Git 教程

By 廖雪峰

1. Install GIT (on Ubuntu)

```
sudo apt-get install git
git config --global user.name "Your Name"
git config --global user.email email@example.com
Other systems: skipped
```

2. Git Commands:

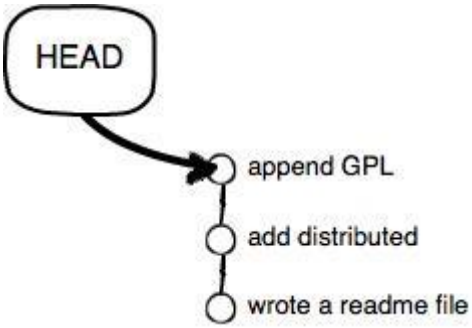
- 初始化一个 Git 仓库，使用 `git init` 命令。
- 添加文件到 Git 仓库，分两步：
  - 第一步，使用命令 `git add <file>`，注意，可反复多次使用，添加多个文件；
  - 第二步，使用命令 `git commit`，完成。
- 要随时掌握工作区的状态，使用 `git status` 命令。
- 如果 git status 告诉你有文件被修改过，用 `git diff` 可以查看修改内容。E.g. git diff readme.txt
- Time Machine:
  - HEAD 指向的版本就是当前版本，因此，Git 允许我们在版本的历史之间穿梭，使用命令 `git reset --hard commit_id`。

    首先，Git 必须知道当前版本是哪个版本，在 Git 中，用 HEAD 表示当前版本，也就是最新的提交 3628164...882e1e0（注意我的提交 ID 和你的肯定不一样），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.

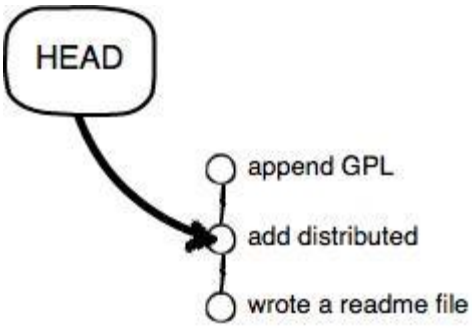
    ```
    $ git reset --hard HEAD^
    HEAD is now at ea34578 add distributed

    $ git reset --hard 3628164
    HEAD is now at 3628164 append GPL
    ```

    版本号没必要写全，前几位就可以了，Git 会自动去找。当然也不能只写前一两位，因为 Git 可能会找到多个版本号，就无法确定是哪一个了。

    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：



- 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。

  ```
  $ git log --pretty=oneline
  3628164fb26d48395383f8f31179f24e0882e1e0 append GPL
  ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed
  cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file
  ```

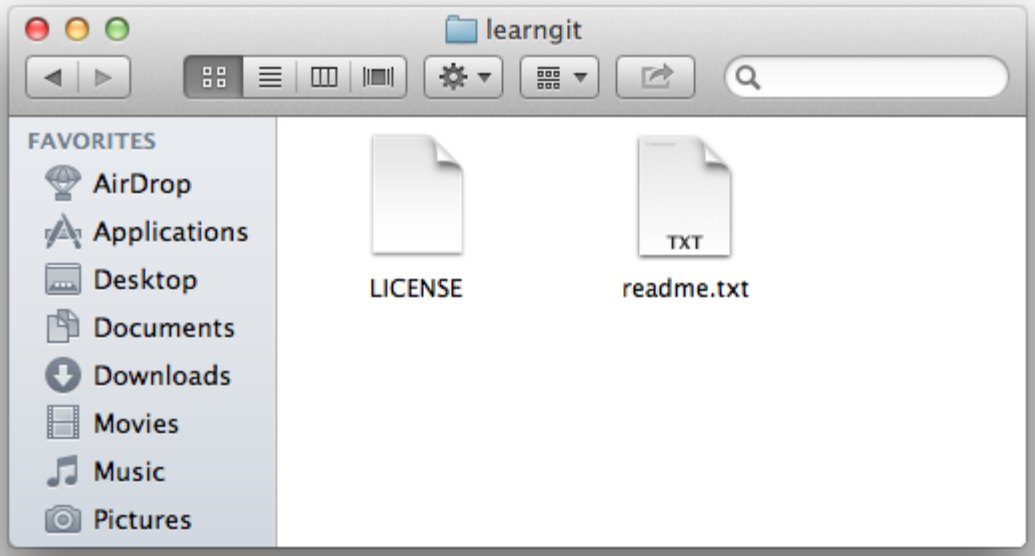
- 要重返未来，用 `git reflog` 查看命令历史，以便确定要回到未来的哪个版本。

  ```
  $ git reflog
  ea34578 HEAD@{0}: reset: moving to HEAD^
  3628164 HEAD@{1}: commit: append GPL
  ea34578 HEAD@{2}: commit: add distributed
  cb926e7 HEAD@{3}: commit (initial): wrote a readme file
  ```

