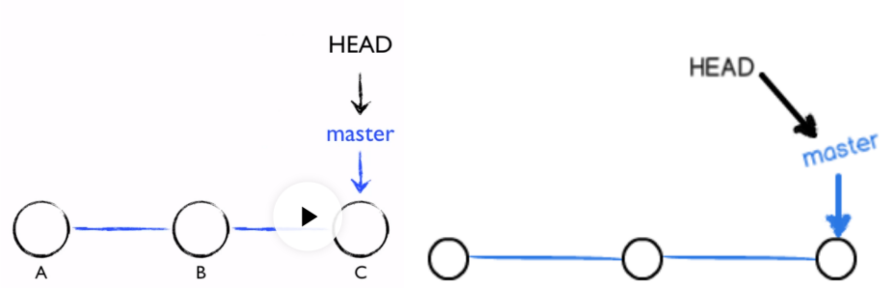
```
D:\mygit>git push origin master
Everything up-to-date
```

16. 克隆远程到本地：（通过 https 或者换 ssh）
    通过 https：git clone https://github.com/EuniceF/gitskills.git
    通过 ssh：git clone git@github.com:EuniceF/gitskills.git
    dir：查看当前目录内容
17. 合并与分支：

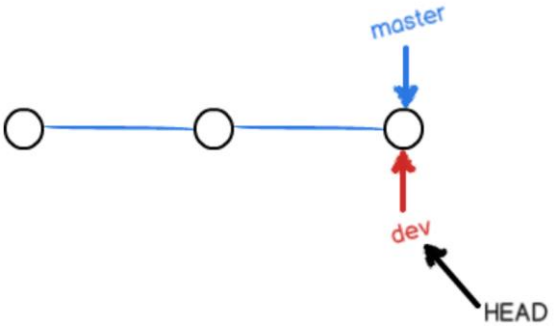
https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001375840038939c291467cc7c747b1810aab2fb8863508000

一开始，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。

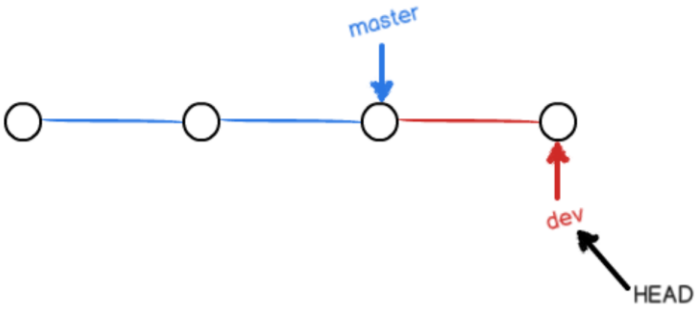
git commit -m “commit C”之后，如下图，master 指向 C



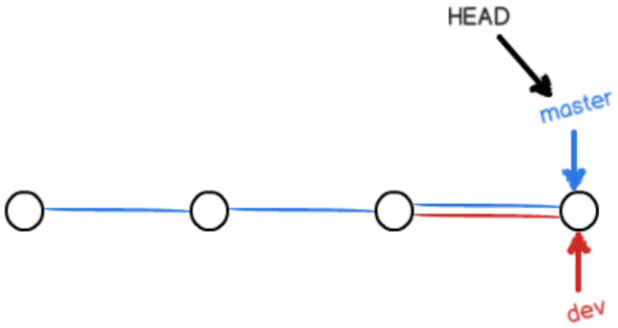
如果创建新的分支 dev 时，Git 新建一个指针叫 dev，指向 master 相同的提交，再把 HEAD 指向 dev，就表示当前分支在 dev 上。



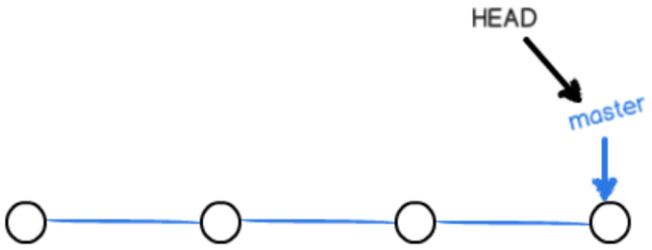
从现在开始，对工作区的修改和提交就是针对 dev 分支了，比如新提交一次后，dev 指针往前移动一步，而 master 指针不变。



假如我们在 dev 上的工作完成了，就可以把 dev 合并到 master 上，直接把 master 指向 dev 的当前提交。



合并完分支后，可以删除 dev 分支，就是把 dev 指针删掉，只剩下 master 一条分支：



18. 分支与合并（实战）：

创建 dev 分支，切换到 dev 分支。在 git checkout 命令加上-b 参数表示创建并切换，相当于 git branch dev; git checkout dev 两条命令。

```
D:\mygit>git checkout -b dev
Switched to a new branch 'dev'
```

