

Git 学习笔记

一、初识 Git

1. Git 是什么？

Git 是世界上目前最先进的**分布式**版本控制系统。

2. 什么是版本控制系统？

简单来说就是用于多人协同开发项目的技术。比如在一个项目组中多人协同完成开发任务，可以记录我们每次的文件改动，还可以让同事们协助编辑，这样就不用自己管理一堆类似的文件，也不需要把文件传来传去，我们如果想要查看某次改动，只需要在版本控制系统中看一眼就可以，十分的方便快捷，用起来如下表。

版本	文件名	改动人	说明	日期
V1	Hello.java	张三	输出 hello, 张三	2023/5/1
V2	Hello.java	张三	改为 hello, zhangsan	2023/5/5
V3	Hello.java	李四	增加 hello,lisi	2023/5/6
V4	Hello.java	张三	删除 hello,lisi	2023/5/8

3. Git 是怎么来的？

想必大家都听过 Linux 操作系统，Linux 自 1991 年创建以来，经过多年的发展已经成为最大的服务器系统软件。Linux 操作系统是靠全世界热心的志愿者参与建设的，那么世界各地的人都在为 Linux 系统编写代码，那 Linux 操作系统是如何管理的呢？事实上，在 2002 年之前，是由 Linus 本人手工合成的，但是 Linux 已经发展几十年了，代码库的庞大已经不能让 linus 通过手工的方式去管理代码了。一家好心的企业 BitK Mover 出于好心，将自家的商用版本控制系统免费授权给 Linux 社区使用，但是 Linux 社区里面的大哥太多了，直接破解了商用版本控制系统 BitKeeper，BitMover 公司一气之下扬言要收回 Linux 社区的免费使用权。谁知 Linus 又高又硬，不仅没有道歉并且仅用了两周时间自己用 C 写了一个**分布式版本控制系统，这就是 Git！**在一个月内 Linux 系统的源码就已经使用 Git 来进行管理了。后来 Git 迅速成为最流行的分布式版本控制系统，尤其是 2008 年，GitHub 网站上线了，它为开源项目免费提供 Git 存储，无数开源项目开始迁移至 GitHub，包括 jQuery，PHP，Ruby 等等。不知道现在 BitMover 公司有没有后悔，不过这也告诉我们一个道理，大神就是大神！



4.集中式版本控制系统和分布式版本控制系统比较

(1) 集中式版本控制系统

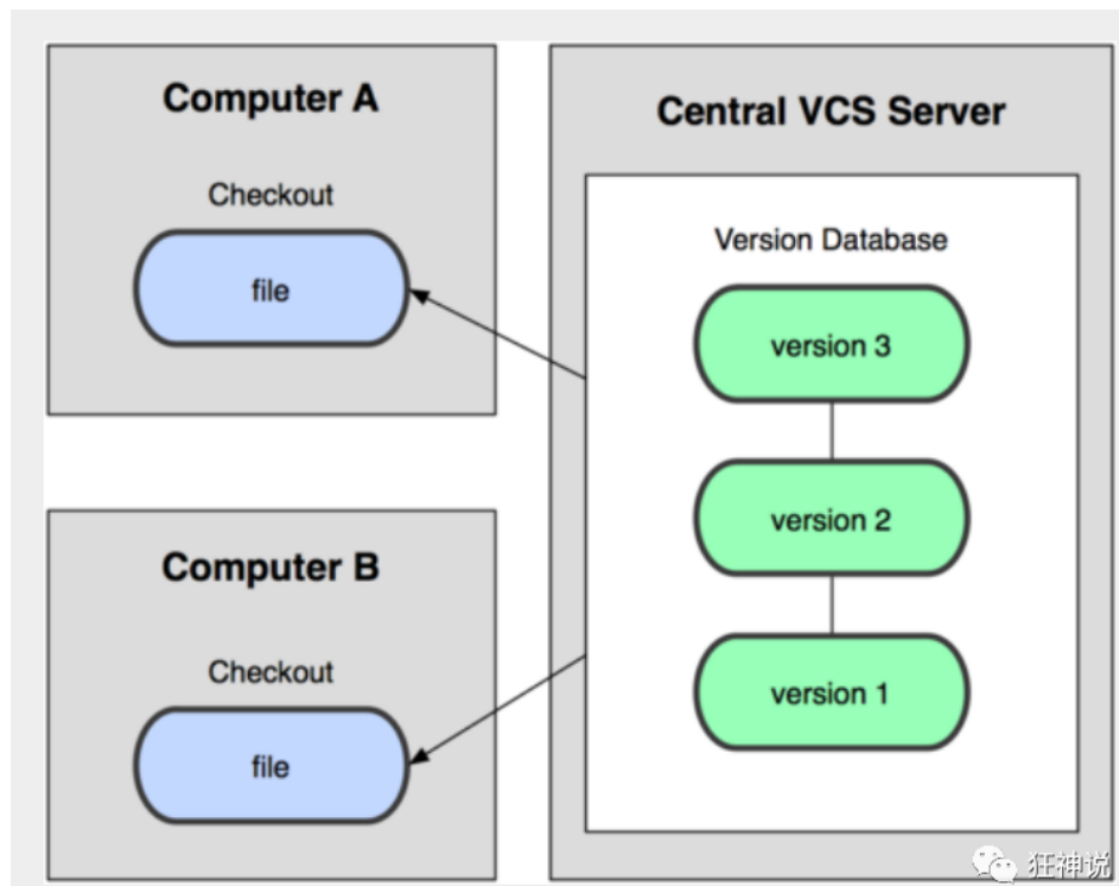
所有版本的数据都保存在服务器上，开发者从服务器上更新或者上传自己的代码。

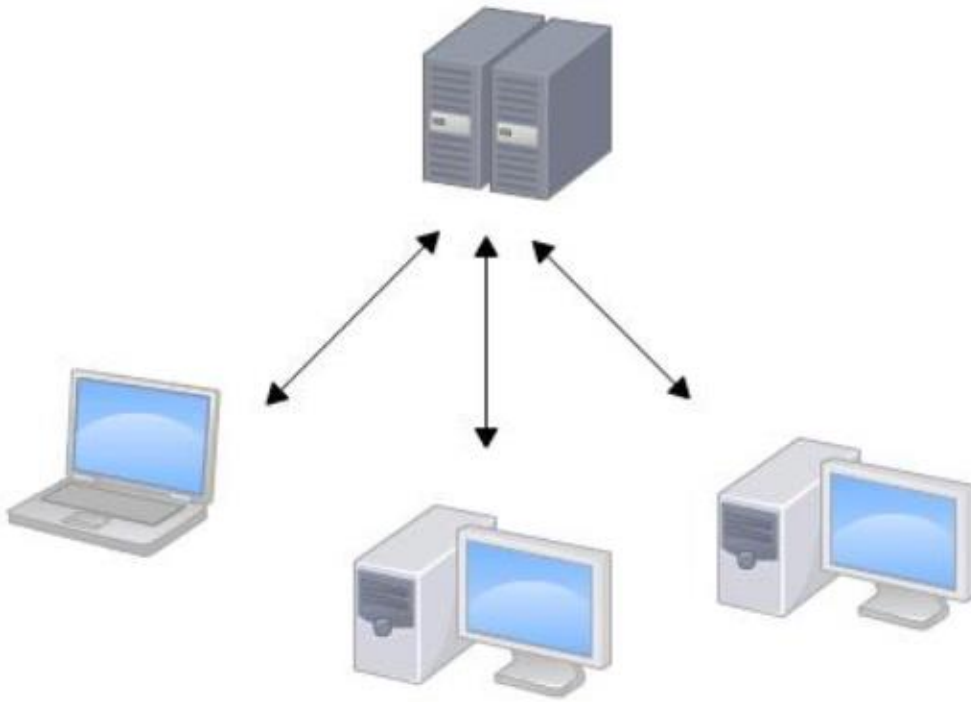
所有的版本数据都存在服务器上，开发者自己的电脑上只有之前同步的版本，如果不联网，用户就看不到历史版本。