3. 工作区和暂存区

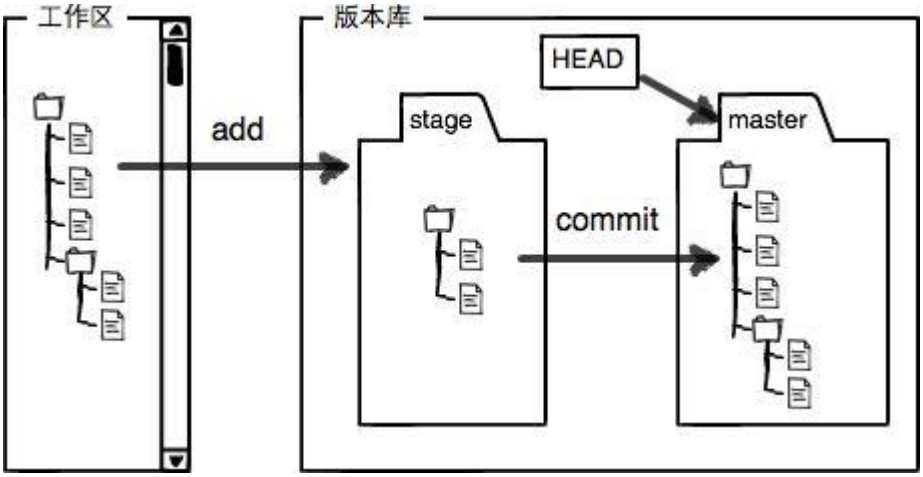
   工作区（Working Directory）

   就是你在电脑里能看到的目录，比如我的 learngit 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。
因为我们创建 Git 版本库时，Git 自动为我们创建了唯一一个 master 分支，所以，现在，git commit 就是往 master 分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

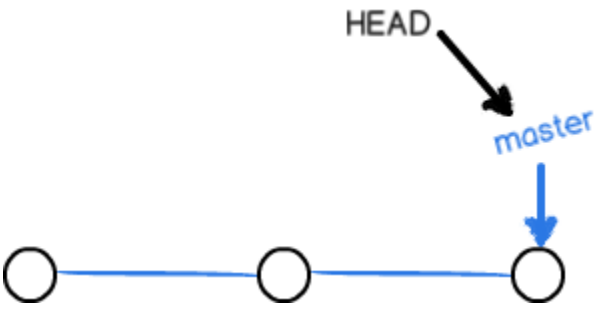
4. 场景 1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令 `git checkout -- file`。

   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。

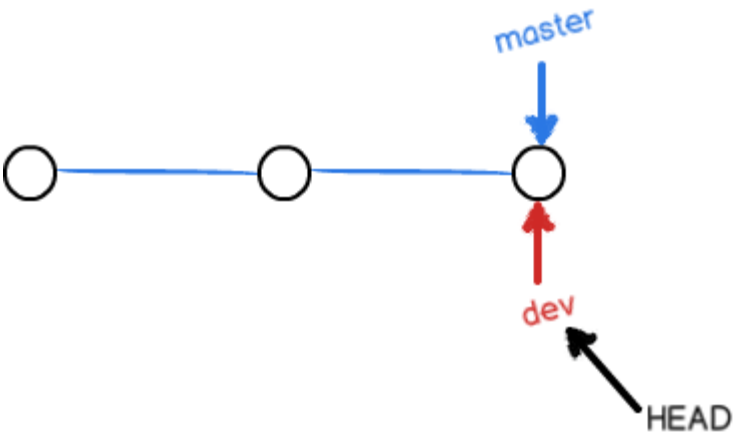
   场景 3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

5. More commands:
   - 命令 `git rm` 用于删除一个文件。
   - 要克隆一个仓库，首先必须知道仓库的地址，然后使用 `git clone` 命令克隆。
     Git 支持多种协议，包括 https，但通过 ssh 支持的原生 git 协议速度最快

