Git Concept Handbook

Basic

Working Directory

在本地文件系统中实际看到和操作的文件和文件夹所在的区域，所有的操作，最初修改都是在工作区域体现的。

Stage Area

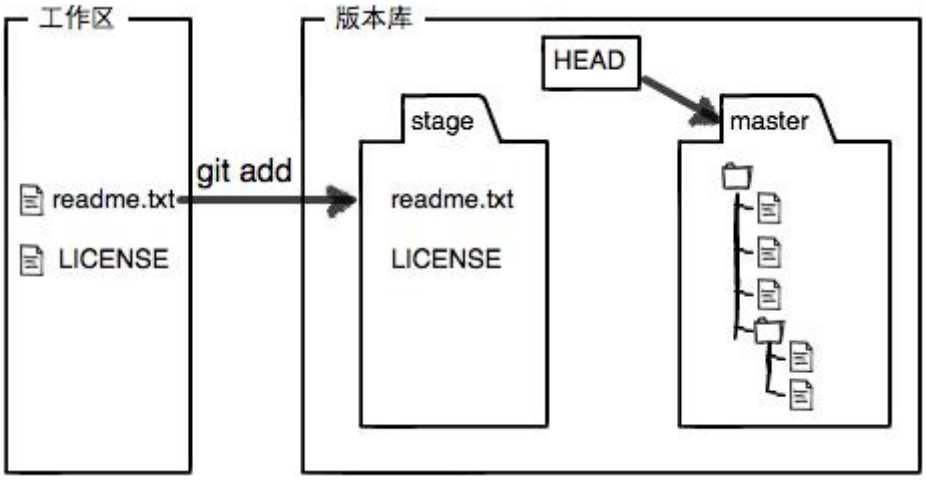
当一个文件被修改（添加）之后，可以将修改过的文件添加进暂存区，此时工作目录中的修改就会被复制一份（从概念上理解）到暂存区，等待被提交到本地仓库，相当于把想要提交的修改先放进一个购物车，然后在想提交的时候结账付款。

Local Repository

每次提交，都会将暂存区内容永久性的保存到本地仓库，本地仓库存储了项目的完整版本历史，每个提交都是项目在某个特定时间点的快照，一旦提交完成，暂存区就会被清空（准备下一轮的添加和提交），而工作目录中的文件仍然保持修改后的状态，可以继续在工作目录中进行新的修改，并重复这个添加到暂存区和提交到本地仓库的过程。

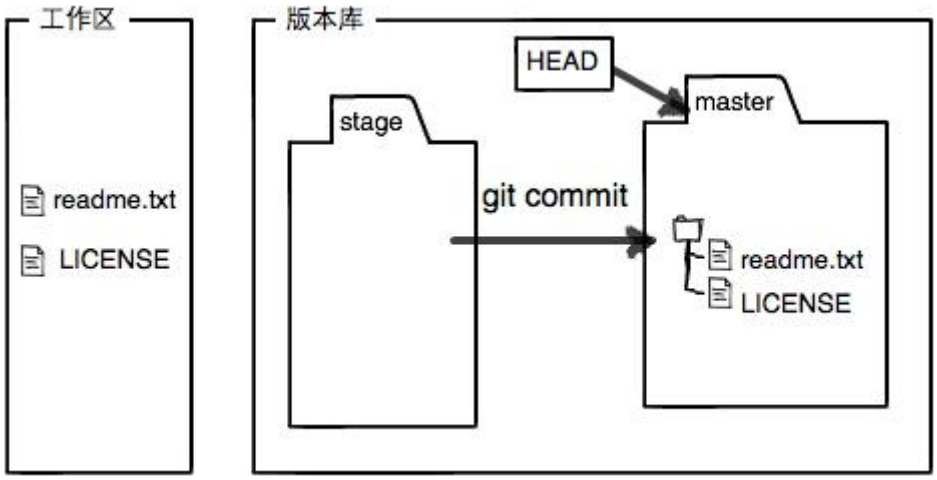
Add

`git add`：将文件修改添加到暂存区



Commit

`git commit`：文件修改到一定程度可以保存一次快照，这种行为称作`commit`，`git commit`只负责把暂存区的修改提交，未放进暂存区的代码不受`commit`控制。