如果所有的数据都保存在一个单一的服务器上，那么这个服务器发生损坏就可能丢失掉所有的数据，当然可以定期备份。

代表的产品有：SVN、CVS、VSS

缺点：**需要联网使用**，如果网络不好，访问速度会变慢。





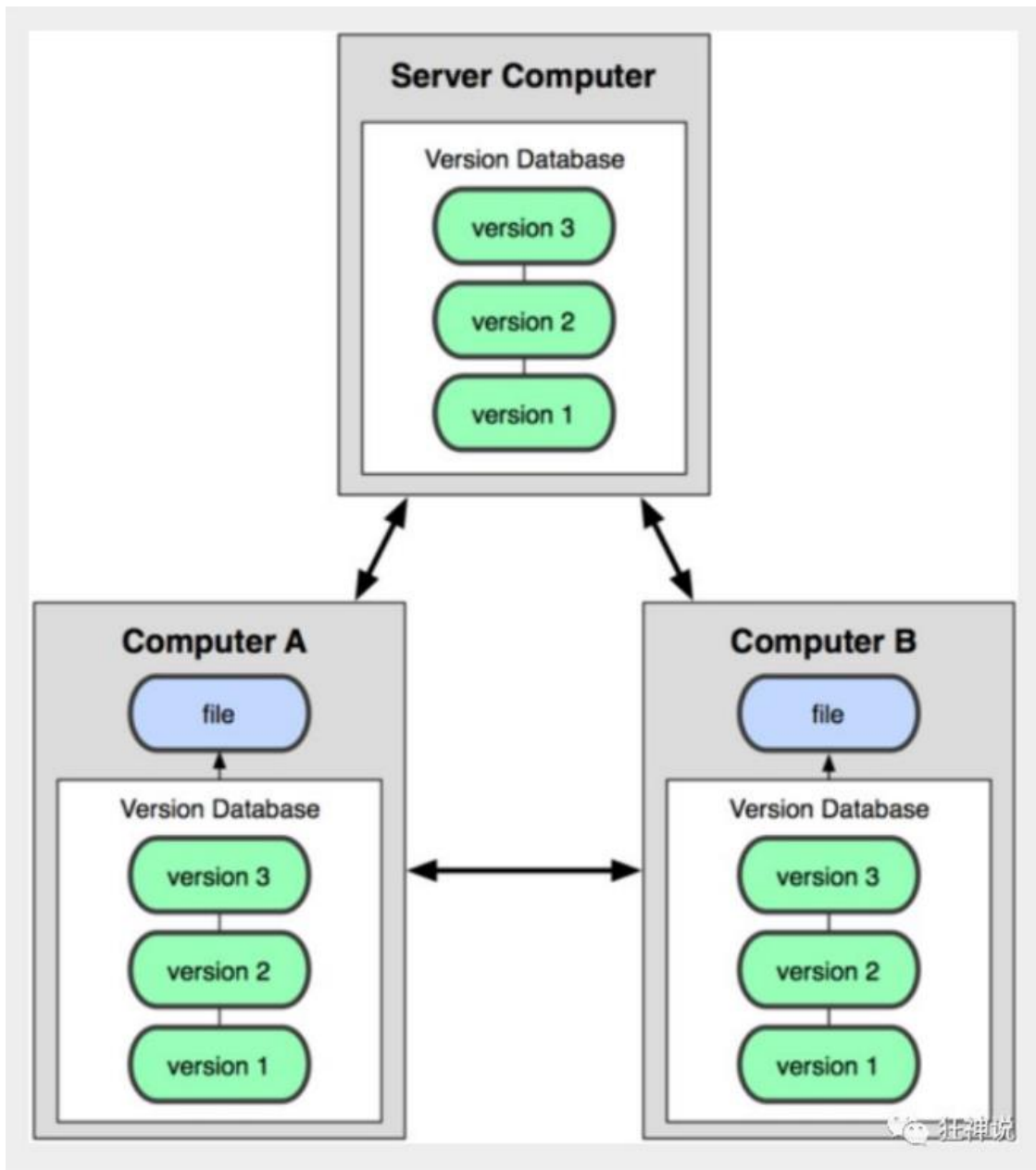
（2）分布式版本控制系统（Git）

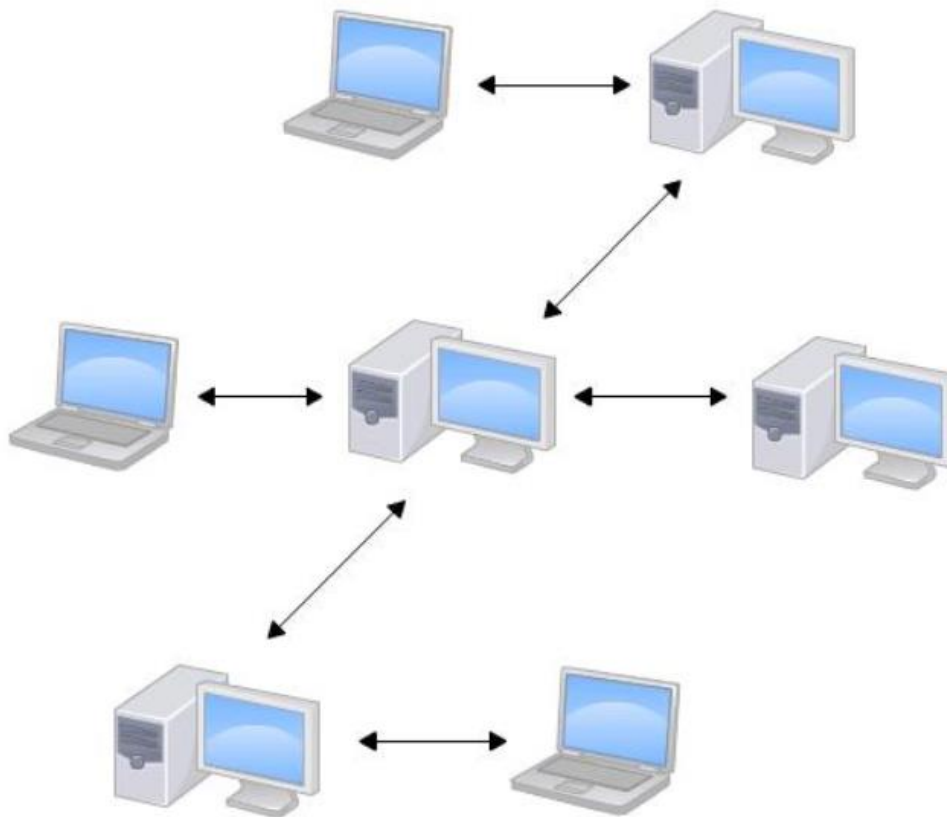
所有的版本信息仓库全部同步到本地的每个用户，这样每个开发者在本地就可以查看所有版本历史。

可以在离线的环境下在本地提交，只需要在联网的时候 **push** 到相应的服务器或者用户那里就可以了。

由于每个用户存放的都是所有版本的数据，只要有一个用户的设备没有问题就可以恢复所有的数据，不会因为服务器损坏或者网络问题，造成不能工作。

小问题就是每个人都拥有所有版本的代码，有一定的安全隐患，同事会占用多一点本地内存（基本上没啥缺点）





为啥分布式控制系统也需要“中央服务器”？

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，**分布式版本控制系统通常也有一台充当“中央服务器”的电脑**，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。

SVN 的服务器和 GitHub 的区别

集中式和分布式的区别是：

你的本地是否有完整的版本库历史！

假设 **SVN** 服务器没了，那你丢掉了所有历史信息，因为你的本地只有当前版本以及部分历史信息。

假设 **GitHub** 服务器没了，你不会丢掉任何 **git** 历史信息，因为你的本地有完整的版本库信息。你可以把本地的 **git** 库重新上传到另外的 **git** 服务商。

二、Git 的使用

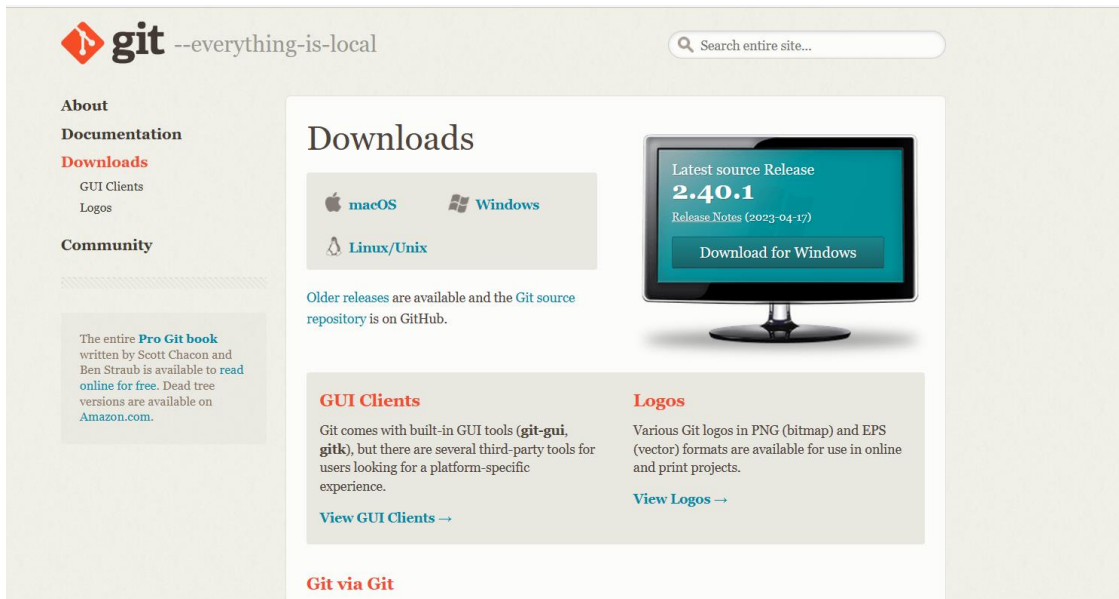
1. Git 的下载与安装

(1) 可以去 Git 官网进行下载，[Git - Downloads \(git-scm.com\)](https://git-scm.com/downloads)

(2) 官网下载太慢，我们可以使用淘宝镜像下载：














<http://npm.taobao.org/mirrors/git-for-windows/>在下载比较慢的时候我们可以去找镜像。

官网下载：

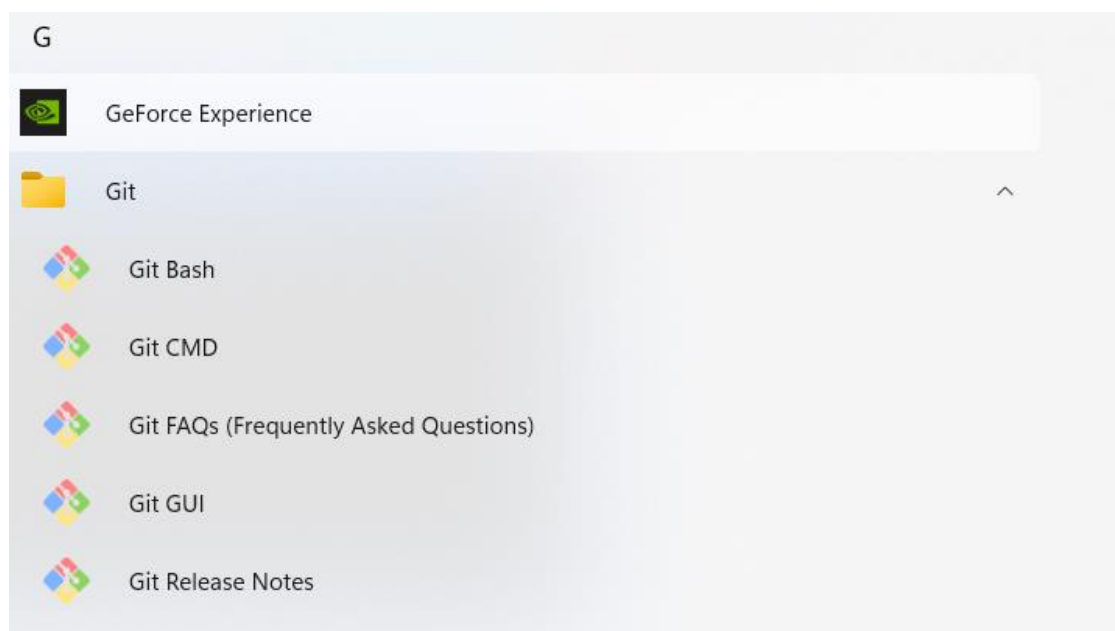


淘宝镜像下载：

Index of /git-for-windows/

	<u>Name</u>	Last modified	Size
	Parent Directory		-
	v2.11.1.mingit-prerelease.4/	2019-08-22T08:32:03Z	-
	v2.11.1.mingit-prerelease.5/	2019-08-22T08:35:29Z	-
	v2.11.1.mingit-prerelease.6/	2019-12-10T18:10:55Z	-
	v2.14.4.windows.3/	2019-08-22T08:47:24Z	-
	v2.14.4.windows.4/	2019-08-22T09:01:28Z	-
	v2.14.4.windows.5/	2019-12-10T18:13:14Z	-
	v2.14.4.windows.6/	2020-04-14T18:51:35Z	-
	v2.14.4.windows.7/	2020-04-20T23:18:44Z	-
	v2.14.4.windows.8/	2020-04-24T13:57:21Z	-
	v2.18.0.windows.1/	2018-06-22T11:58:54Z	-
	v2.19.0-rc0.windows.1/	2018-08-21T20:19:53Z	-
	v2.19.0-rc0.windows.2/	2018-08-23T22:52:29Z	-

安装：无脑下一步即可，安装完毕就可以使用了。



我们一般需要关注以下三个程序：

Git Bash: Unix 与 Linux 风格的命令行，使用的最多，推荐的最多。

Git CMD: Windows 风格的命令行。

Git GUI: 图形界面的 Git,不建议初学者使用，今尽量先熟悉常用命令。

2.常用的 Linux 命令

1、cd : 改变目录

2、cd : 回退到上一个目录，直接 cd 进入默认目录

3、pwd : 显示当前所在的目录路径。

4、ls(ll) : 都是列出当前目录中的所有文件，只不过 ll(两个 l)列出的内容更加详细。

5、touch : 新建一个文件 如 touch index.js 就会在当前目录下新建一个 index.js 文件。

6、rm : 删除一个文件，rm index.js 就会把 index.js 文件删除。

7、mkdir : 新建一个目录，就是新建一个文件夹。

8、rm -r : 删除一个文件夹，rm -r src 删除 src 目录

rm -rf / 切勿在 Linux 中使用！作用是删除电脑中的全部文件。

9、mv 移动文件，mv index.html src index.html src 是目标文件夹，这样写必须保证文件和目标文件在同一目录下。

10、reset 重新初始化终端/清屏。

11、clear 清屏。

12、history 查看命令历史。

13、help 帮助。

14、exit 退出。

15、#表示注释。

3.设置用户名与邮箱

```
git config --global user.name "yourName" #名称
```

```
git config --global user.email "xxx@qq.com" #邮箱
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit
$ git config --global user.name "yanghao"

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit
$ git config --global user.email "1@qq.com"
```

设置完成后我们可以查询一下是否设置成功

```
git config --global --list
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit
$ git config --global --list
user.name=yanghao
user.email=1@qq.com
credential.helper=manager
```

为什么要设置用户名和邮箱？

因为 Git 是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和 Email 地址。

注意 `git config` 命令的 `--global` 参数，用了这个参数，表示你这台机器上所有的 Git 仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和 Email 地址。

4. 创建版本库

什么是版本库？

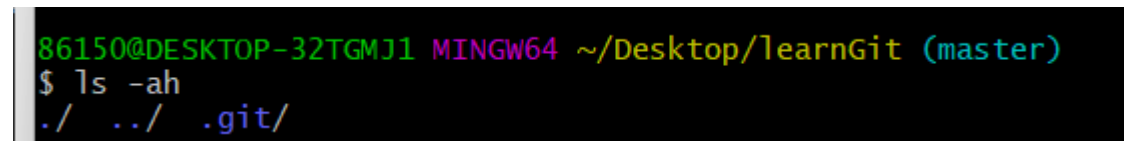
版本库又叫做仓库，英文名 `repository`，你可以简单理解成一个目录，这个目录里面所有的文件都可以被 Git 管理起来，每个文件的修改、删除，Git 都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻进行“还原”。

