

# 1、Git的安装

---

## # 1.1 linux

---

### 1. centos平台下的安装

使用命令：

```
sudo yum install git
```

查看Git安装的版本：

```
git --version
```

### 2. ubuntu平台下的安装

使用命令：

```
sudo apt-get install git
```

查看Git安装的版本的命令和centos平台下一样。

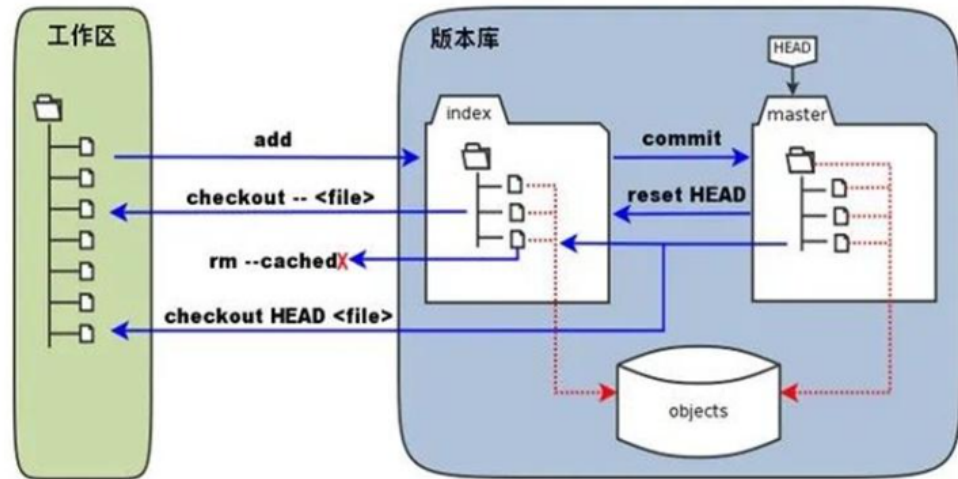
## # 1.2 windows

---

### 1. 工作区、暂存区、版本库

- 工作区：是在电脑上你要写代码或文件的目录
- 暂存区：英文叫stage或index.一般存放在.git目录下的index文件（.git/index）中，我们把暂存区有时也叫索引（index）

- 版本库：又叫仓库。工作区有一个隐藏目录.git，它不算工作区，而是Git的版本库，这个版本库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，以便任何时候都可以追踪历史，或者在将来某个时刻可以“还原”。



+ 图中左侧为工作区，右侧为版本库。Git的版本库里存了很多东西，其中最重要的就是暂存区。

+ 在创建Git版本库时，Git会为我们自动创建一个唯一的master（这个是主分支，但是在2020年后，GitHub、GitLab等平台以及Git官方（Git 2.28+支持配置默认分支名）都将默认主分支名改为了main,本文仍使用master)分支，以及指向master的一个指针叫HEAD。

+ 当对工作区修改（或新增）的文件执行\*\*git add\*\*命令时，暂存区目录树的文件索引会被更新。

+ 当执行提交操作\*\*git commit\*\*时，master分支会做相应的更新，可以简单理解为暂存区的目录树才会被真正写到版本库中。

## 2. git 的常见命令

最常见的是git add/git commit/git push

- **git status**

用于查看在你上次提交之后是否有对文件进行再次修改(本例是在已提交成功的工作区里新建一个test.txt内容空白文本文件)

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       test.txt

nothing added to commit but untracked files present (use "git add" to track)
```

- **git diff**

用来显示暂存区和工作区文件的差异（本例是将test.txt添加内容并且提交到暂存区之后再在工作区内更改）