Log

`git log`：显示最近到最远的提交日志

`git reflog`：显示最近到最远记录的每一次命令

如果版本已经退回并且无法通过终端历史命令行和`log`找到退回ID之后的ID，可以使用`reflog`来找到每次`commit`对应的历史命令，并在历史命令中找到ID

下面演示了从最新版本退回到`HEAD^`之后，再用`reflog`查找`HEAD^`之后的版本（在`log`中找不到）

```
PS D:\project\learn-git> git log
commit c7fd9214d7175af0e25c2c3dcc4cb4a522272d93 (HEAD -> master)
Author: UserName <niezshan@outlook.com>
Date:   Wed Dec 4 22:42:08 2024 +0800

    add distributed

commit ede7da9dd80cc3a6a92062529e480d7e87b3dfa7
Author: UserName <niezshan@outlook.com>
Date:   Wed Dec 4 22:37:13 2024 +0800

    commit 3 files

commit 17dc1170224332e1380366eda8dd856ee59f76be
Author: UserName <niezshan@outlook.com>
Date:   Wed Dec 4 22:35:59 2024 +0800

    write a readme file: readme.txt
PS D:\project\learn-git> git reflog
c7fd921 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
176a961 HEAD@{1}: commit: add GPL
c7fd921 (HEAD -> master) HEAD@{2}: commit: add distributed
ede7da9 HEAD@{3}: commit: commit 3 files
17dc117 HEAD@{4}: commit (initial): write a readme file: readme.txt
PS D:\project\learn-git> git reset --hard 176a
HEAD is now at 176a961 add GPL
PS D:\project\learn-git> git log
commit 176a961d3fc426ce9084aa8be5152bf088379ed8 (HEAD -> master)
Author: UserName <niezshan@outlook.com>
Date:   Wed Dec 4 22:44:47 2024 +0800

    add GPL

commit c7fd9214d7175af0e25c2c3dcc4cb4a522272d93
Author: UserName <niezshan@outlook.com>
Date:   Wed Dec 4 22:42:08 2024 +0800

    add distributed

commit ede7da9dd80cc3a6a92062529e480d7e87b3dfa7
Author: UserName <niezshan@outlook.com>
Date:   Wed Dec 4 22:37:13 2024 +0800

    commit 3 files

commit 17dc1170224332e1380366eda8dd856ee59f76be
Author: UserName <niezshan@outlook.com>
Date:   Wed Dec 4 22:35:59 2024 +0800
```

```
49
50      write a readme file: readme.txt
```

HEAD

每次`commit`会有一个对应的ID：`commit e475afc93c209a690c39c13a46716e8fa000c366`

但是这种ID难记住或输入，所以git用`HEAD`指向当前分支的当前版本，也就是当前分支最新的提交ID。

类似地，上一个版本是`HEAD^`，上上一个版本是`HEAD^^`，当然往上N个版本写N个`^`比较容易数不过来，所以写成`HEAD~N`。

Revocate

命令`git checkout -- readme.txt`意思就是，把`filename`文件在工作区的修改全部撤销，这里有两种情况：

一种是文件修改后还没有被放到暂存区，此时，撤销操作会使工作区回到和`HEAD`版本一模一样的状态（回到最近一次`commit`状态）；

一种是已经添加到暂存区后，又作了修改，此时，撤销操作会使工作区回到添加到暂存区后的状态（回到最近一次`add`状态）。

撤销`add`操作：

用命令`git reset HEAD <filename>`可以把暂存区的修改撤销掉（unstage），git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。此时，被存入暂存区的文件被移除暂存区回到工作区，此时，在工作区依然能看到`add`操作之前的修改，此时再使用`checkout`操作，将文件区的修改撤销至最近的一次`commit`状态。