通过 `git init` 命令把目录变成 Git 可以管理的仓库。

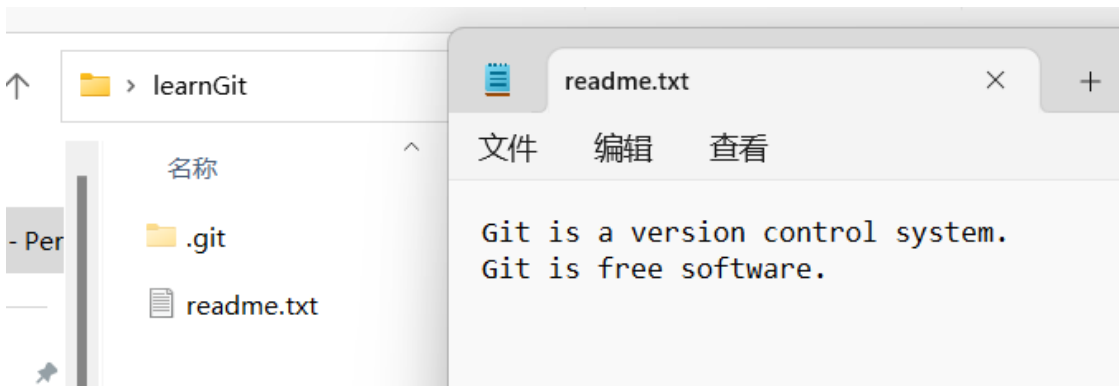


在执行玩 `git init` 的命令后会发现在文件夹中多了一个 `.git` 的文件，这个目录是 `git` 来跟踪管理版本库的，没事千万不要手动修改这个目录文件里的文件，不然改乱了，就把 `git` 库给破坏了。

如果没有看见这个 `.git` 目录，那是因为这个目录默认是隐藏的，用 `ls -ah` 命令就可以查看。



创建一个 `readme.txt` 文件，内容如下：



第一步，用命令 `git add` 告诉 `Git`，把文件添加到仓库：

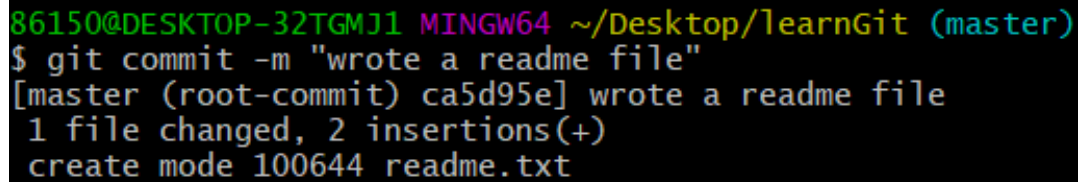
```
git add readme.txt
```



执行上面的命令，没有任何显示，这就对了，Linux 中“没有消息就是好消息”，说明添加成功。

第二部，用 `git commit` 告诉 git，把文件提交到仓库去：

```
git commit -m "wrote a readme file"
```



```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git commit -m "wrote a readme file"
[master (root-commit) ca5d95e] wrote a readme file
1 file changed, 2 insertions(+)
create mode 100644 readme.txt
```

简单解释一下 `git commit` 命令，`-m` 后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的内容，这样就能从历史中很方便的找到修改记录。

`git commit` 命令执行成功后会告诉你，**1 file changed**：1 个文件被改动（我们新添加的 `readme.txt` 文件）；**2 insertions**：插入了两行内容（`readme.txt` 有两行内容）。

为什么 Git 添加文件需要 `add,commit` 一共两步呢？

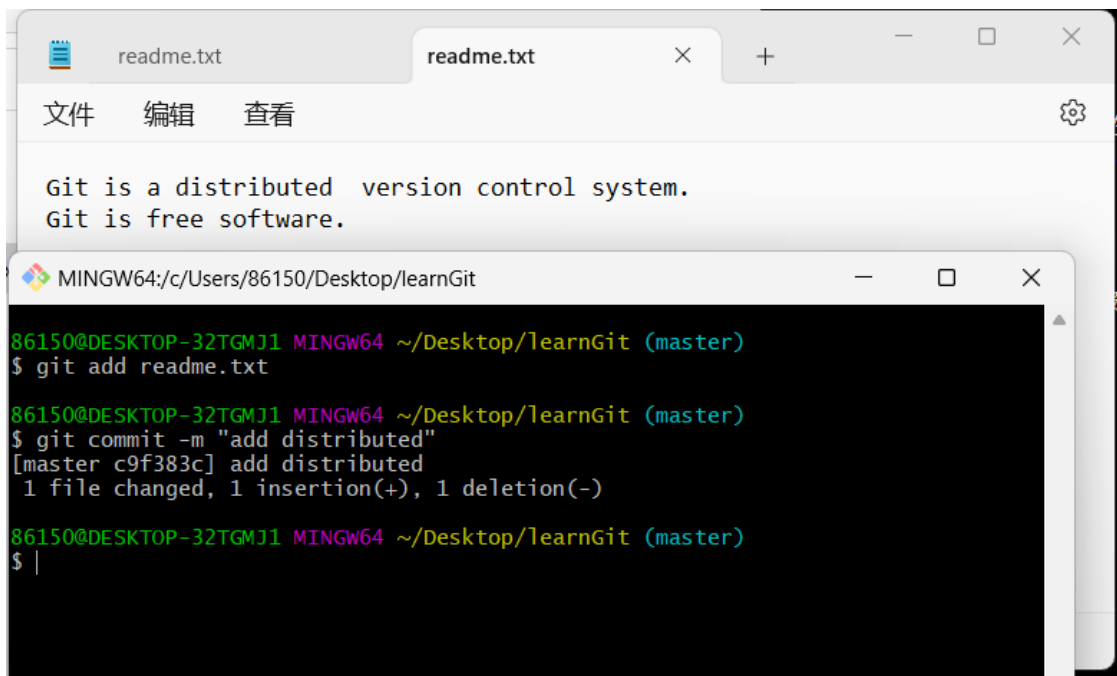
因为 `commit` 可以一次提交很多文件，所以你可以多次 `add` 不同的文件，比如：

```
git add file1.txt
git add file2.txt file3.txt
git commit -m "add 3 files."
```

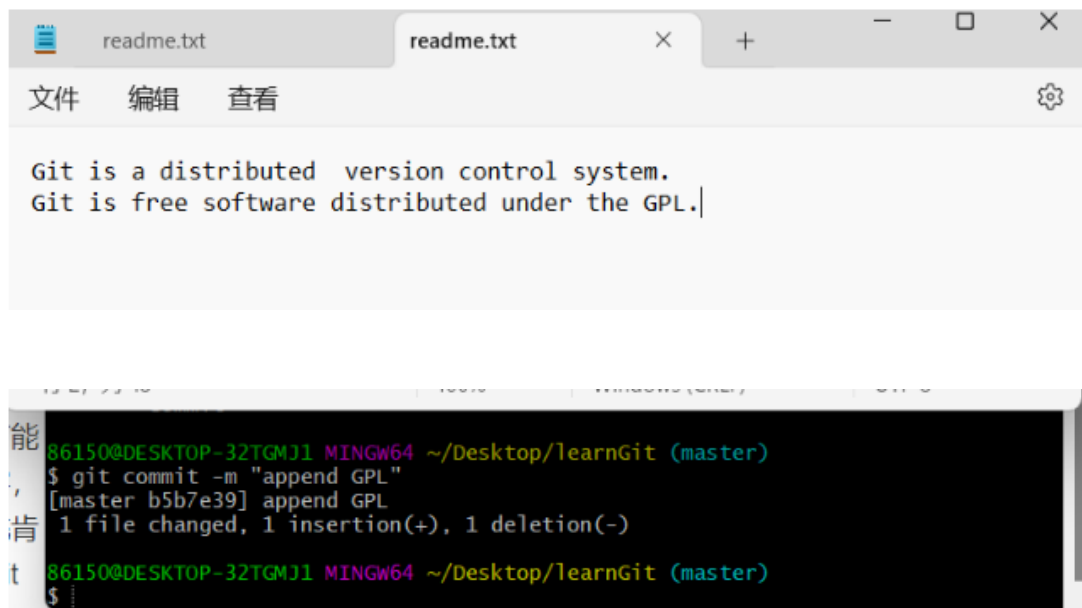
5.版本之间的切换

1.版本的回退

在上面的步骤下，修改 `readme.txt` 文件，修改的内容如下：



我们继续修改 readme.txt 文件，然后 git add 和 git commit：



在实际的开发工作中，我们的脑子不可能记住每个版本改了哪些内容，但是版本控制系统中的某个命令可以告诉我们历史记录，在 git 中我们使用 git log 命令查看

```
MINGW64:/c/Users/86150/Desktop/learnGit

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log
commit b5b7e39ea8780feeaf41d90f34e68fbb65d0993f (HEAD -> master)
Author: 
Date:   Thu May 18 21:11:15 2023 +0800

    append GPL

commit c9f383c36e2e2d35c6b3378f9a71f63827dc8103
Author: 
Date:   Thu May 18 21:11:15 2023 +0800

    add distributed

commit ca5d95eca8d957b1247d71debe9e9a8df07eec70
Author: 
Date:   Thu May 18 21:11:15 2023 +0800

    wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$
```

git log 可以看到从最近到最远的提交日志，如果嫌输出的信息太多的话可以在 git log 后面加上--pretty=oneline

```
MINGW64:/c/Users/86150/Desktop/learnGit

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
b5b7e39ea8780feeaf41d90f34e68fbb65d0993f (HEAD -> master) append GPL
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file
```

大家可以看到的是，commit id 都是一大串，而不是 1、2、3、4.....这样递增的，commit id 是一个 SHA1 计算出来的，用十六进制表示，不采用递增这种简单的 id 号的原因是因为同一个版本库可能有多个人在使用这样命名容易冲突。

我们现在想将 readme.txt 文件回退到上一个版本也就是 append GPL 的那个版本，我们需要怎么做呢？

(1) 我们首先需要知道当前版本是哪个版本，在 git 中用 HEAD 表示当前版本。我们可以看到当前版本是 append GPL

```
MINGW64:/c/Users/86150/Desktop/learnGit
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
b5b7e39ea8780feeaf41d90f34e68fbb65d0993f (HEAD -> master) append GPL
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file
```

(2) HEAD 表示的是当前版本, HEAD^表示的就是上一个版本, 上上个版本就是 HEAD^^,那要是往上 100 个版本怎么写呢? HEAD~100

(3) 现在我们需要把当前版本回退到上一个版本, 我们可以使用 git reset 命令:
注意在这个地方要加上--hard

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset --hard HEAD^
HEAD is now at c9f383c add distributed
```

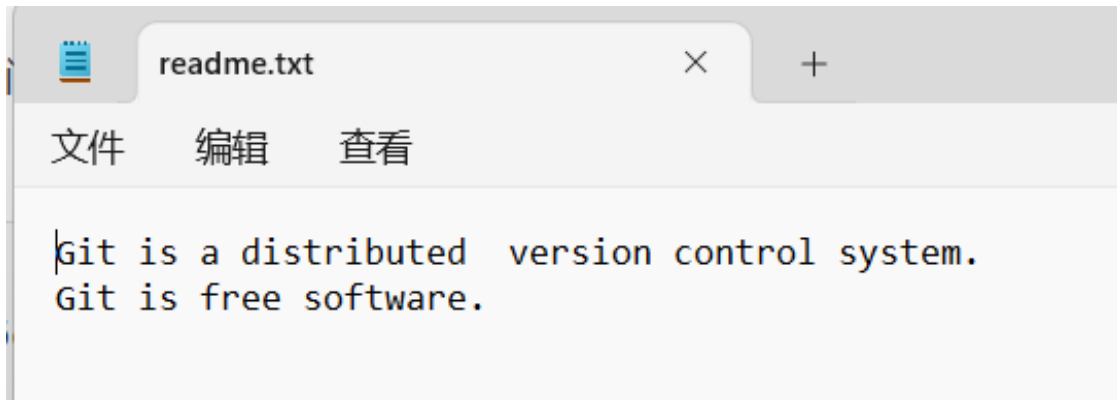
我们可以查看一下日志记录: 发现已经回退到了上一个版本。

```
MINGW64:/c/Users/86150/Desktop/learnGit
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
b5b7e39ea8780feeaf41d90f34e68fbb65d0993f (HEAD -> master) append GPL
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset --hard HEAD^
HEAD is now at c9f383c add distributed

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 (HEAD -> master) add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ |
```



(4) 我们可以发现最新的那个版本 append GPL 已经看不到了，那么我们怎么才能再切回来呢？

方法一：在当前命令行窗口没有关的情况下，我们往上滑动，可以找到 append GPL 那个提交版本的 commit id.