```
$ git diff test.txt
diff --git a/test.txt b/test.txt
index 1d61521..8cd8ee4 100644
--- a/test.txt
+++ b/test.txt
@@ -1,2 @@
-测试是否有变化
\ No newline at end of file
+测试是否有变化
+这是在已经提交到暂存区之后，在工作区进行的更改
\ No newline at end of file
```

- **git reset**

用于回退版本，可以指定退回某一次提交的版本。回退的本质是要将版本库中的内容进行回退，工作区或暂存区是否回退由命令参数决定

git reset命令语法格式为：git reset [--soft|--mixed|--hard] [HEAD]

HEAD说明：

可直接写成commit id,表示指定退回的版本

HEAD表示当前版本

HEAD^上一个版本

HEAD^^上上一个版本 以此类推

**--soft**参数对于工作区和暂存区的内容都不变，只是将版本库回退到某个指定版本。通俗理解是“反悔提交，但保留所有修改”----相当于撤销了git commit，但git add的内容还在暂存区，工作区的修改也还在。适用场景是刚提交完发现提交信息写错了，想重新提交：git reset --soft HEAD^ ,然后修改提交信息重新git commit.这是最温和的一个回退参数。

**--mixed** 为默认选项，使用时可以不用带该参数。该参数将暂存区的内容退回为指定版本提交内容，工作区文件保持不变。通俗理解是“反悔提交+反悔暂存”----相当于撤销了git commit和git add,但工作区的修改还在，需要重新git add。适用场景：1.提交后发现漏加了文件，想重新选择要提交的文件：git reset HEAD^, 然后重新git add需要的文件再git commit。2.误把不需要提交的文件git add 了，想取消暂存。

**--hard**参数将暂存区与工作区都退回到指定版本。工作区有未提交的代码不要使用这个命令，因为工作区会回滚，你没有提交的代码就再也找不回了(git还提供了个保底的命令：git reflog,该命令记录本地的每一次命令，通过这个命令找到删除前提交的HEAD ID来找回文件)，所以使用该参数前一定要慎重。通俗理解就是“彻底回退”----让仓库完全回到指定提交的状态，所有未提交的修改都消失。适用场景：1.本地修改完全错误，想彻底放弃，回到之前的稳定版本：git reset --HEAD^, 2.拉取远程最新代码前，清空本都所有未提交的修改（避免冲突）

- git checkout -- [file] : 对于工作区的代码，还没有add的时候，我们可以使用这个命令让工作区的文件回到最近一次add或commit时的状态

## 2、分支管理

### # 2.1 理解分支

如何理解分支呢？

分支就是平行宇宙，当你在学习c++的时候，另一个你正在另一个平行宇宙里努力学习JAVA。当有一天两个平行宇宙合并到了一起，那么你就既学会了C++也学会了JAVA；

git支持我们查看或者创建其他分支，可以通过git branch查看本地现有的分支，并且通过git branch [分支名] 创建新分支。

```
chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git branch
* master

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git branch test

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git branch
* master
  test
```

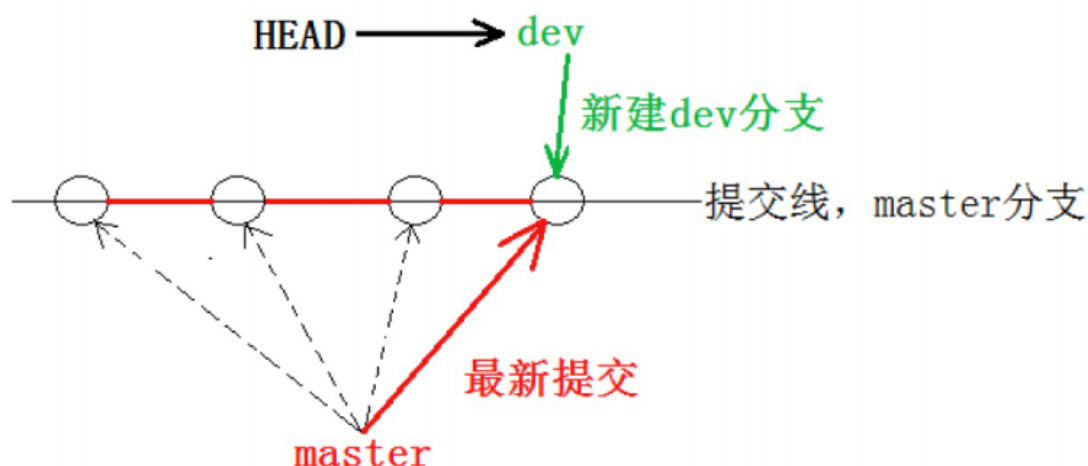
当我们创建新的分支以后，Git新建了一个指针叫test,\*表示当前HEAD指向的分支是master分支，

## # 2.2 切换分支

我们可以通过命令 `git checkout` 进行切换分支。