Remove

在Git中，删除也是一个修改操作，一般情况下，可以直接在文件管理器中删除文件，或者用rm命令。删除之后，Git会知晓此次删除，因此，工作区和版本库就不一致了，`git status`命令会显示哪些文件被删除了：

`git rm <filename>` 相当于`rm <filename>` 然后`git add`

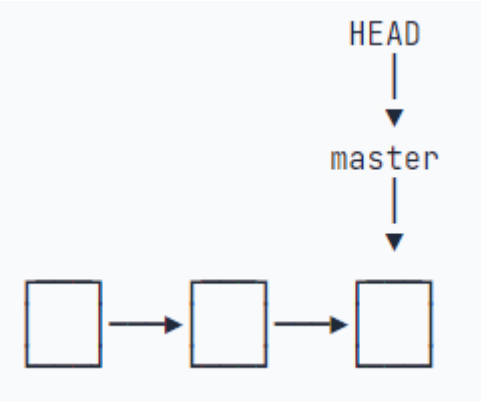
`git rm -r --cached  <filename>`命令可以将`filename`移除出工作区，并不会影响该文件在系统中的状态

Branch

Basic Operation

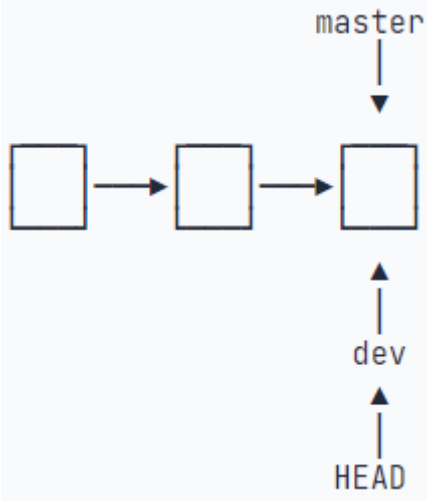
每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即`master`分支。`HEAD`严格来说不是指向提交，而是指向`master`，`master`才是指向提交的，所以，`HEAD`指向的就是当前分支。

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



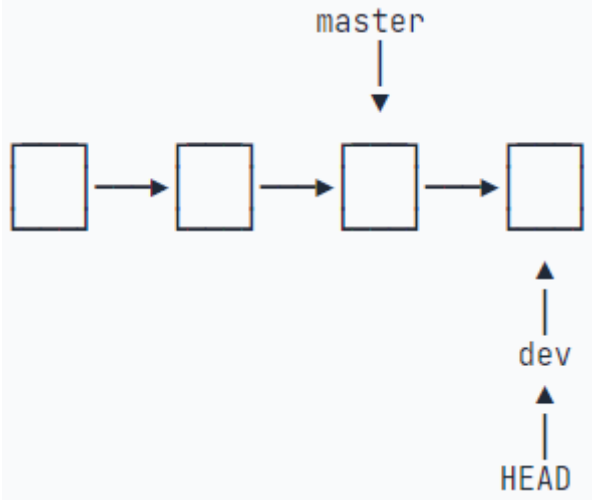
每次提交，`master`分支都会向前移动一步。

当创建新的分支，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，此操作等效为增加一个`dev`指针，并修改`HEAD`的指向为`dev`，工作区的文件没有任何变化。



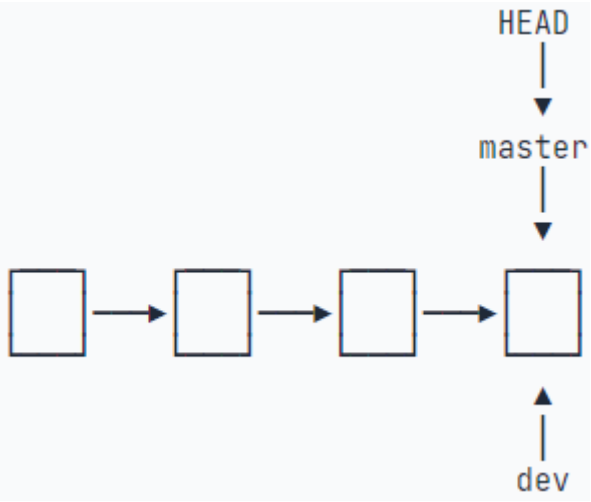
`git branch dev`命令会创建一个`dev`分支
`git branch`命令会列出所有分支，当前分支前面会标一个`*`号。
`git switch dev`命令可以在分支间切换。
`git switch -c dev`命令可以创建一个新的分支并切换至这个分支。