```
MINGW64:/c/Users/86150/Desktop/learnGit

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
b5b7e39ea8780feeaf41d90f34e68fbb65d0993f (HEAD -> master) append GPL
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset --hard HEAD^
HEAD is now at c9f383c add distributed

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 (HEAD -> master) add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file
```

那么根据这个 id 我们就可以退回到我们最新提交的 append GPL 的那个版本。


```
MINGW64:/c/Users/86150/Desktop/learnGit
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
b5b7e39ea8780feeaf41d90f34e68fbb65d0993f (HEAD -> master) append GPL
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset --hard HEAD^
HEAD is now at c9f383c add distributed

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 (HEAD -> master) add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset --hard b5b7e
HEAD is now at b5b7e39 append GPL

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ |
```

版本号没必要写很全，git 会自动去找，但是也不能过于短，不然会找到多个版本号，就无法确定是哪一个了。我们现在去看一下当前是哪个版本：回退成功。

```
MINGW64:/c/Users/86150/Desktop/learnGit
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset --hard HEAD^
HEAD is now at c9f383c add distributed

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 (HEAD -> master) add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset --hard b5b7e
HEAD is now at b5b7e39 append GPL

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --pretty=oneline
b5b7e39ea8780feeaf41d90f34e68fbb65d0993f (HEAD -> master) append GPL
c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ |
```

方法二：在当前窗口关闭的情况下，我们可以打开一个新的 Git Bash 界面，我们可以使用 `git reflog` 来记录我们每一次的命令，在找到最新版本的 commit id 后我们可以再采用方法一中的方式回到最新的版本。

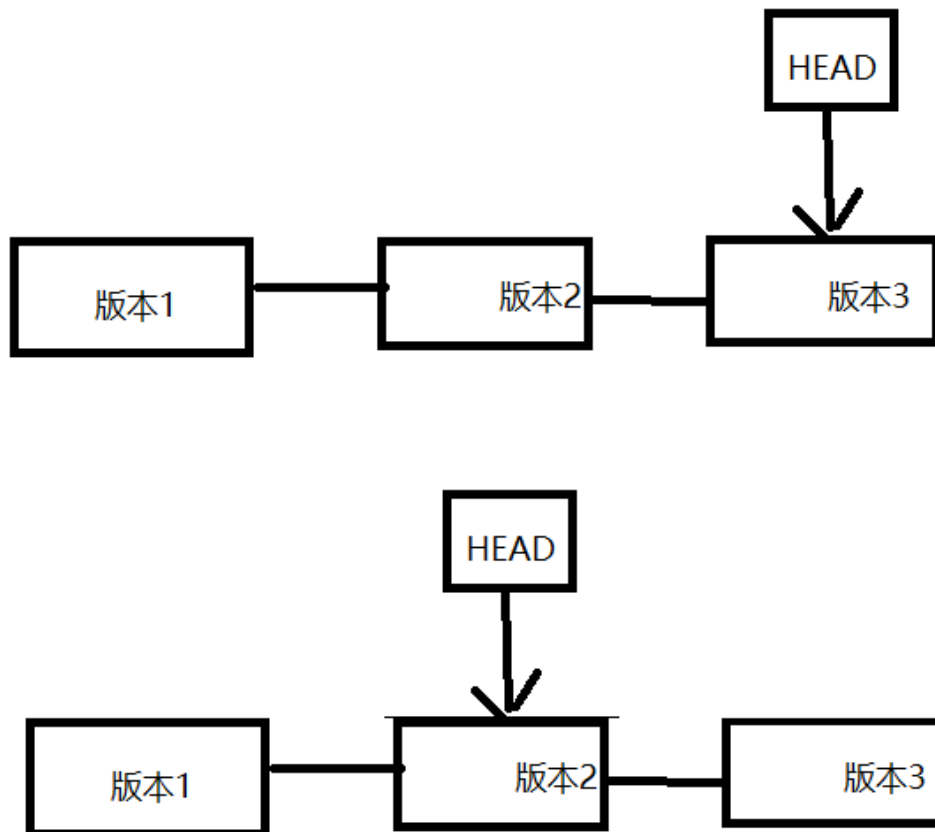
```

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reflog
b5b7e39 (HEAD -> master) HEAD@{0}: reset: moving to b5b7e
c9f383c HEAD@{1}: reset: moving to HEAD^
b5b7e39 (HEAD -> master) HEAD@{2}: commit: append GPL
c9f383c HEAD@{3}: commit: add distributed
ca5d95e HEAD@{4}: commit (initial): wrote a readme file

```

Git 的版本回退原理：

在 Git 内部有一个 HEAD 指针指向当前版本，当我们需要进行版本回退的时候，仅仅改变指针的指向就可以了，所以在 Git 中版本的回退速度是很快，同时把工作区的内容给更新了。



git log 和 git reflog 的区别：

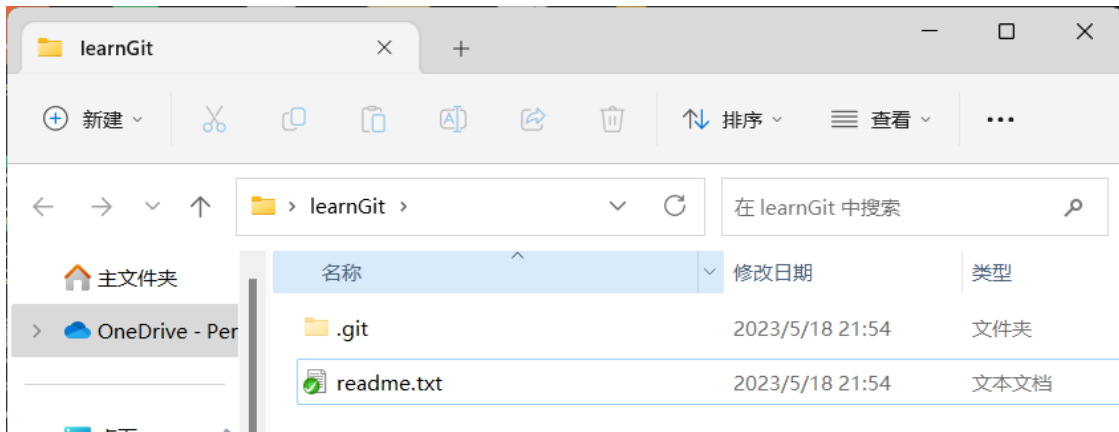
git log 和 git reflog 的最大区别是能不能查询到被删除的 commit 记录和 reset 的操作记录，log 不能，而 reflog 可以；所以以后要买后悔药就去找 reflog。

2.工作区和暂存区之间的概念

Git 和其他的版本控制系统如 SVN 的一个不同之处就是有暂存区的概念。

工作区（Working Directory）

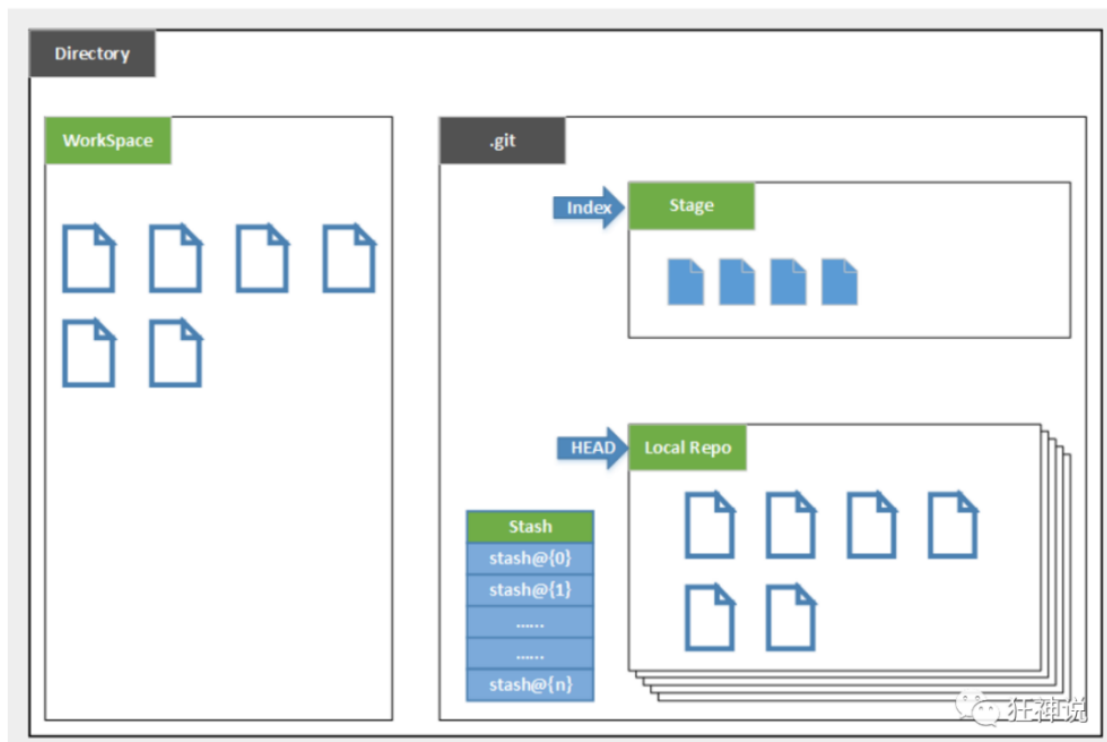
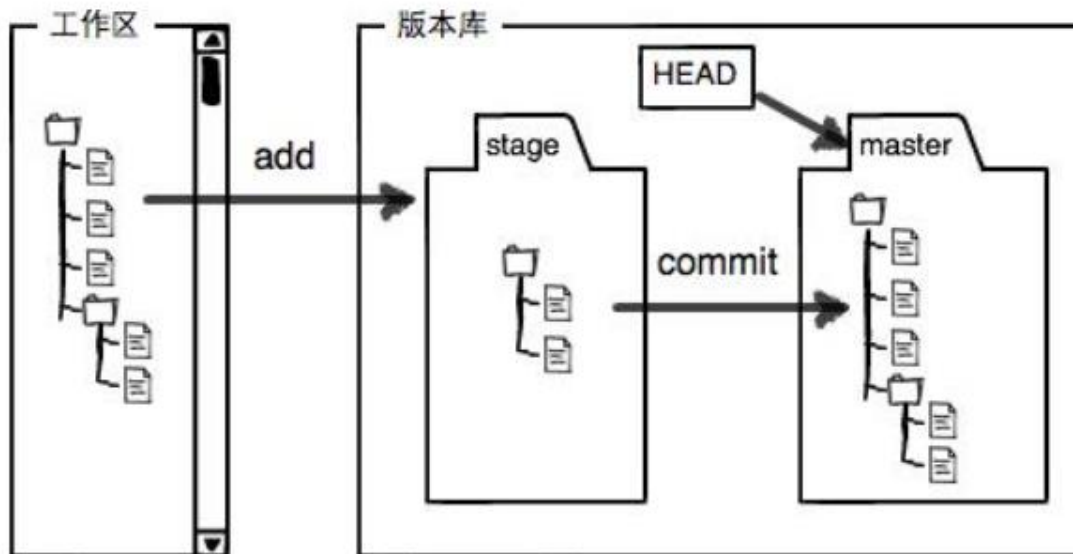
就是咱们电脑可以看到的目录，就相当于一个文件夹，我就是我们平时放代码的地方：



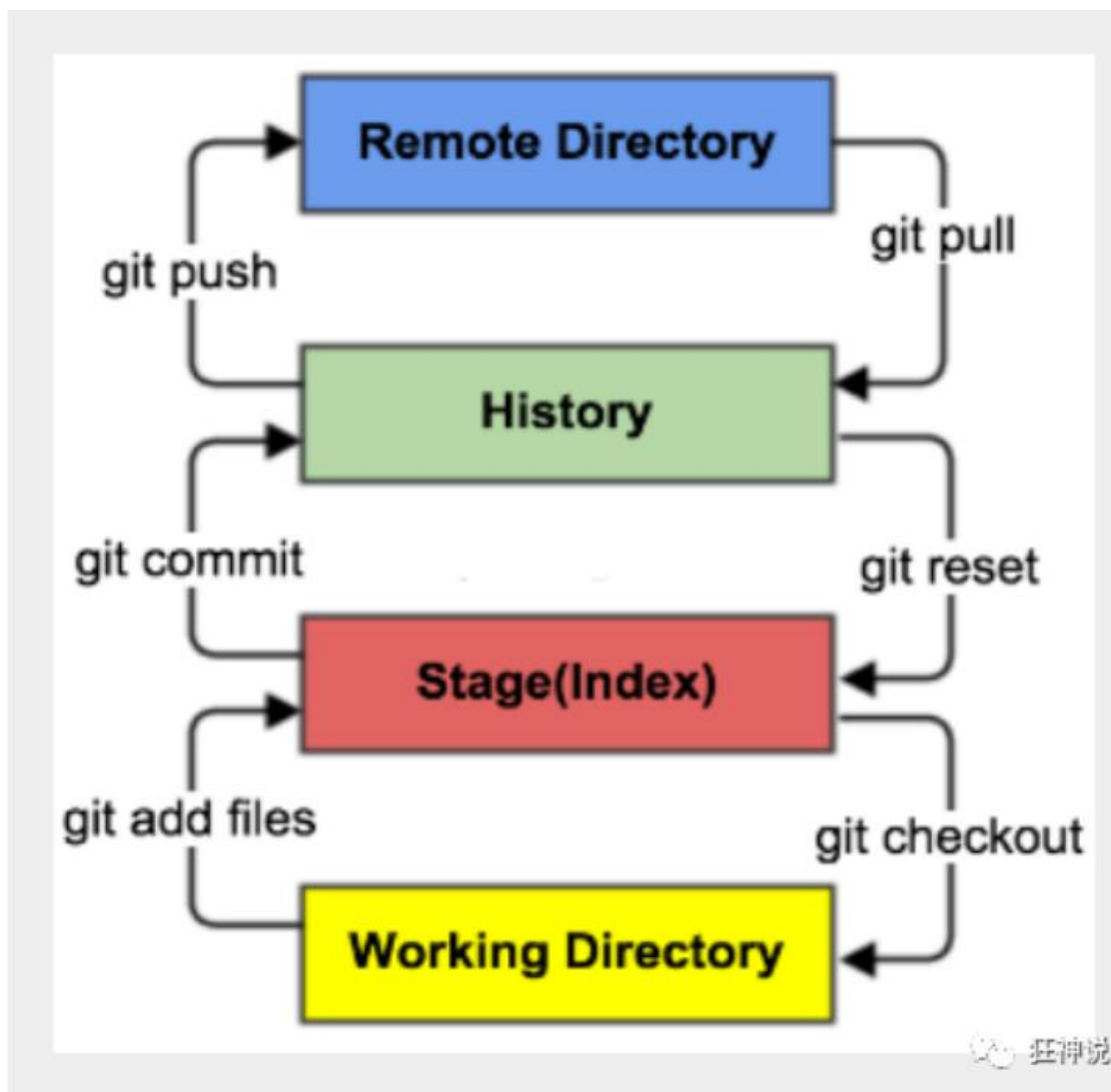
版本库（Repository）

在我们的文件夹中也就是工作区里，有一个.git 文件，这个就不算工作区，而是 git 的版本库。

在 Git 的版本库里面有很多的东西，其中最重要的就是 stage（或者叫 index）的暂存区，还有 Git 为我们创建的第一个分支 master,以及指向 master 分支的指针 Head



大致的工作流程以及不同指令之间进行的切换



- **Workspace:** 工作区，就是你平时存放项目代码的地方
- **Index / Stage:** 暂存区，用于临时存放你的改动，事实上它只是一个文件，保存即将提交到文件列表信息
- **Repository:** 仓库区（或本地仓库），就是安全存放数据的位置，这里面有你提交到所有版本的数据。其中 **HEAD** 指向最新放入仓库的版本
- **Remote:** 远程仓库，托管代码的服务器，可以简单的认为是你项目组中的一台电脑用于远程数据交换

现在我们来实际操作一下：

第一步：我们可以用 `git status` 来看一下状态，发现现在工作区是“干净”的，没有对工作区做出任何修改。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
nothing to commit, working tree clean
```