```
chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git checkout test
Switched to branch 'test'

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (test)
$ git branch
  master
* test
```



可以看到切换分支成功，HEAD指向了test分支。接下来在test分支下创建一个hello.md文件并且进行提交。

```
$ cat hello.md
# 我是在test分支下创建的文件

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (test)
$ git add hello.md

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (test)
$ git commit -m "add hello.md"
[test ed29d21] add hello.md
1 file changed, 2 insertions(+)
create mode 100644 hello.md
```

这个时候我们切换回master分支，查看hello.md文件，发现在master分支下没有这个文件

```
chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (test)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ cat hello.md
cat: hello.md: No such file or directory
```

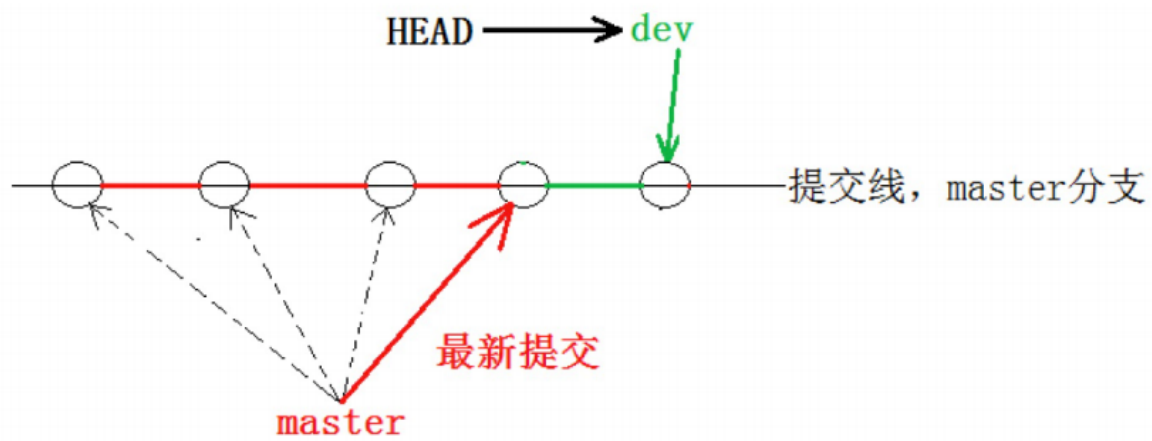
那我们在test分支下是否还有这个文件？经验证在test分支下这个文件是存在的，但是为什么在master分支下这个文件就看不到呢？？？

我们查看一下master和test分支的指向，发现两者的指向是不一样的