从现在开始，对工作区的修改和提交针对`dev`分支，新提交一次后，`dev`指针往前移动一步，而`master`指针不变：

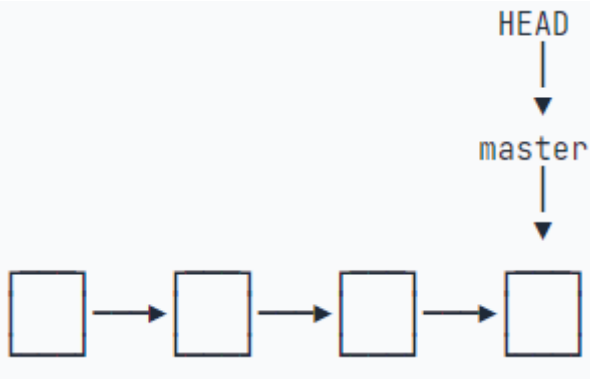


合并就是直接把`master`指向`dev`的当前提交：

`git merge dev`命令用于合并指定分支到当前分支。



`git branch -d dev`用于删除分支：



若分支还没有被合并，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。

`git branch -D dev`

Slove Conflict

Git 冲突通常出现在以下情况下：

- 同一文件的同一部分在两个分支中被修改。
- 合并分支时，Git 无法自动合并这部分改动，因为它不确定应该保留哪个修改。

例如，假设在 `master` 分支上修改了某行内容，在 `feature1` 分支上也修改了同一行内容。当尝试合并这两个分支时，Git 会无法自动选择哪个修改保留，便会标记冲突。

解决步骤：

1. 理解冲突标记：冲突部分会被标记为：
    - `<<<<<<< HEAD`：当前分支的内容。
    - `=======`：分隔符，表示冲突的开始和结束。
    - `>>>>>>> feature1`：目标分支的内容。
2. **手动合并**：根据需求，选择保留当前分支、目标分支的内容，或者两者的结合。
3. **删除冲突标记**：解决冲突后，删除冲突标记（`<<<<<<<`，`=======`，`>>>>>>>`），保持文件清晰。
4. **添加到暂存区并提交**：

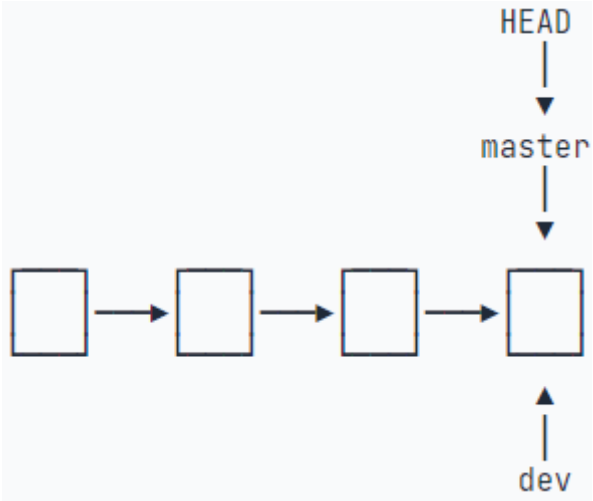
- 使用 `git add <文件>` 添加解决后的文件。
- 使用 `git commit` 提交。

```
PS D:\project\learn-git> git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
PS D:\project\learn-git> git add .\readme.txt
PS D:\project\learn-git> git commit -m "slove the conflict"
[master 14d521f] slove the conflict
PS D:\project\learn-git> git log --graph --pretty=oneline --abbrev-commit
*   14d521f (HEAD -> master) slove the conflict
|\
| * 4edd20b (feature1) commit 2 in feature1 branch
| * 562f9b1 commit 1 in feature1 branch
* | 235a472 commit 1 in master branch
|/
* 58a61cd add a line in dev branch
* 654d204 remove file:removefile.txt
* 83bb013 add a file to remove
* bab68eb commit second add of stage
* f85400c git track changes
* 55be549 understand how stage works
* 176a961 add GPL
* c7fd921 add distributed
* ede7da9 commit 3 files
* 17dc117 write a readme file: readme.txt
```