用 git branch 查看所有分支（当前分支前会标注*）：

```
D:\mygit>git branch
* dev
  master
```

修改 readme.txt，然后提交：

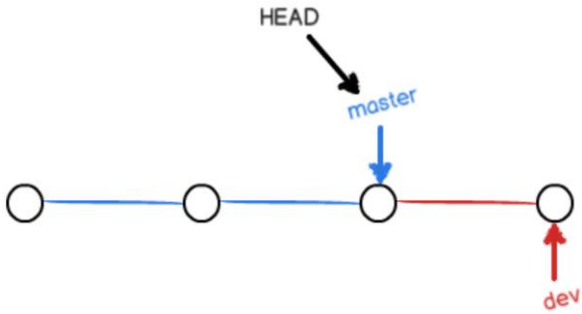
```
D:\mygit>git add readme.txt
```

```
D:\mygit>git commit -m "branch test"
[dev 91a13e5] branch test
 1 file changed, 2 insertions(+), 1 deletion(-)
```

现在 dev 分支的工作完成，我们可以切换回 master 分支：**git checkout** master

```
D:\mygit>git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

切换到 master 后，会发现 readme.txt 修改的内容不见了。因为刚刚提交是在 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

```
D:\mygit>git merge dev
Updating ad3376a..91a13e5
Fast-forward
 readme.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

合并后查看 readme.txt，就和 dev 分支最新提交是完全一样的。
合并完成后，可以删除 dev 分支了：git branch -d dev

```
D:\mygit>git branch -d dev
Deleted branch dev (was 91a13e5).
```

这时查看 branch，只有 master:

```
D:\mygit>git branch
* master
```

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>`

19. 冲突解决:
    如果在两个分支上对同一个文件进行了修改，想要合并这两个分支，会产生冲突。先创建 feature1 分支，修改文件，add&commit，再切换到 master 分支，修改文件，add&commit，现在合并 feature1:

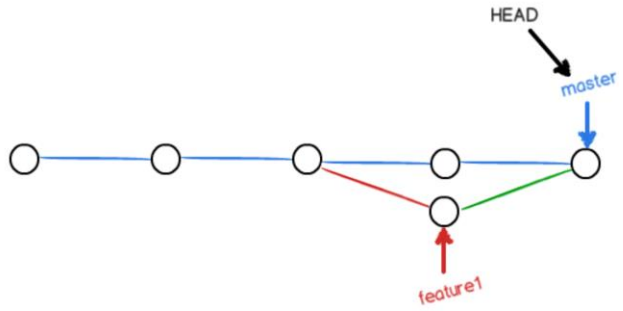
```
D:\mygit>git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

查看 readme.txt 内容:

```
D:\mygit>type readme.txt
Git is a distributed version control system.
Git is a free software.
Git has a mutable index called stage.
Git tracks changes of files.
first change.
second change.
<<<<<<< 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

再次修改文件内容，然后 add & commit，冲突解决。现在 master 和 feature1 分支变成如下



git log --graph --pretty=oneline --abbrev-commit 可以看到分支合并图:

```
D:\mygit>git log --graph --pretty=oneline --abbrev-commit
*   3f8dc65 (HEAD -> master) conflict fixed
|\
| * aedd469 (feature1) AND simple
* | 77dfc36 & simple
|/
* 91a13e5 branch test
* ad3376a (origin/master) add a new local file
* 217d3de add test.txt
* 7426d17 commit first and second changes
* 73e1cd6 second commit
* 71e3c1f git tracks changes
* 8503823 understand how stage works
* a38a25c add distributed
* fedd123 add file1
* 8783c29 wrote a readme file
```

20. 禁用 fast forward 模式:

    合并 dev 分支: git merge **--no-ff -m** "merge with no-ff" **dev**

    --no-ff 表示禁用 fast forward； 因为要创建一个新的 commit，所以加上-m 参数，把 commit 描述写进去。

    然后显示分支图:

```
$ git log --graph --pretty=oneline --abbrev-commit
*   2ad7857 (HEAD -> master) merge with no-ff
|\
| * 2674939 (dev) add merge
|/
*   7b1b435 fixed conflicts
|\
```