6. Branching
   一开始的时候，master 分支是一条线，Git 用 master 指向最新的提交，再用 HEAD 指向 master，就能确定当前分支，以及当前分支的提交点：

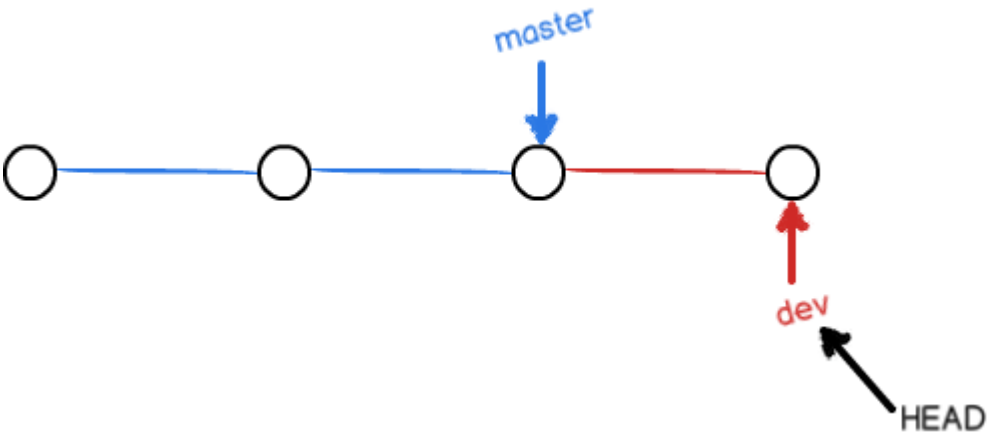


每次提交，master 分支都会向前移动一步，这样，随着你不断提交，master 分支的线也越来越长：
当我们创建新的分支，例如 dev 时，Git 新建了一个指针叫 dev，指向 master 相同的提交，再把 HEAD 指向 dev，就表示当前分支在 dev 上：

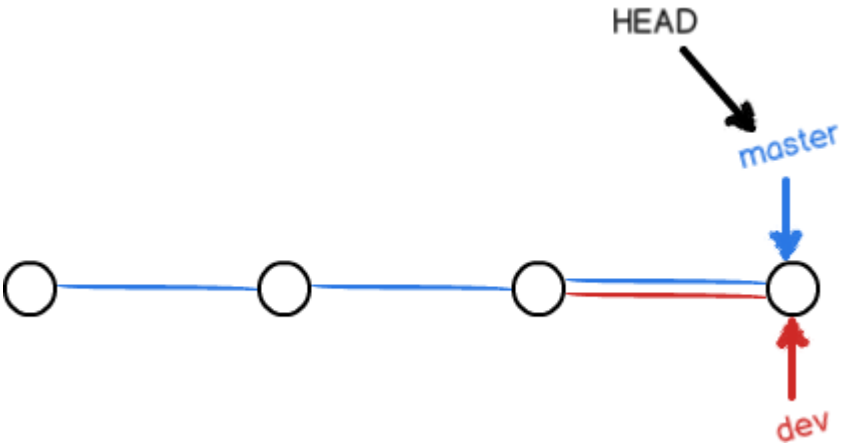


你看，Git 创建一个分支很快，因为除了增加一个 dev 指针，改改 HEAD 的指向，工作区的文件都没有任何变化！

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

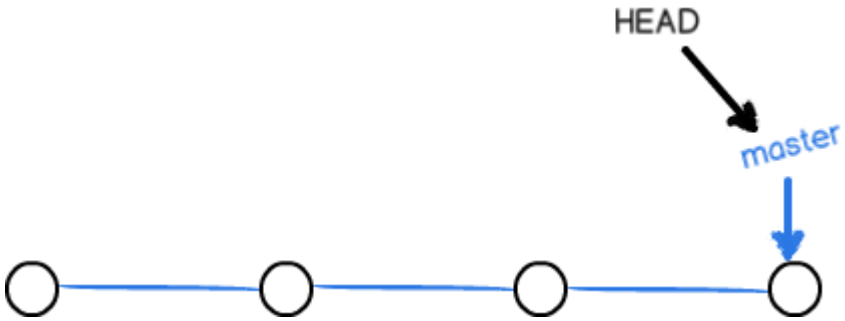


假如我们在 dev 上的工作完成了，就可以把 dev 合并到 master 上。Git 怎么合并呢？最简单的方法，就是直接把 master 指向 dev 的当前提交，就完成了合并：



所以 Git 合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除 dev 分支。删除 dev 分支就是把 dev 指针给删掉，删掉后，我们就剩下了一条 master 分支：