```
$ cat .git/refs/heads/test
ed29d21c59ecbf32875c60b8c834d07173dfb17d

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (test)
$ cat .git/refs/heads/master
3680b4815c4fca46bd4a6a7c594d6930d9cbd613
```

看到这里就明白了，因为我们是在test分支上提交的，而master分支此刻的提交点没有变化，此时的状态图如下：



当切换到master分支时，HEAD就指向了master,当然看不到创建的文件了。

## # 2.3 合并分支

为了在master分支上能看到最新的提交，就需要将test分支合并到master分支

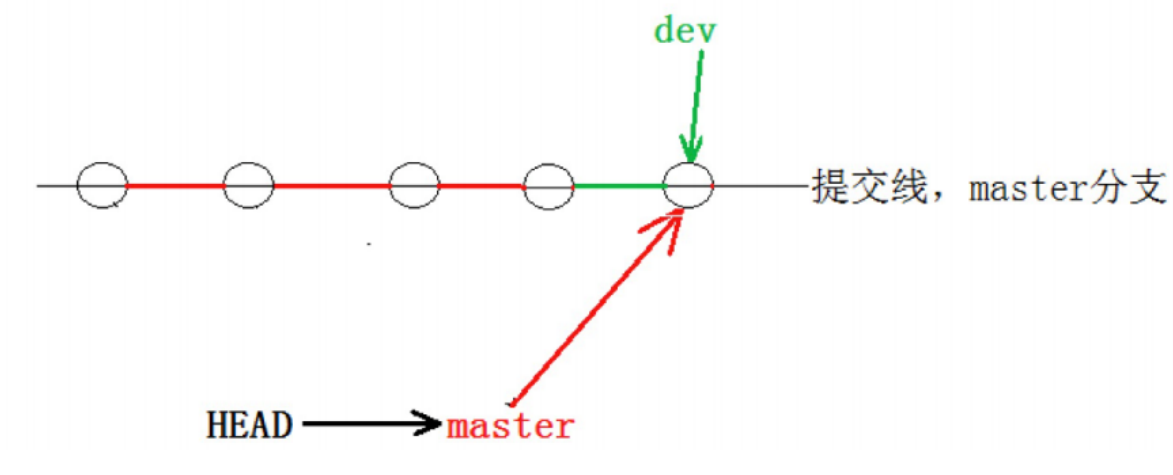
```
$ git branch
master
* test

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (test)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git merge test
Updating 3680b48..ed29d21
Fast-forward
 hello.md | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 hello.md

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ cat hello.md
# 我是在test分支下创建的文件
```

`git merge` 命令用于合并指定分支到当前分支。合并后，`master`分支就能看到`test`分支提交的文件。此时的状态如下图所示：



## # 2.4 删除分支

---

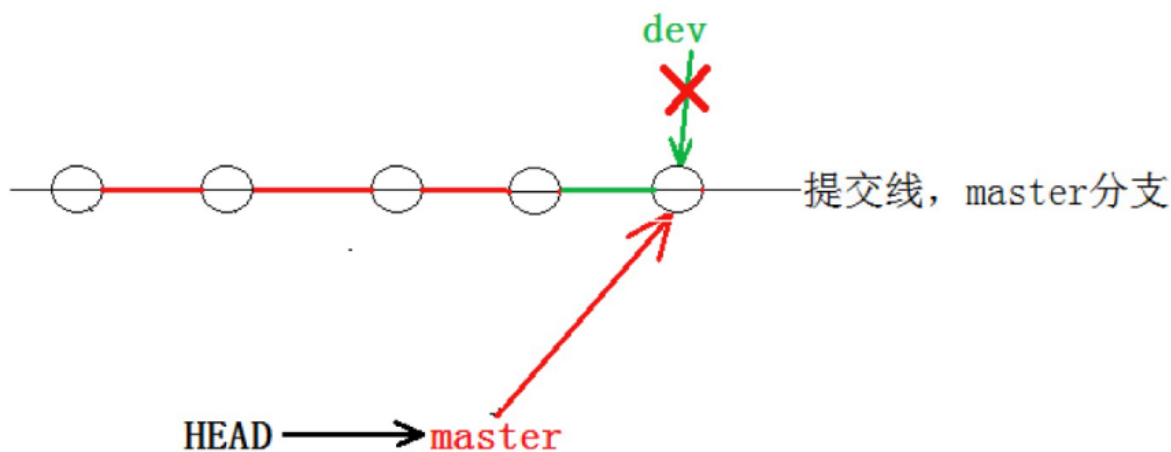
合并完成后，`test`分支对于我们就没有用了，那么`test`分支就可以删除了，注意如果当前正处于某分支下，就不能删除当前分支。而可以在其他分支下删除当前分支。

```
$ git branch
* master
  test

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git branch -d test
Deleted branch test (was ed29d21).