第二步：现在我们在工作区添加一个新的文件，license.txt 内容随便写点。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    license.txt

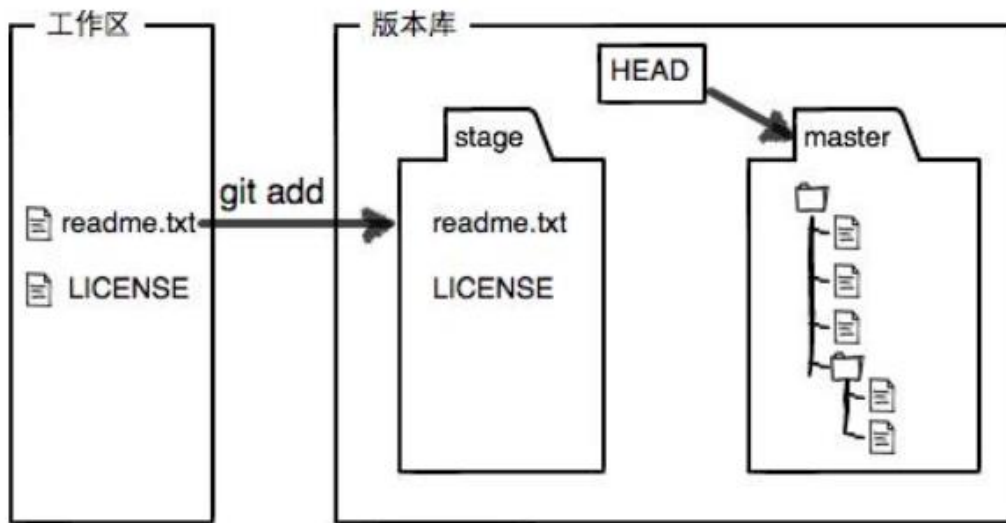
nothing added to commit but untracked files present (use "git add" to track)
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ |
```

LICENSE 还没有使用 `git add` 命令添加过，也就是只存在于工作区而不在暂存区，所以它的状态是 `Untracked`

第三步：我们使用 `git add` 命令将 `license` 文件添加到暂存区。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git add license.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   license.txt
```

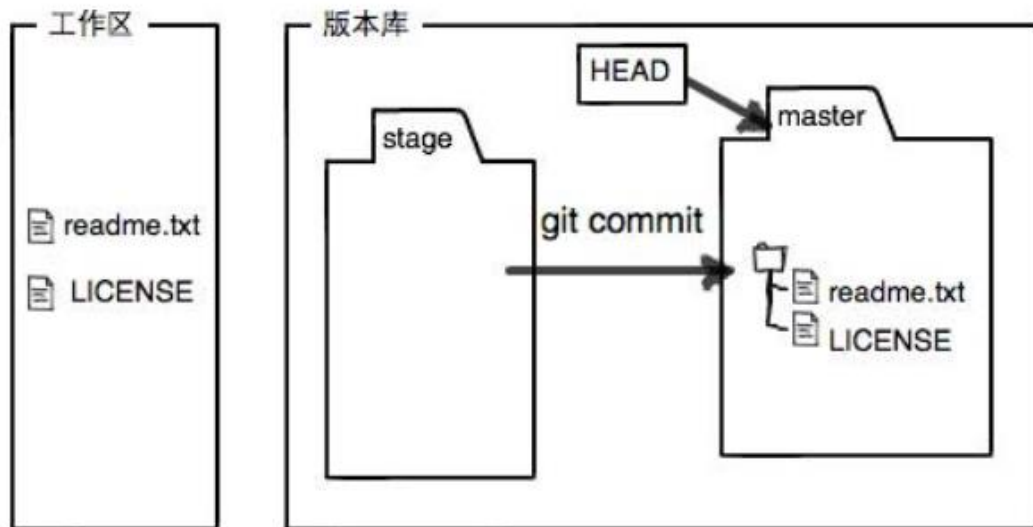


我们可以看到文件已将被添加到了暂存区，但是还没有被提交（没有被提交到本地库）。

第四步：我们将暂存区的文件进行提交，然后使用 `git status` 看一下状态。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git commit -m "understand stage how work"
[master b443727] understand stage how work
1 file changed, 1 insertion(+)
create mode 100644 license.txt

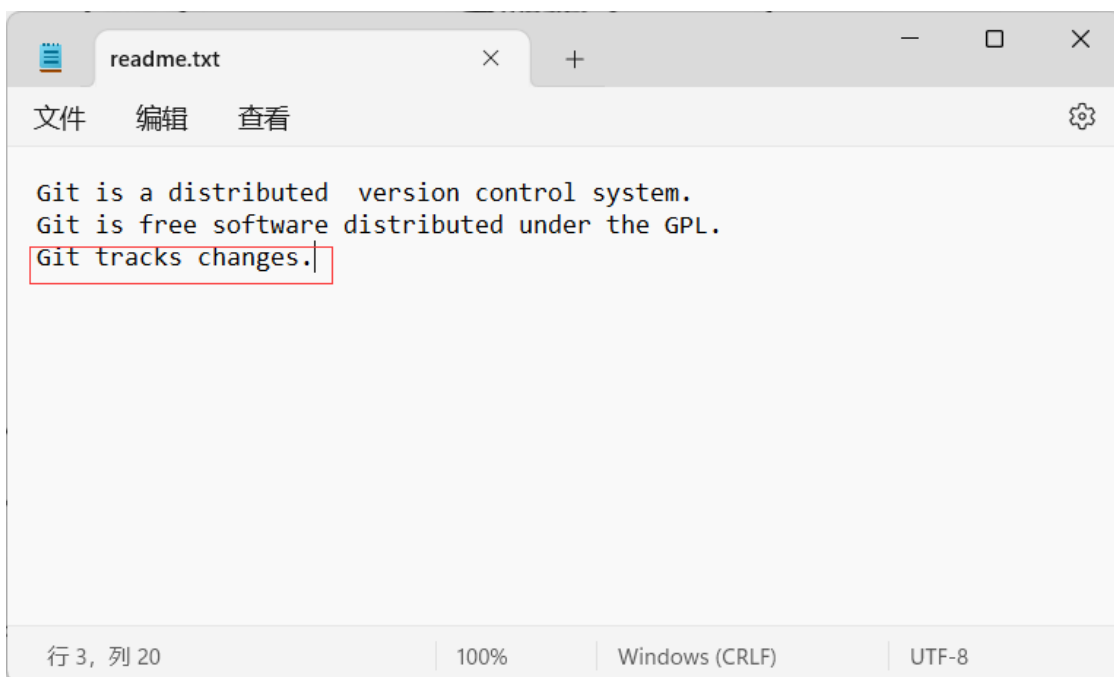
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
nothing to commit, working tree clean
```



现在暂存区的东西都被提交了，没有任何东西了。

3.管理修改

第一步：我们在 `readme.txt` 文件里面新增加一行内容。



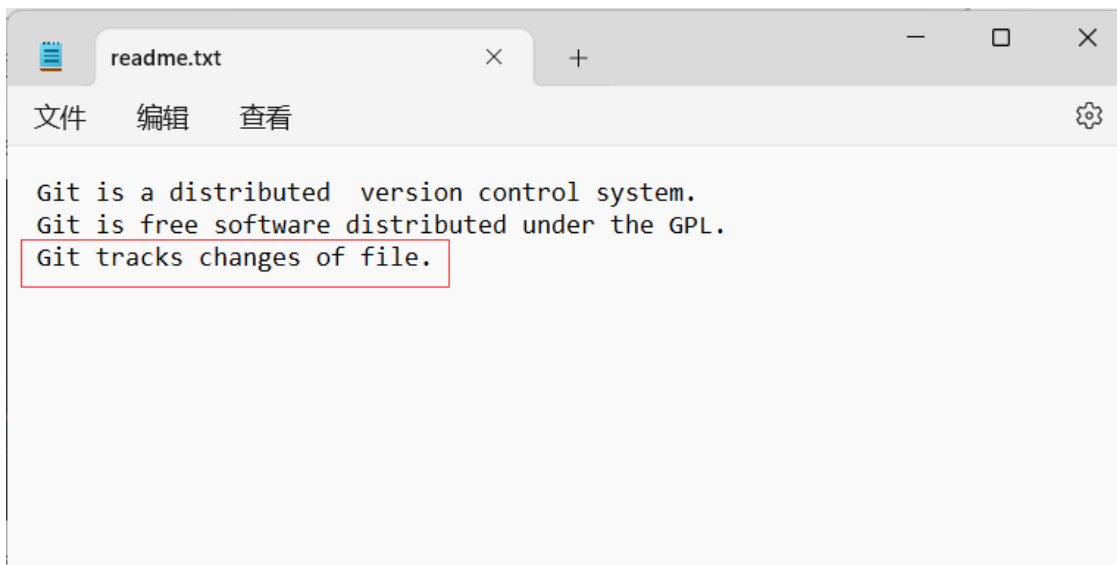
然后我们将 `readme.txt` 文件添加到暂缓区中，并查看一下状态。


```
MINGW64:/c/Users/86150/Desktop/learnGit

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git add readme.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.txt
```

第二步：我们把文件内容改一下，并进行提交，然后查看一下版本库的状态。



```

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git commit -m "git tracks changes"
[master 17b819c] git tracks changes
1 file changed, 2 insertions(+), 1 deletion(-)