下面开始实战。

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：

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

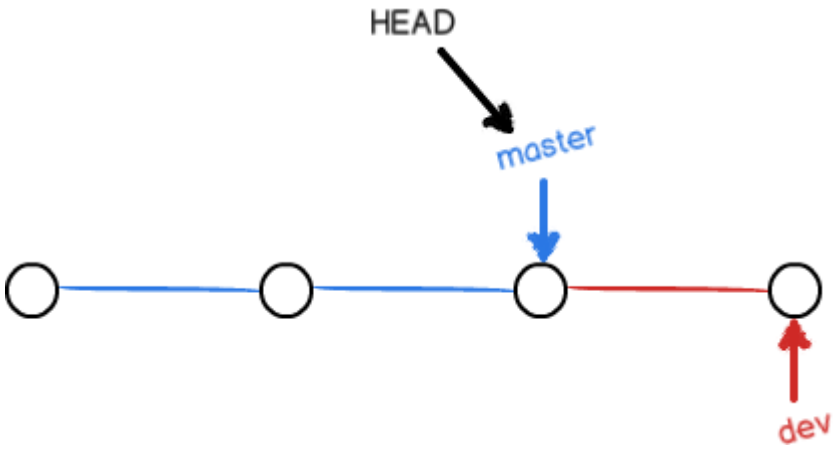
然后提交：

```
$ git add readme.txt
$ git commit -m "branch test"
[dev fec145a] branch test
 1 file changed, 1 insertion(+)
```

现在，dev 分支的工作完成，我们就可以切换回 master 分支：

```
$ git checkout 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 d17efd8..fec145a
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，我们后面会将其他方式的合并。

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 fec145a).
```

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。

Summary:

查看分支：`git branch`

创建分支：`git branch <name>`

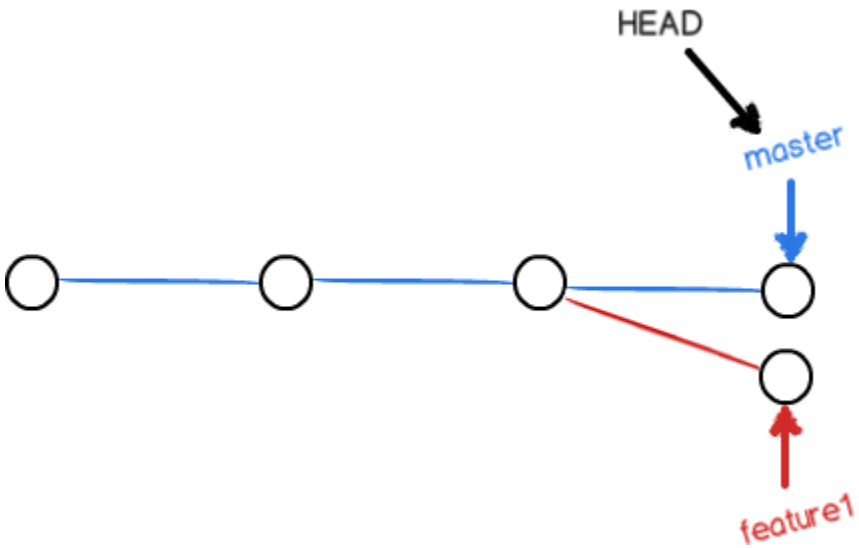
切换分支：`git checkout <name>`

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>`

7. 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.

   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.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      readme.txt
```

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

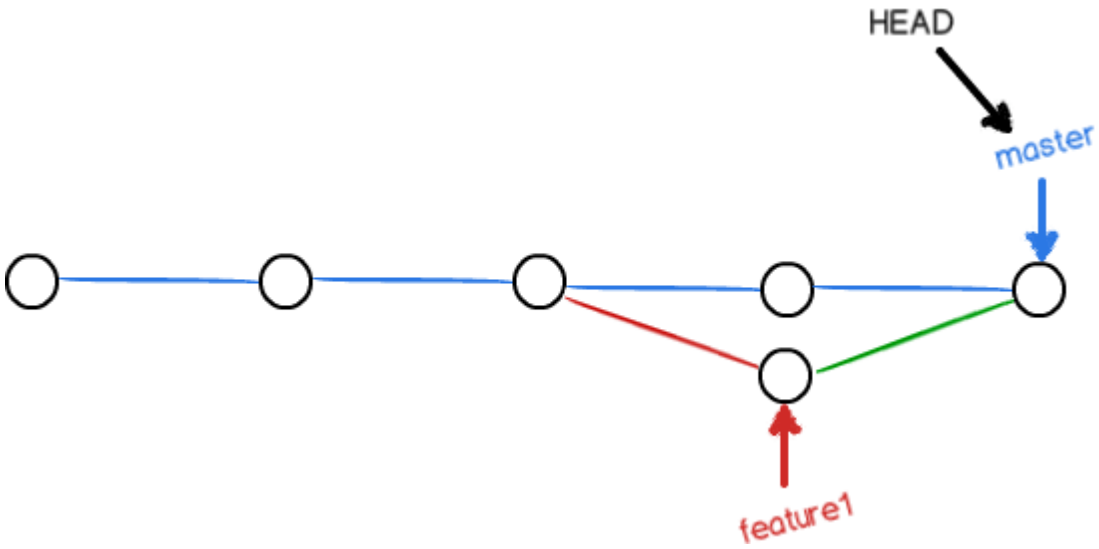
Git 用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：

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

再提交：