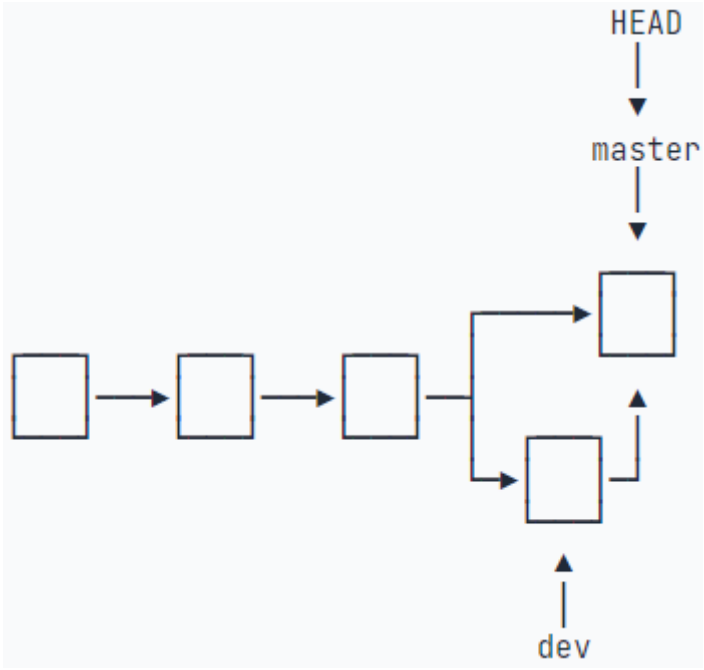
Fast Forward

使用场景：如果当前分支（如 `master`）自从目标分支（如 `feature`）创建以来没有任何新的提交，Git 可以执行快进合并。

特点：在快进合并中，Git 直接将当前分支指向目标分支的最新提交，没有生成额外的合并提交，比如如果 `master` 没有其他提交，Git 会将 `master` 分支指针直接指向到 `feature` 分支的最新提交，因此合并速度非常快。



有时可能希望避免快进合并，保持合并记录。这可以通过使用 `--no-ff` 参数强制 Git 创建一个合并提交，合并的同时要求生成一个新的`commit`，这样，从分支历史上就可以看出分支信息，就算分支最终被删除，合并提交也会记录曾经存在的分支信息。




```
PS D:\project\learn-git> git switch -c dev
PS D:\project\learn-git> git add .\readme.txt
PS D:\project\learn-git> git commit -m "commit 1 in dev branch"
[dev 8a87f56] commit 1 in dev branch
 1 file changed, 1 insertion(+)
PS D:\project\learn-git> git log --graph --pretty=oneline --abbrev-commit
* 8a87f56 (HEAD -> dev) commit 1 in dev branch
*   14d521f (master) slove the conflict
|\
| * 4edd20b commit 2 in feature1 branch
| * 562f9b1 commit 1 in feature1 branch
* | 235a472 commit 1 in master branch
|/
* 58a61cd add a line in dev branch
* 654d204 remove file:removefile.txt
* 83bb013 add a file to remove
* bab68eb commit second add of stage
* f85400c git track changes
* 55be549 understand how stage works
* 176a961 add GPL
* c7fd921 add distributed
* ede7da9 commit 3 files
* 17dc117 write a readme file: readme.txt
PS D:\project\learn-git> git switch master
Switched to branch 'master'
PS D:\project\learn-git> git merge --no-ff dev
Merge made by the 'ort' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