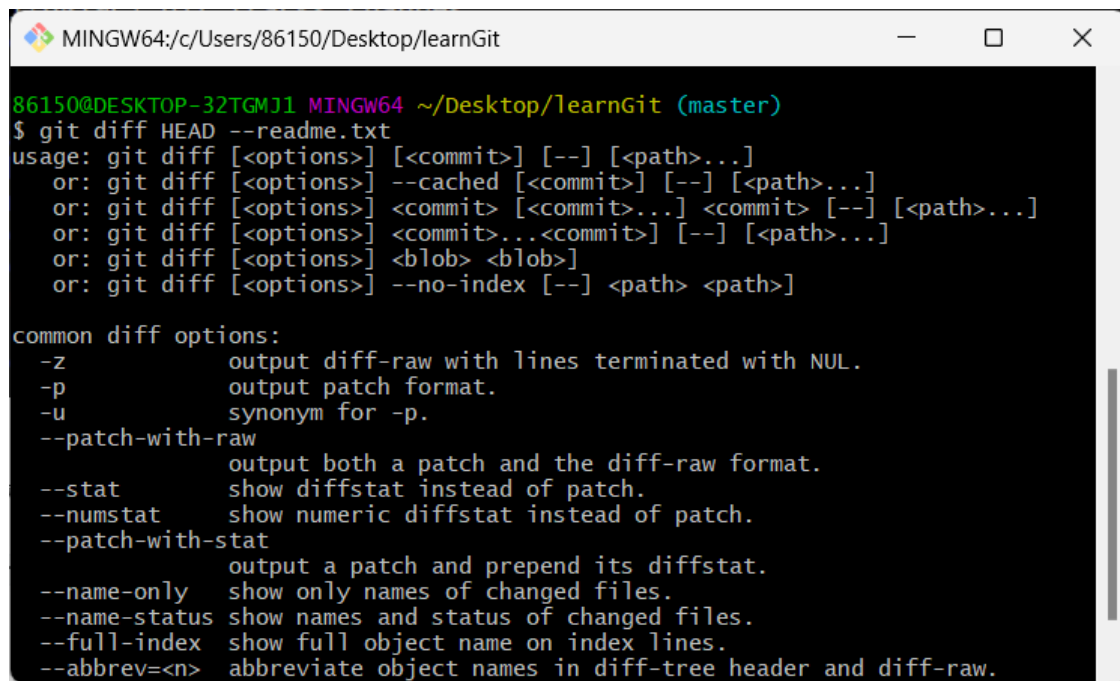
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.txt

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

```

我们的第二次修改没有被提交，原因是因为我们在第二次修改之后，没有将第二次的修改文件 add 到暂存区，所以在提交之前我们应该将文件 add 到在暂存区，然后一次性提交。

提交后，用 `git diff HEAD -- readme.txt` 命令可以查看工作区和版本库里面最新版本的差别：



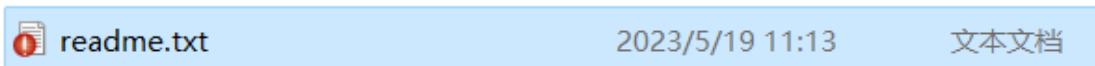
```

MINGW64:/c/Users/86150/Desktop/learnGit
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git diff HEAD --readme.txt
usage: git diff [<options>] [<commit>] [--] [<path>...]
or: git diff [<options>] --cached [<commit>] [--] [<path>...]
or: git diff [<options>] <commit> [<commit>...] <commit> [--] [<path>...]
or: git diff [<options>] <commit>...<commit> [--] [<path>...]
or: git diff [<options>] <blob> <blob>]
or: git diff [<options>] --no-index [--] <path> <path>]

common diff options:
-z          output diff-raw with lines terminated with NUL.
-p          output patch format.
-u          synonym for -p.
--patch-with-raw
            output both a patch and the diff-raw format.
--stat      show diffstat instead of patch.
--numstat   show numeric diffstat instead of patch.
--patch-with-stat
            output a patch and prepend its diffstat.
--name-only show only names of changed files.
--name-status show names and status of changed files.
--full-index show full object name on index lines.
--abbrev=<n> abbreviate object names in diff-tree header and diff-raw.

```

我们可以看到工作区域的文件有一个红色感叹号！



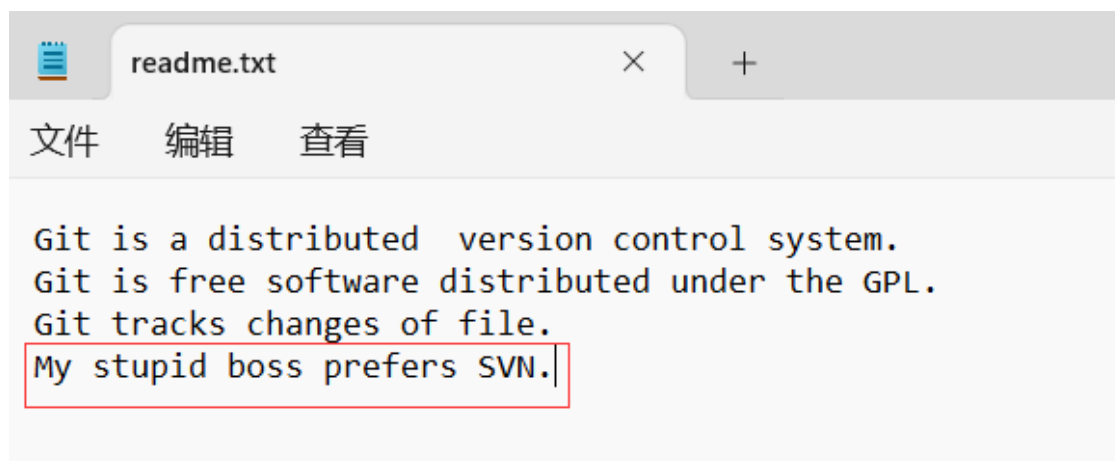
我们可以直接 add，然后提交到版本库里面去。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git commit -m "2 commit"
[master 612c660] 2 commit
1 file changed, 1 insertion(+), 1 deletion(-)
```

4.撤销修改

命令 `git checkout -- readme.txt` 意思就是，把 `readme.txt` 文件在工作区的修改全部撤销。（下面的两种情况都是撤销工作区的修改）

情况一：当我们在工作区的文件添加了一些不该加的内容，在还没 add 和提交的时候，你可以自己手动删除，或者使用命令 `git checkout -- readme.txt` 撤销刚刚的修改。这样工作区的内容就会回到和版本库一样的状态。（这个时候加入暂存区是空的，那么撤回的版本就是到最近一次的 commit 记录）



```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git checkout -- readme.txt
```

情况二：正确的版本已经 add 到了暂存区，然后你又在工作区中对文件做了修改，现在想要撤回到暂存区的那个版本

当然我们还是使用 `git checkout --readme.txt` 这个命令来进行撤销。（这个时候暂存区是有内容的，因为我们进行了 `add` 操作，这个时候使用撤回命令撤回到工作区的就是暂存区的内容）

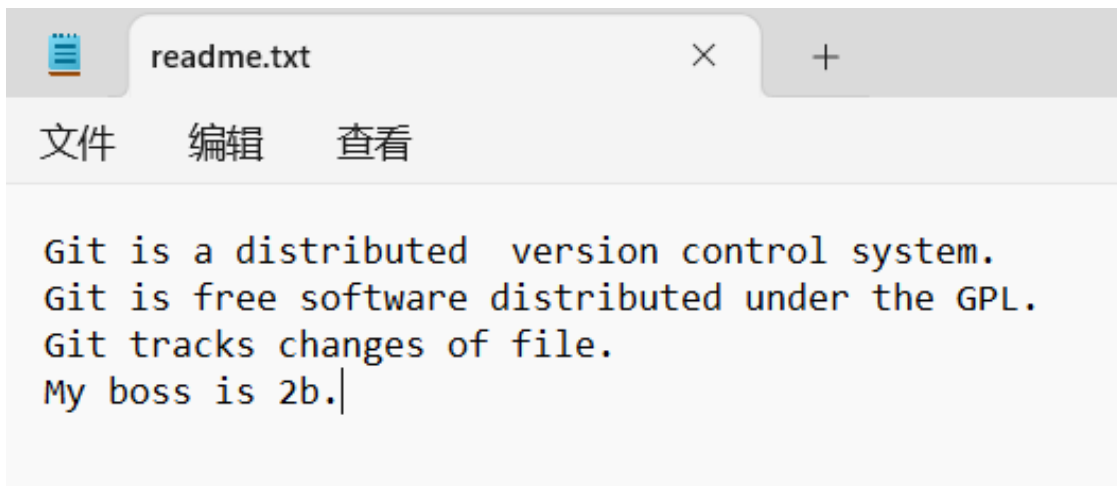
```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git add readme.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git checkout --readme.txt
error: unknown option `readme.txt'
usage: git checkout [<options>] <branch>
       or: git checkout [<options>] [<branch>] -- <file>...
```

在这个使用 `git checkout -- readme.txt` 这个文件的时候，大家需要注意的是--后面需要有一个空格，否则会报错。

总之，`git checkout -- readme.txt` 就是让这个文件回到最近一次 `git commit` 或 `git add` 时的状态。

撤销暂存区的修改



你一个不小心把这个文件添加到了暂存区里面去，这种大逆不道的话可不能被老板给看到，那我们怎么才能撤回暂存区的修改呢？

```

6 86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git add readme.txt
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.txt

```

我们可以看到文件只是被我们 add 到了暂存区，还没有提交到分支上面，咱们还有抢救的机会这个时候我们可以使用 `git reset HEAD readme.txt` 就可以把暂存区的修改给撤销掉。

```

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git reset HEAD readme.txt
Unstaged changes after reset:
M       readme.txt
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.txt

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

```

现在已经撤回到了工作区，那么怎么撤销工作区的修改呢？看上面 `git checkout -- readme.txt`

如果已经进行了提交但是还没有推送到远程仓库，怎么办呢？

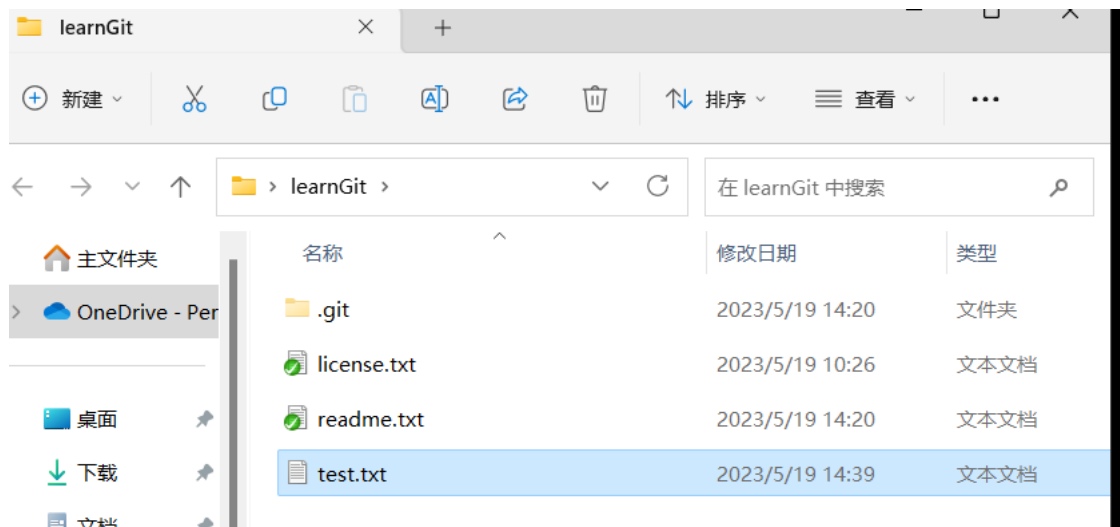
这个时候咱们可以使用版本的回退，上面有讲，命令 `git reset --hard HEAD^`，如果推送到了远程库直接 GG。

总结：

- 1.没有 `git add` 时，用 `git checkout -- file`
- 2.已经 `git add` 时，先 `git reset HEAD <file>` 回退到 1.，再按 1 操作
- 3.已经 `git commit` 时，用 `git reset` 回退版本

5.删除文件

我们先在工作区新建一个文件 `test.txt`



一般情况下，我们通常直接在文件管理器中把没用的文件删了，或者用 `rm` 命令删了：

```
MINGW64:/c/Users/86150/Desktop/learnGit

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git add test.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git commit -m "add test.txt"
[master 8c1cd1b] add test.txt
1 file changed, 3 insertions(+)
create mode 100644 test.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ rm test.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    test.txt

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

git 知道我们删除了哪个，我们可以选择删除或者删除版本库中的该文件。

恢复：因为版本库中这个文件还存在所以我们可以使用 `git checkout -- test.txt` 文件进行撤回。

删除版本库：使用 `git rm test.txt` 然后 `commit` 一下就可以了。现在文件就从版本库中删除掉了。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git rm test.txt
rm 'test.txt'

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git commit -m "remove test.txt"
[master 6c31f98] remove test.txt
1 file changed, 3 deletions(-)
delete mode 100644 test.txt
```

小提示：先手动删除文件，然后使用 `git rm <file>`和 `git add<file>`效果是一样的。

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！

三、远程仓库

为什么需要远程仓库？这里的远程仓库以 GitHub 为例


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。

1.添加远程仓库

首先，登陆 GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：我们输入仓库名，其他的配置保持默认，点击创建。

[import a repository.](#)

Owner *

 SundayYoung5 ▾

Repository name *

/ learngit

✔ learngit is available.