```
$ git add readme.txt
$ git commit -m "conflict fixed"
[master 59bc1cb] conflict fixed
```

现在，master 分支和 feature1 分支变成了下图所示：



用带参数的 git log 也可以看到分支的合并情况：

```
$ git log --graph --pretty=oneline --abbrev-commit
*   59bc1cb conflict fixed
|\
| * 75a857c AND simple
* | 400b400 & simple
|/
* fec145a branch test
...
```

最后，删除 feature1 分支：

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

工作完成。

用`git log --graph`命令可以看到分支合并图

8. 分支策略

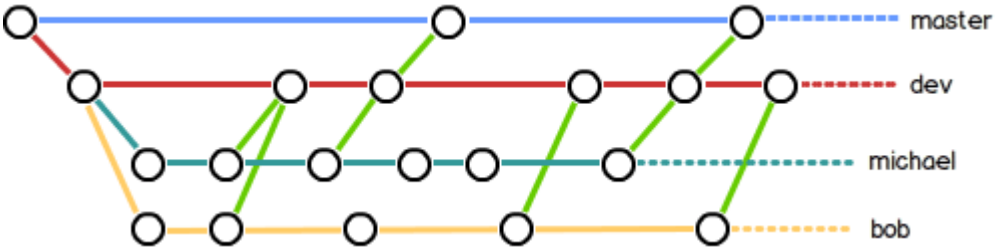
在实际开发中，我们应该按照几个基本原则进行分支管理：

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：

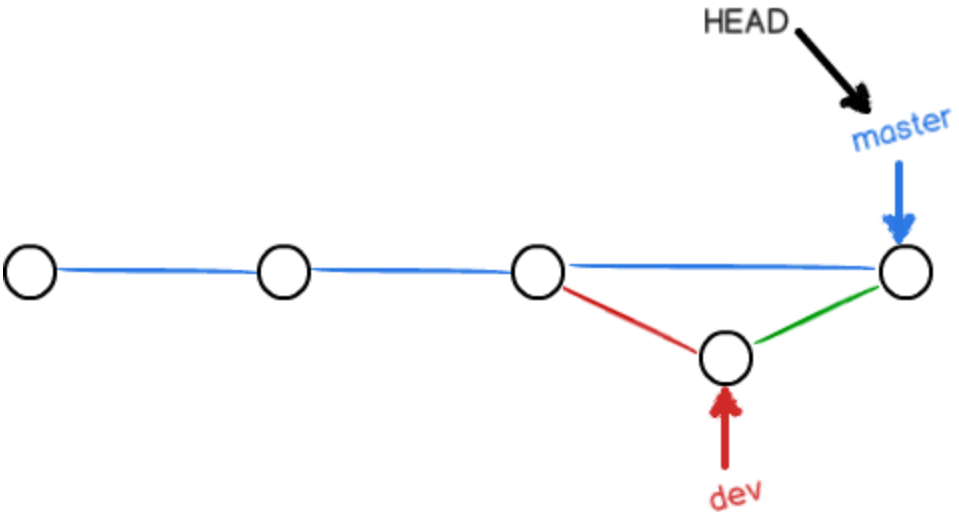


小结

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。

```
$ git merge --no-ff -m "merge with no-ff" dev
```



9. Bug Branch

   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。

10. Feature Brancgh

    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。

11. Remote

    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。

12. Tag
    - 命令 `git tag <name>`用于新建一个标签，默认为 `HEAD`，也可以指定一个 commit id；
    - `git tag -a <tagname> -m "blablabla..."`可以指定标签信息；
    - `git tag -s <tagname> -m "blablabla..."`可以用 PGP 签名标签；
    - 命令 `git tag` 可以查看所有标签。
    - 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；
    - 删除一个本地标签后，用命令 `git push origin :refs/tags/<tagname>`可以删除一个远程标签。
13. GitHub
    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。
14. Customize Git

    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！
    Alias:

```
$ cat .gitconfig
[alias]
    co = checkout
    ci = commit
    br = branch
    st = status
[user]
    name = Your Name
    email = your@email.com
```

15. Create Git Server

- 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。