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git branch
* master
```

此时的状态图如下图所示：



## # 2.5 合并冲突

在实际的分支合并的时候，并不是想合并就能合并成功的，有时候可能会遇到代码冲突的问题。

我们这里创建一个dev分支用来演示，创建dev分支并切换到dev分支。

```
$ git branch
* master

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git branch dev
chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git branch dev
* master

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git checkout dev
Switched to branch 'dev'

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (dev)
$ git branch
* dev 合并冲突
master
```

在dev分支下修改hello.md文件并且进行提交一次。

```
$ cat hello.md
# 我是在test分支下创建的文件

# 我是在dev分支下添加的


chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (dev)
$ git add hello.md

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (dev)
$ git commit -m "dev add"
[dev c5fb200] dev add
1 file changed, 2 insertions(+)
```

此时按照我们之前所示切换回master分支，应该是看不到在dev分支下添加的文字的，事实也是这样的。这个时候我们在master分支下也对hello文件进行添加内容并提交，如下图所示：

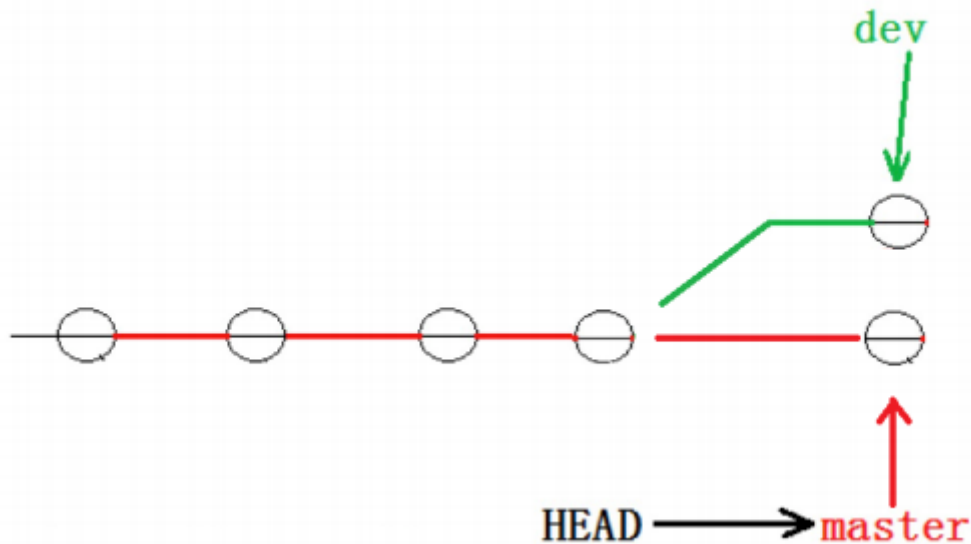
```
$ cat hello.md
# 我是在test分支下创建的文件

# 我是在master分支下添加的


chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git add .

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git commit -m "master add"
[master de30c92] master add
1 file changed, 2 insertions(+)
```

现在master分支和dev分支都有了各自的提交，此时的状态变成了这样：



这种情况下，Git 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，如下所示：

```
$ git merge dev
Auto-merging hello.md
CONFLICT (content): Merge conflict in hello.md
Automatic merge failed; fix conflicts and then commit the result.

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master|MERGING)
$ 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:   hello.md