Great repository names are short and memorable. Need inspiration? How about [musical-pancake?](#)

Description (optional)

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

Initialize this repository with:

☐ Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None ▾

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

 You are creating a public repository in your personal account.

Create repository

根据 GitHub 的提示，在本地的仓库运行下面命令

```
git remote add origin https://github.com/SundayYoung5/learngit.git
```

Quick setup — if you've done this kind of thing before

 Set up in Desktop or **HTTPS** **SSH** `https://github.com/SundayYoung5/learngit.git` 

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# learngit" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/SundayYoung5/learngit.git
git push -u origin main
```

这个命令的意思是将本地的版本库和远程的仓库关联起来。

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。

我们把本地仓库的内容推送到远程，使用的是 **git push** 命令，实际上是把当前分支 **master** 推送到远程。

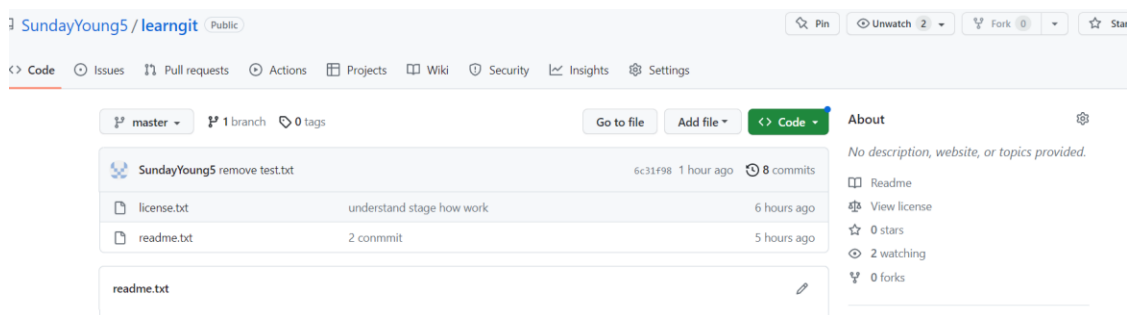
由于远程仓库是空的，我们第一次推送 **master** 分支时，加上了 **-u** 参数，**Git** 不但会把本地的 **master** 分支内容推送到远程的新的 **master** 分支，还会把本地的 **master** 分支和远程的 **master** 分支关联起来，在以后推送或者拉取的时候就可以简化命令。

下面我们可以把本地所有库的内容推送到远程库上面

```
git push -u origin master
```

如果用户是第一次提交，可能会设计到重新登录和输入密码的情况，具体解决办法参考下面这篇博客：

https://blog.csdn.net/qq_46780256/article/details/127285058



推送成功后，我们立刻可以在 **GitHub** 的页面中看到远程库已经和本地库一模一样了。

从现在起，只要本地做了提交，我们都可以通过命令

```
git push origin master
```

把本地分支的最新修改推送到 **GitHub**，现在你就拥有了真正的分布式版本库。

2.删除远程库

如果我们添加时地址写错了，或者就是想删除远程仓库可以使用 **git remote rm** 命令，在使用前可以先用 **git remote -v** 查看远程版本库的信息：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git remote -v
origin https://github.com/SundayYoung5/learngit.git (fetch)
origin https://github.com/SundayYoung5/learngit.git (push)
```

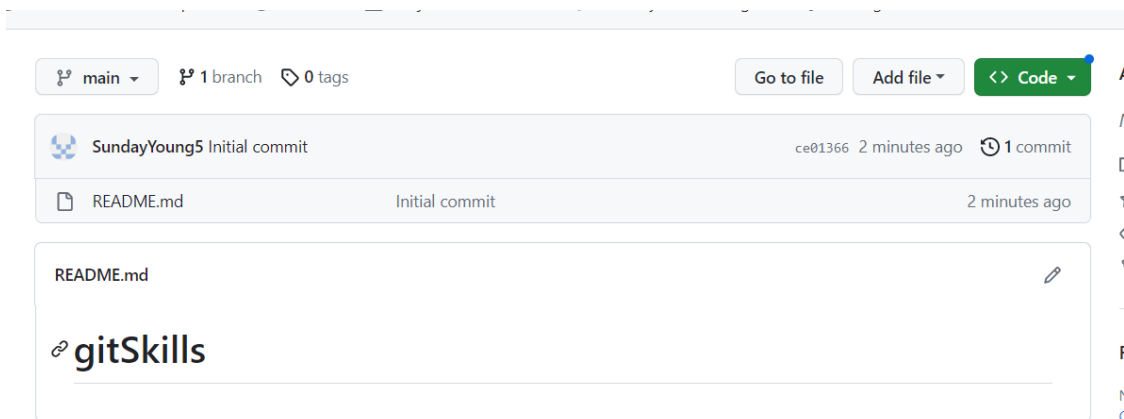
然后，根据名字删除，我们刚刚在连接远程库的时候名字叫做 **origin**:

```
git remote rm origin
```

此处的删除是解除和本地的远程绑定关系，并不是真正意义上的删除了远程库。远程库本身是没有任何改动的。要删除真正的远程仓库，我们需要去 **GitHub** 上进行删除操作。

3.从远程仓库克隆

(1) 在 **GitHub** 上创建一个新的仓库



(2) 我们可以使用 **ssh** 或者 **https** 进行克隆，使用 **ssh** 进行克隆的速度要比 **https** 链接克隆的速度快。

```
git clone 链接
```

Go to file

Add file ▾

<> Code ▾

Local

Codespaces New

Clone ?

HTTPS

SSH

GitHub CLI New

https://github.com/SundayYoung5/gitSkills.git

📋

Use Git or checkout with SVN using the web URL.

📂 Open with GitHub Desktop

📄 Download ZIP

About

No description, website, or

📖 Readme

☆ 0 stars

👁 1 watching

🍴 0 forks

Releases

No releases published

[Create a new release](#)

```
MINGW64:/c/Users/86150/Desktop/skills
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/skills
$ git clone https://github.com/SundayYoung5/gitSkills.git
Cloning into 'gitSkills'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 596 bytes | 54.00 KiB/s, done.
```

> skills > gitSkills >

在 gitSkills 中搜索 🔍

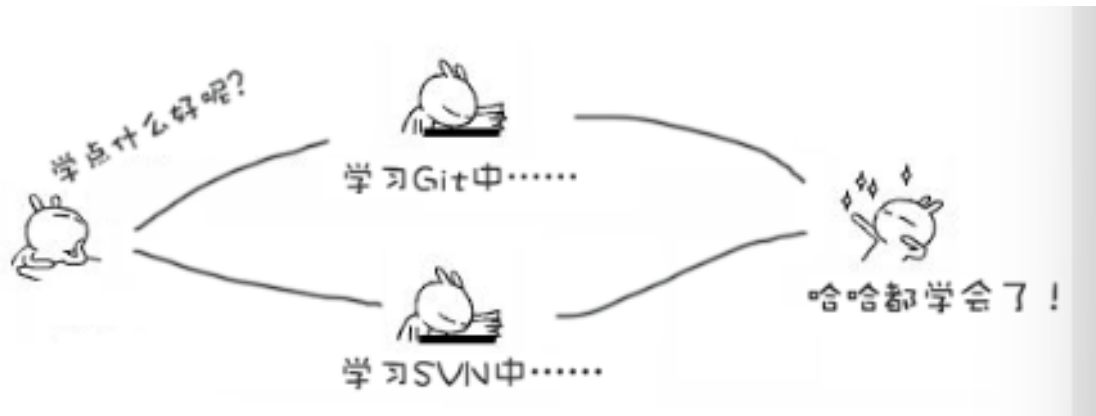
名称	修改日期	类型
📁 .git	2023/5/19 17:21	文件夹
📄 README.md	2023/5/19 17:21	Markdown F

可以看到克隆成功。

四、分支管理

什么是分支？

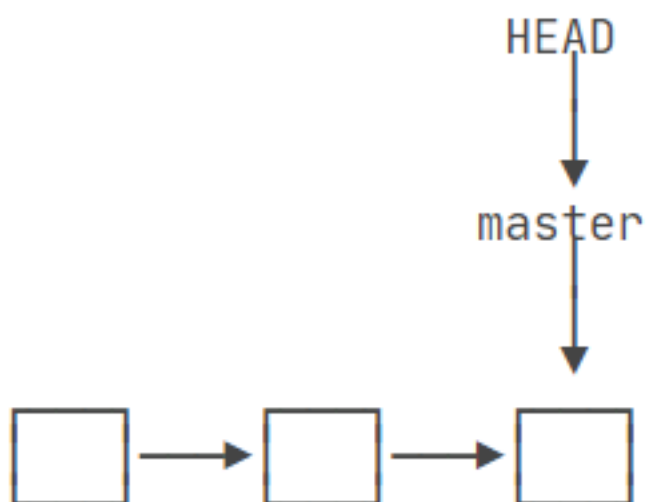
分支就是科幻电影里面的平行宇宙，当你正在电脑前努力学习 Git 的时候，平行宇宙的你正在努力学习 SVN。如果两个平行宇宙相互不干扰，那没也没啥影响，不过在某个时间节点，两个平行宇宙合并了，结果你即学会了 Git 又学会了 SVN。



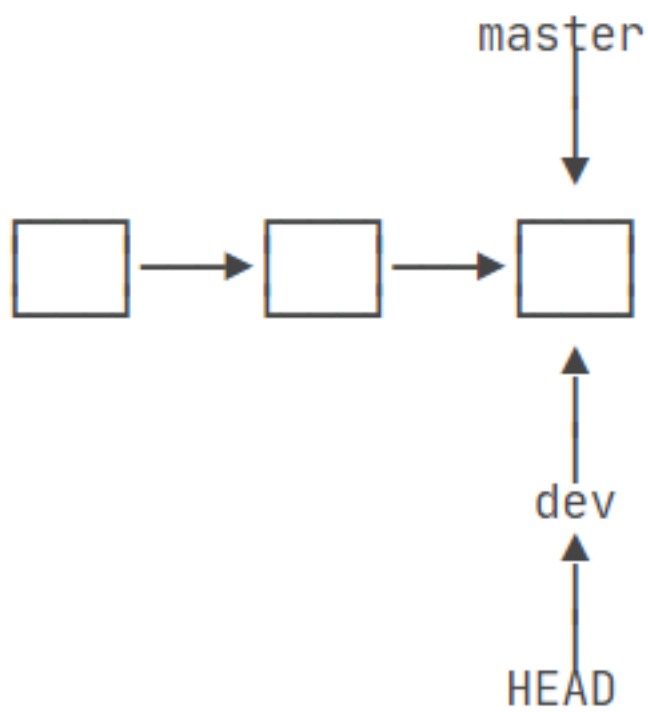
其他的版本控制系统也有分支比如 SVN，但是用过之后我们就会发现，这些版本控制系统创建和切换分支的速度很慢，简直让人无法忍受，但是 Git 的分支是与众不同的，无论创建、切换和删除分支，Git 都能在 1 秒钟内完成切换。

1. 创建与分支的合并

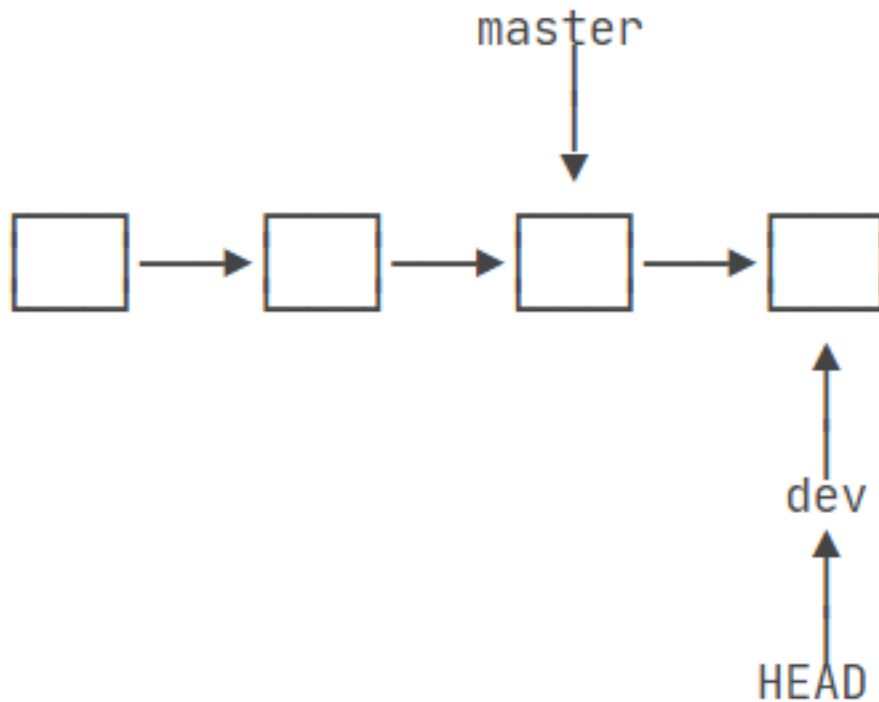
master 分支是是一条线，Git 用 master 指向最新的提交，再用 HEAD 指向 master，就能确定当前分支，以及当前分支的提交点，每次提交，master 分支就会向前移动一步，这样，随着你不断的提交，master 分支的线也会越来越长。