```
PS D:\project\learn-git> git log --graph --pretty=oneline --abbrev-commit
*   10b20fa (HEAD -> master) Merge branch 'dev'
|\
| * 8a87f56 (dev) commit 1 in dev branch
|/
*   14d521f slove the conflict
|\
| * 4edd20b commit 2 in feature1 branch
| * 562f9b1 commit 1 in feature1 branch
* | 235a472 commit 1 in master branch
|/
* 58a61cd add a line in dev branch
* 654d204 remove file:removefile.txt
* 83bb013 add a file to remove
* bab68eb commit second add of stage
* f85400c git track changes
* 55be549 understand how stage works
* 176a961 add GPL
* c7fd921 add distributed
* ede7da9 commit 3 files
* 17dc117 write a readme file: readme.txt
```

从指针的角度理解：

执行快进合并时：

- Git 会直接将 `master` 分支指针从 `C` 移动到 `E`（`feature` 分支的最新提交）。
- `master` 的提交历史将不再有合并记录，而是直接继承 `feature` 分支的所有提交。

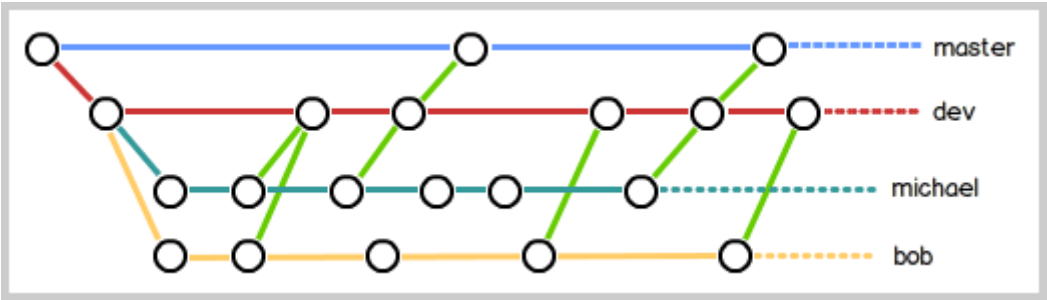
```
A---B---C  (master)
           \
            D---E  (feature)

            merge

A---B---C---D---E  (master, feature)
```

执行非快进合并时：

- Git 会创建一个新的合并提交 `M`，合并 `master` 和 `feature` 分支的更改。
- `master` 分支指针将指向合并提交 `M`。

```
A---B---C---F---M  (master)
           \     /
            D---E  (feature)
```

Branch Strategy



Stash

有时在一个分支上的工作还没做完，就要切换到另一个分支上工作，但是我们不想让此分支未提交的工作消失。

`stash`功能可以把当前工作分支现场“储藏”起来，等以后恢复现场后继续工作，等效于将当前工作区的内容暂时存储起来，并清空此分支上未提交的修改。

储藏完毕后，就可以切换当前分支，在另一个分支中完成任务之后，在切换回原来的分支，将`stash`中的工作恢复出来。

用`git stash list`命令查看`stash`存储的内容：

需要恢复原来的工作内容，有两个办法：

1. 一是用`git stash apply`恢复，但是恢复后，`stash`内容并不删除，还需要用`git stash drop`来删除；
2. 另一种方式是用`git stash pop`，恢复的同时把`stash`内容也删了。

还可以多次`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}
```

Remote Repository

本地仓库与远程库首先需要关联，可以使用`git remote add origin <ssh location>`，添加后，远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

下一步，就可以把本地库的所有内容推送到远程库上：

```
$ git push -u origin master
Counting objects: 20, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.
Total 20 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), done.
To github.com:michaelliao/learngit.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

`git push`命令实际上是把选择的本地仓库分支`master`推送到远程仓库。

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。

Cooperation

View

查看远程库信息，使用`git remote -v`；

Push

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；

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