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

这种情况下，

执行git status命令确认冲突文件和当前状态：

发现 ReadMe 文件有冲突后，可以直接查看文件内容，要说的是 Git 会用 <<<, ==, >>>

来标记出不同分支的冲突内容，如下所示：

```
$ cat hello.md
# 我是在test分支下创建的文件

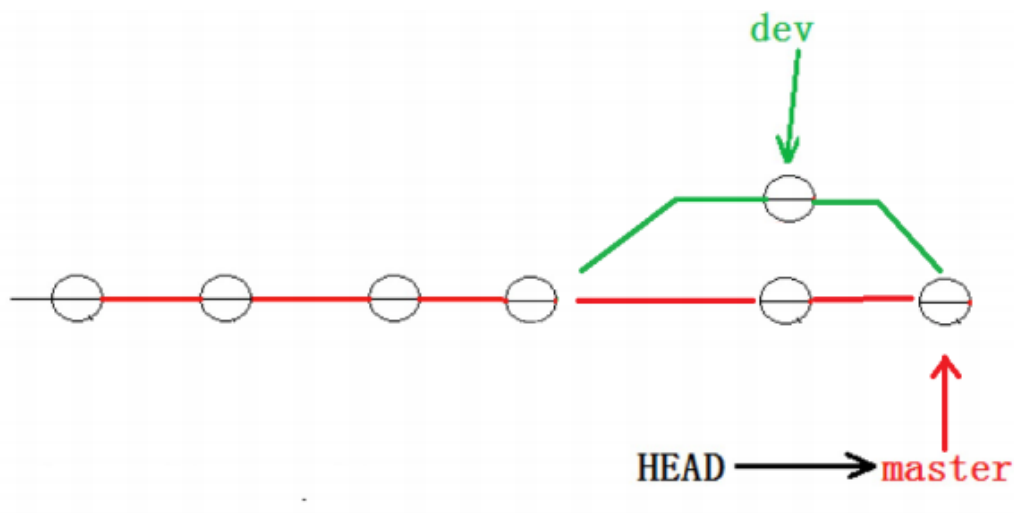
<<<<<< HEAD
# 我是在master分支下添加的
=====
# 我是在dev分支下添加的
>>>>>> dev
```

此时我们必须手动调整冲突代码，并需要再次提交修正后的结果！

```
$ cat hello.md
# 我是在test分支下创建的文件

# 我是在dev分支下添加的
```

到这里冲突就解决结束了，此时的状态为：



最后删除这个示例用的dev分支。

## # 2.6 删除临时分支

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，最好新建一个分支，我们可以将其称之为 `feature` 分支，在上面开发，完成后，合并，最后，删除该 `feature` 分支。

可是，如果我们今天正在某个 `feature` 分支上开发了一半，被产品经理突然叫停，说是要停止新功能的开发。虽然白干了，但是这个 `feature` 分支还是必须就地销毁，留着无用了。这时使用传统的 `git branch -d` 命令删除分支的方法是不行的（`git branch -d` 是 Git 提供的安全删除分支命令，它的设计初衷是防止你误删包含“未合并到当前分支（通常是主分支）”提交记录的分支——因为这些提交记录一旦删除，就可能永久丢失（除非你知道提交哈希值去恢复））

此时使用强制删除命令 `git branch -D [分支名]` 就可以删除临时分支。

## 3、远程操作

### # 3.1 远程仓库

Git 是分布式版本控制系统，同一个 Git 仓库，可以分布到不同的机器上。实际情况往往是这样，找一台电脑充当服务器的角色，每天24小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

完全可以自己搭建一台运行 Git 的服务器，不过现阶段，为了学 Git 先搭个服务器绝对是小题大作。好在这个世界上有个叫 GitHub 的神奇的网站，从名字就可以看出，这个网站就是提供 Git 仓库托管服务的，所以，只要注册一个GitHub账号，就可以免费获得 Git 远程仓库。

GitHub仓库的创建和克隆这里略过。

当我们从远程仓库克隆后，实际上 Git 会自动把本地的 master 分支和远程的 master 分支对应起来，并且，远程仓库的默认名称是 origin 。在本地我们可以使用 git remote (+参数v可以显示更多信息) 命令，来查看远程库的信息，

```
$ git remote -v
origin  git@github.com:coderchen02/practice.git (fetch)
origin  git@github.com:coderchen02/practice.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。

`git push origin master` : 首次推送时，绑定本地 master 到 origin/master 并推送

`git push` : 普通推送

如果在远端仓库进行了修改，此时，远程仓库是要领先于本地仓库一个版本，为了使本地仓库保持最新的版本，我们需要拉取下远端代码，并合并到本地。Git 提供了 git

pull 命令，该命令用于从远程获取代码并合并本地的版本。如下所示