分支策略

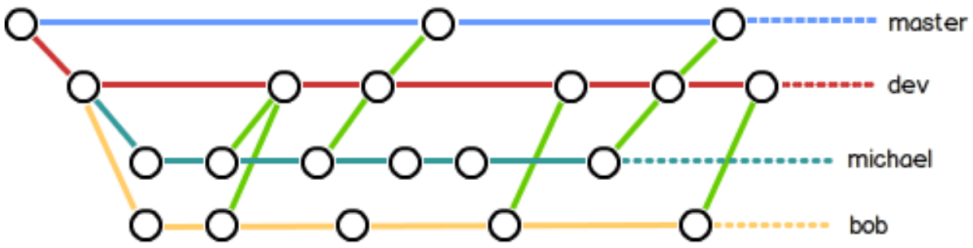
在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，`master`分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在`dev`分支上，也就是说，`dev`分支是不稳定的，到某个时候，比如1.0版本发布时，再把`dev`分支合并到`master`上，在`master`分支发布1.0版本；

你和你的小伙伴们每个人都在`dev`分支上干活，每个人都有自己的分支，时不时地往`dev`分支上合并就可以了。

所以，团队合作的分支看起来就像这样：



小结

Git分支十分强大，在团队开发中应该充分应用。

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

21. Bug 分支:
如果你现在在 dev 上进行工作，但现在需要修复一个 bug，在 master 分支上，你 dev 的工作未完成，所以不能切换到 master 分支，除非 dev 的 git status 是干净的或者用 stash 存储当前工作现场。
如果在当前 dev 上进行的工作还没有提交，可以用 stash，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: 90f125b commit dev
```

用 git status 查看工作区是干净的，可以开始创建分支来修复 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 issue-101:

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git checkout -b issue-101
Switched to a new branch 'issue-101'
```

对 readme.txt 文件进行修改，add & commit，然年切换到 master 分支，merge & delete issue-101 branch.
然后接着回到 dev 分支干活。查看工作现场：**git stash list**

```
$ git stash list
stash@{0}: WIP on dev: 90f125b commit dev
```

两个恢复工作现场的方法：

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 apply stash@{0}
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


Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git stash list
stash@{0}: WIP on dev: 90f125b commit dev
```

用 git stash list 查看，stash 内容并未删除。

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} (fde5284b4c9473f0cf01826de6bc738a93cc0b6c)
```

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。

22. 开发一个新 feature，最好新建一个分支。
    若分支 feature 在未合并前试图删除，则会出现错误：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ 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'.
```

    根据提示信息，我们需要使用大写 D 强行删除：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git branch -D feature-vulcan
Deleted branch feature-vulcan (was 4a4640d).
```

23. 当你从远程仓库 clone 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。
    查看远程库的信息，用 **git remote (git remote -v 显示更详细信息)**
    显示可以抓取和推送的 origin 的地址，如果没有推送权限，就看不到 push 地址

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git remote
origin

Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git remote -v
origin  https://github.com/EuniceF/learngit.git (fetch)
origin  https://github.com/EuniceF/learngit.git (push)
```

24. 推送分支(push):
    把该分支上所有本地提交推送到远程库，推送时要**指定本地分支**。

```
$ git push origin master
Counting objects: 45, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (43/43), done.
Writing objects: 100% (45/45), 4.11 KiB | 95.00 KiB/s, done.
Total 45 (delta 28), reused 0 (delta 0)
remote: Resolving deltas: 100% (28/28), completed with 1 local object.
To https://github.com/EuniceF/learngit.git
   ad3376a..fe8000f  master -> master
```

    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。

25. 抓取分支（fetch）
    连接远程仓库，必须先 git push -u origin master，这样才能用远程库里的 branch。
    在 newgit 目录下创建远程的 origin/dev 分支到本地，用以下命令创建本地 dev:

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/newgit (master)
$ git checkout -b dev origin/dev
Switched to a new branch 'dev'
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

    对本地文件修改，add & commit，然后 push 到远程：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/newgit (dev)
$ 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), 318 bytes | 106.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/EuniceF/learngit.git
   745aa0c..bb537c0  dev -> dev
```

    在另一目录下(mygit)，对同一文件进行了修改，并 push，会出现错误：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git push origin dev
To https://github.com/EuniceF/learngit.git
 ! [rejected]        dev -> dev (fetch first)
error: failed to push some refs to 'https://github.com/EuniceF/learngit.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

想要 pull 远程 origin/dev 的内容到本地 dev，要先建立本地 dev 和远程 origin/dev 的连接：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git branch --set-upstream-to=origin/dev dev
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

然后 git pull，但是会出现冲突，因为双方都修改了文件。此时就本地解决冲突，然后 add & commit，再 push 到 remote 就可以了。

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/EuniceF/learngit
   bb537c0..f33cc8f  dev        -> origin/dev
Auto-merging newtest.txt
CONFLICT (content): Merge conflict in newtest.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev|MERGING)
$ cat newtest.txt
newgit file
make some changes
<<<<<<< HEAD
mygit changes
=======
newgit changes
>>>>>>> f33cc8f1c5e22226b12de5c0e9fab71cf01dd4fd
```

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (dev)
$ git push origin dev
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 629 bytes | 125.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/EuniceF/learngit.git
   f33cc8f..e1c08e8  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>`。

- 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。

26. rebase

git rebase 可以把本地未 push 的分叉提交历史整理成直线，查看历史提交的变化时会更容易。

27. 创建标签：

切换到要创建标签的分支，然后：`git tag <name>`

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git tag v1.0
```