```
$ git push origin <branchname>
```

分支推送原则：`master`，`dev`时刻与远程同步，其他分支根据工作要求确定。

Pull

从远程库clone时，默认情况下，只能看到本地的`master`分支。

如果要在其他分支上开发，就必须创建远程`origin`的其他分支到本地，使用这个命令创建远程`origin`的本地分支，前提是此分支已经被推送至远程。

```
$ git checkout -b dev origin/dev
```

如果同一分支被拉取至不同用户的本地，并且不同用户对同一分支做了不同修改后同时推送到远程，则推送失败。

这是因为不同用户的最新提交有冲突，远程分支比本地的分支更新，所以推送需要有先后顺序，如果用户1先将修改过的分支推送到远程，用户2就需要先用`git pull`把最新的提交从对应的`origin/branch`上抓下来，然后，在本地合并，解决冲突，再推送。

如果直接使用`git pull`，回提示错误：

```
$ 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
```

根据提示，在`pull`之前，还需要指定本地`dev`分支与远程`origin/dev`分支的链接：

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

再`pull`：

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

这回`git pull`成功，但是合并有冲突，需要手动解决后，提交，再push：

```
$ git commit -m "fix env conflict"
[dev 57c53ab] fix env conflict

$ git push origin dev
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 621 bytes | 621.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To github.com:michaelliao/learngit.git
   7a5e5dd..57c53ab  dev -> dev
```

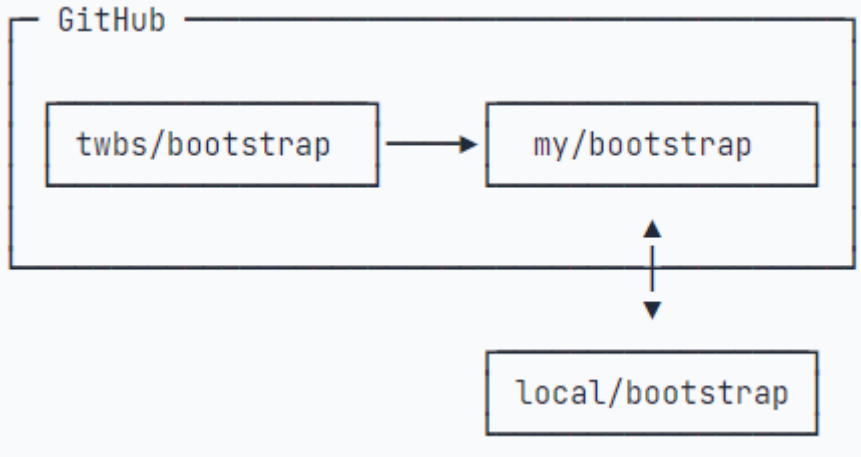
Appendix

Fork

使用Fork可以参与其他人的一个开源项目。Fork可以在自己的账号下克隆其他人的远程仓库，然后，从自己的账号下clone。

一定要从自己的账号下clone仓库，这样才能推送修改，如果直接clone其他人的仓库并在本地修改，因为没有权限，所以将不能推送本地修改。

以开源项目boostrap为例，Bootstrap的仓库`twbs/bootstrap`、在GitHub上克隆的仓库`myname/bootstrap`，以及自己克隆到本地电脑的仓库关系图如下：



如果希望官方能接受本地修改，就可以在GitHub上发起一个pull request，等待对方允许。

Config

`git config`中的`--global`参数是全局参数，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。

每个仓库的Git配置文件都放在`.git/config`文件中

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

```
d:\project\learn-git>cat ~/.gitconfig
[alias]
        br = branch
        st = status
        ci = commit
        re = reset
        sw = switch
        sk = stash
        glog = log --graph --pretty=oneline --abbrev-commit
[user]
        email = [UserEmail]
        name = [Username]
```