```
$ git pull origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 974 bytes | 88.00 KiB/s, done.
From github.com:coderchen02/practice
* branch                master      -> FETCH_HEAD
  b6e9d71..be7d10f      master      -> origin/master
Updating b6e9d71..be7d10f
Fast-forward
 hello.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

## # 3.2 配置Git

---

### 1. 忽略特殊文件

在日常开发中，我们有些文件不想或者不应该提交到远端，比如保存了数据库密码的配置文件，那怎么让 Git 知道呢？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了。

例如我们想忽略以 .so 和 .ini 结尾的所有文件，.gitignore 的内容如下：

```
#my configurations
*.so
*.ini
#也可指定具体的文件，这个我们测试使用test.txt
test.txt
```

写好.gitignore之后进行添加到暂存区并且推送到远端。

此时我们在本地添加两个测试文件testini.ini和test.txt之后

```
$ ls
hello.md  test.txt  testini.ini

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

检验 .gitignore 的标准就是 git status 命令是不是说 working tree clean 。我们发现Git并没有提示在工作区中有文件新增，果然 .gitignore 生效了。

但是有些时候，你就是想添加一个文件到 Git，但由于这个文件被 .gitignore 忽略了，根本添加不了，那么可以用 -f 强制添加

```
$ git add -f [filename]
```

## # 3.3 标签管理

### 1. 理解标签

标签 tag ，可以简单的理解为是对某次 commit 的一个标识，相当于起了一个别名。例如，在项目发布某个版本的时候，针对最后一次 commit 起一个 v1.0 这样的标签来标识里程碑的意义。当我们需要回退到某个重要版本时，直接使用标签就能很快定位到。

### 2. 创建标签

首先切换到需要打标签的分支上，然后使用命令 git tag [name] 就可以打一个标签。可以使用 git tag 来查看所有标签。默认标签是打在最新提交的 commit 上的。如果需要对特定提交打标签，只需要使用命令 git tag [name] [commid\_id]。

```
$ git branch
* master

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git tag v1.0

chen@DESKTOP-NQ08IEQ MINGW64 ~/Desktop/practice (master)
$ git tag
v1.0
```

Git 还提供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，格式为

```
git tag -a [name] -m "xxx" [commit_id]
```

### 3. 操作标签

删除标签命令：

```
git tag -d [name]
```

如果要推送某个标签到远端，使用命令：

```
git push origin [tag_name]
```

当然如果在本地有很多标签，也可以一次性推送到远端：

```
git push origin --tags
```

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，然后，从远程删除。删除命令也是push，但是格式如下：

```
git push origin :ref/tags/[tag_name]
```

## # 3.4 多人协作

---

目前我们已经掌握了Git 的所有本地库的相关操作，git基本操作，分支理解，版本回退，冲突解决等等。但是在项目开发中，往往是多个人在一个分支上共同开发，比如多个人在dev分支上进行开发。

1. 首先开发的时候使用 `git pull` 拉取分支最新的代码。
2. 同事A在dev分支上进行了开发，并且通过 `git push origin dev` 推送到了远端
3. 这个时候同事A推送到修改代码已经可以在远端看到，如果你碰巧也需要对其中一个文件修改，并且尝试推送到远端的dev分支，这个时候会推送失败，因为你的你伙伴的最新提交和你推送的提交有冲突。
4. 解决办法也很简单，先用 `git pull` 把最新的提交origin/dev 抓下来，然后，在本地进行合并，并解决冲突，再推送。操作如下：

1. `git pull`
2. `cat` [冲突文件]
3. 手动处理冲突
4. 重新添加到暂存区并推送到远端dev分支

由此，两名开发者已经开始可以进行协同开发了，不断的 `git pull/add/commit/push` ,遇到了冲突，就使用我们之前讲的冲突处理解决掉冲突。

总结一下，在同一分支下进行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！
- 功能开发完毕，将分支 `merge` 进 `master`，最后删除分支。