`git tag` 查看所有标签：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git tag
v1.0
```

默认标签是打在最新提交的 commit 上的，若之前应该打标签但未打，则需要先找到历史提交的 commit id，然后打上标签就可以了：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git log --pretty=oneline --abbrev-commit
fe8000f (HEAD -> master, tag: v1.0, origin/master) fix bug 101
fce3509 Merge branch 'bob' merge branch bob
1451f6b Merge branch 'dev' merge dev branch
90f125b commit dev
dd75fb8 commit bob
2ad7857 merge with no-ff
2674939 add merge
```

比如要对 add merge 这次提交打标签，对应 commit id 为 2674939：

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git tag v0.9 2674939

Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git tag
v0.9
v1.0
```

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

```
$ git show v0.9
commit 2674939c9f6958b61e8948cad2568cb3ebfc0395 (tag: v0.9)
Author: DESKTOP-2KRTUGC\Eunice <15221092161@163.com>
Date:   Wed Aug 29 13:43:29 2018 +1000

    add merge

diff --git a/readme.txt b/readme.txt
```

创建带有说明的标签，`-a` 指定标签名，`-m` 指定说明文字：

```
$ git tag -a v0.1 -m "verstion 0.1 released" 3f8dc65
```

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，则在这两个分支上都可以看到这个标签。

- 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。

28. 操作标签：

删除标签：`$ git tag -d v0.1`

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git tag -d v0.1
Deleted tag 'v0.1' (was 8ef2430)
```

推送标签到远程：`git push origin <tagname>`

```
$ git push origin v1.0
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/EuniceF/learngit.git
 * [new tag]         v1.0 -> v1.0
```

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 https://github.com/EuniceF/learngit.git
 * [new tag]         v0.9 -> v0.9
```

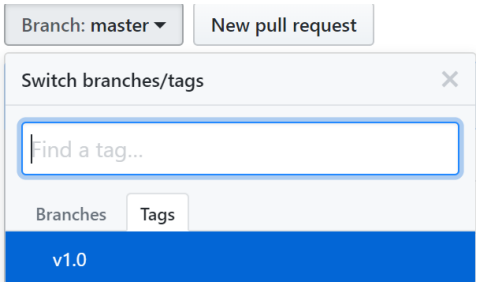
如果标签已经推送到远程，要删除远程标签，首先从本地删除：

```
$ git tag -d v0.9
Deleted tag 'v0.9' (was 2674939)
```

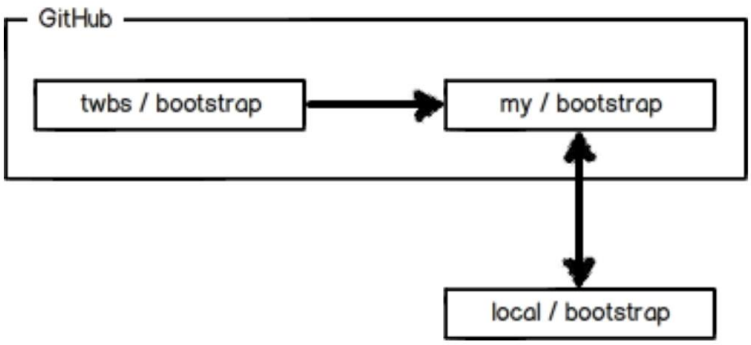
然后从远程删除：

```
$ git push origin :refs/tags/v0.9
To https://github.com/EuniceF/learngit.git
 - [deleted]         v0.9
```

查看是否删除成功，登录 GitHub：



29. 使用 GitHub：



twbs/bootstrap 是官方仓库，本地没有修改权限，必须先 fork 到自己的 GitHub 远程仓库，然后再 clone 到本地修改，再 push 修改到自己远程仓库。如果想要给官方仓库贡献代码，则需要推送 pull request 给官方仓库。

30. 配置别名：

    用 st 表示 status, co 表示 checkout，ci 表示 commit，br 表示 branch

```
Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git config --global alias.st status

Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git st
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git config --global alias.co checkout

Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git config --global alias.ci commit

Eunice@DESKTOP-2KRTUGC MINGW64 /d/mygit (master)
$ git config --global alias.br branch
```

    把撤销修改 reset HEAD 设置为 unstage 别名：

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

    配置 `git last`，让其显示最后一次提交信息：

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