当我们创建了一个新的分支，例如 `dev` 时，Git 新建了一个指针叫 `dev`,指向 `master` 的相同提交，再把 `HEAD` 指向 `dev`,就表示当前分支在一个 `dev` 上：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，工作区的文件没有任何的变化。

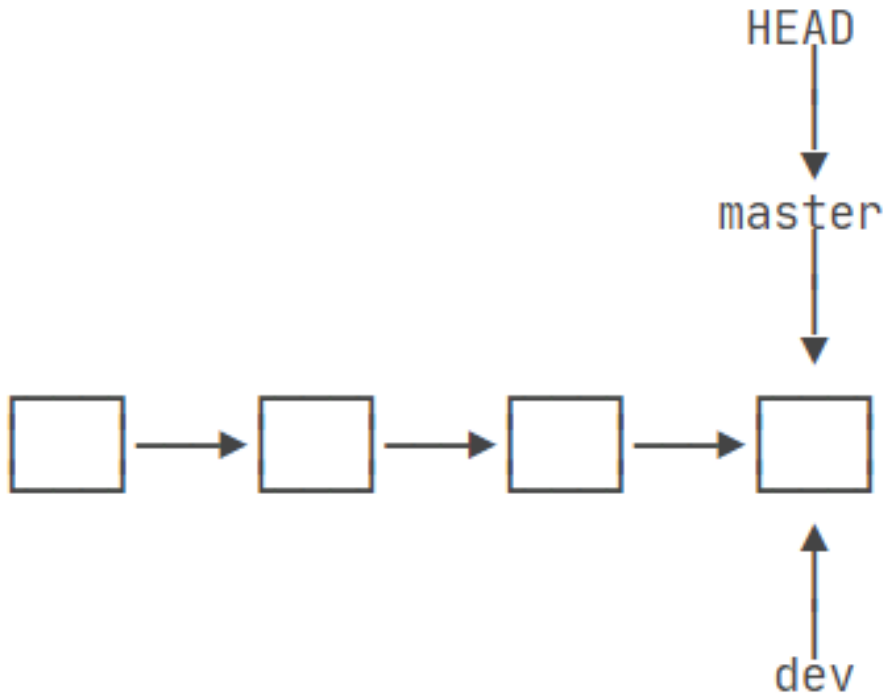


当 HEAD 指正指向 dev 分支的时候，对工作区的修改和提交就是针对 dev 分支了，比如新提交一次后，dev 指针往前移动一步，而 master 指正不变：



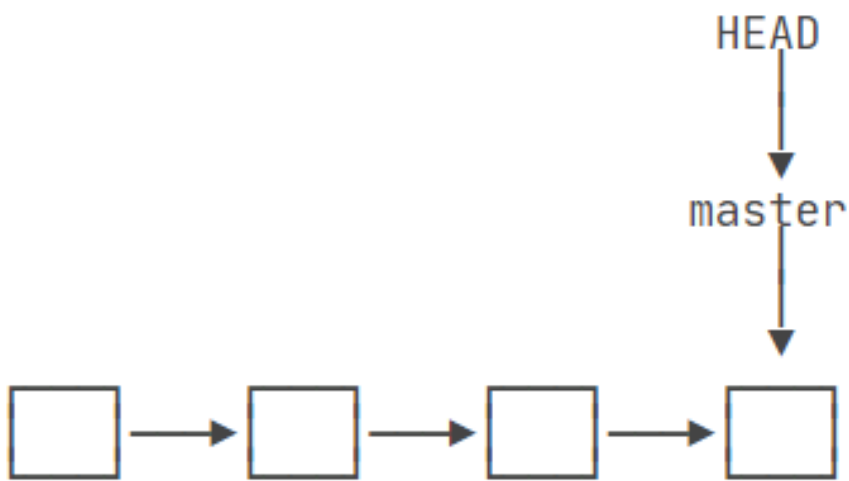
假如我们在 dev 上的工作完成了，就可以把 dev 合并到 master 上。Git 是怎么完成合并的呢？

最简单的方法就是把 master 指向 dev 的当前提交，就完成了合并。



为什么说 Git 合并分支的速度很快！因为改改指针就可以，而工作区的内容则可以做到不变。

合并完分支后，我们可以删除 dev 分支。删除 dev 分支就是把 dev 的指针给删掉，删除后，我们就剩下一条 master 分支：



下面我们来实际操作一下：

首先我们创建 dev 分支，然后切换到 dev 分支，切换的命令如下：

```
git checkout -b dev
```

A terminal window with a title bar 'MINGW64:/c/Users/86150/Desktop/learnGit'. The prompt is '86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)'. The user enters '\$ git checkout -b dev' and the output is 'Switched to a new branch 'dev''.

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git checkout -b dev
Switched to a new branch 'dev'
```

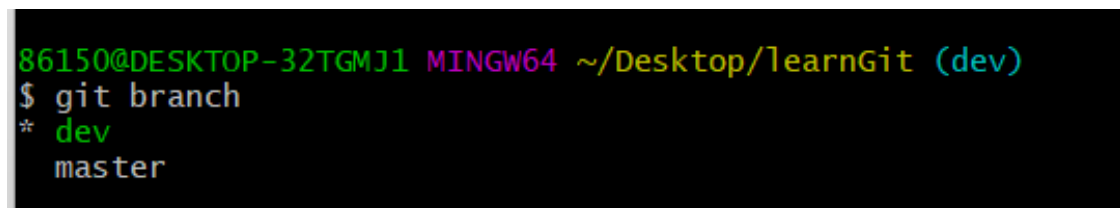
git checkout 命令加上 -b 参数表示创建并切换，相当于下面的两条命令：

```
git branch dev
```

```
git checkout dev
```

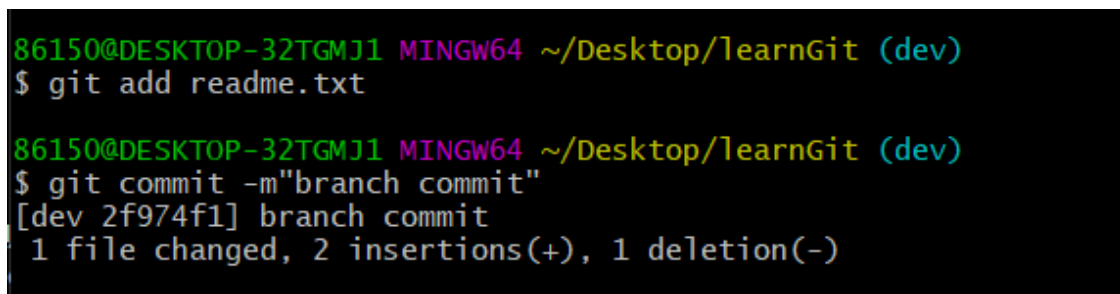
可以使用

git branch 命令会列出所有的分支，是当前分支的话前面会有一个*的符号，现在我们的修改提交都是在 dev 分支上。

A terminal window showing the prompt '86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)'. The user enters '\$ git branch' and the output is '* dev' and 'master'.

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git branch
* dev
master
```

我们现在对 readme.txt 文件加上一行 Creating a new branch is quick.

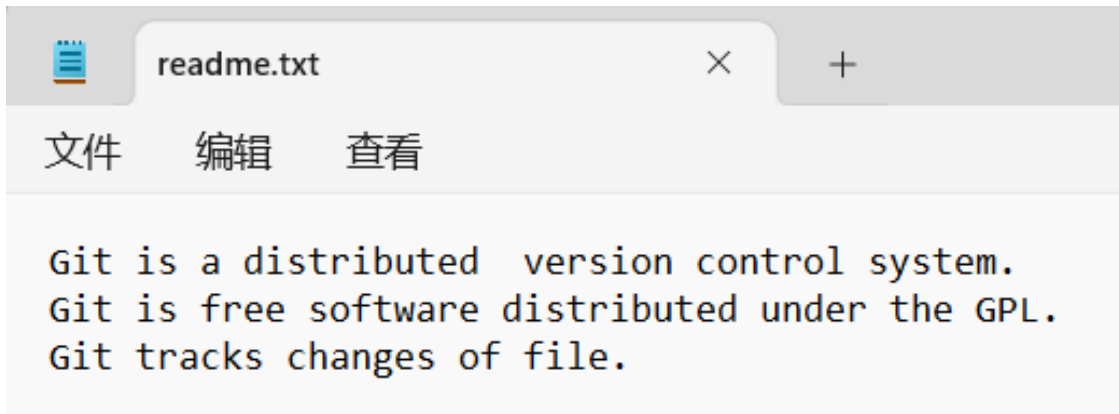
A terminal window showing two commands. First, the prompt is '86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)' and the user enters '\$ git add readme.txt'. Then, the prompt is '86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)' and the user enters '\$ git commit -m"branch commit"'. The output is '[dev 2f974f1] branch commit' and '1 file changed, 2 insertions(+), 1 deletion(-)'.

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git add readme.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git commit -m"branch commit"
[dev 2f974f1] branch commit
1 file changed, 2 insertions(+), 1 deletion(-)
```

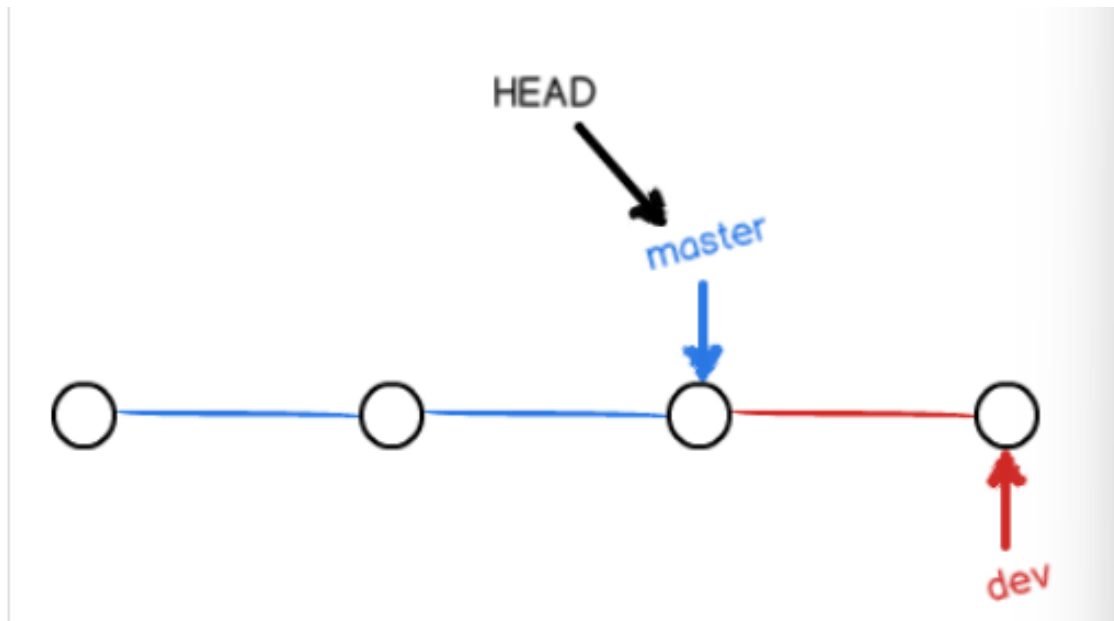
选择我们将当前分支切换到 master 分支上，然后 readme.txt 文件中新增加的一行内容会不见。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```



那主分支上的刚刚添加的内容为什么会不见呢？

因为刚刚提交的内容在 `dev` 分支上，而 `master` 分支此刻的提交点并没有变：

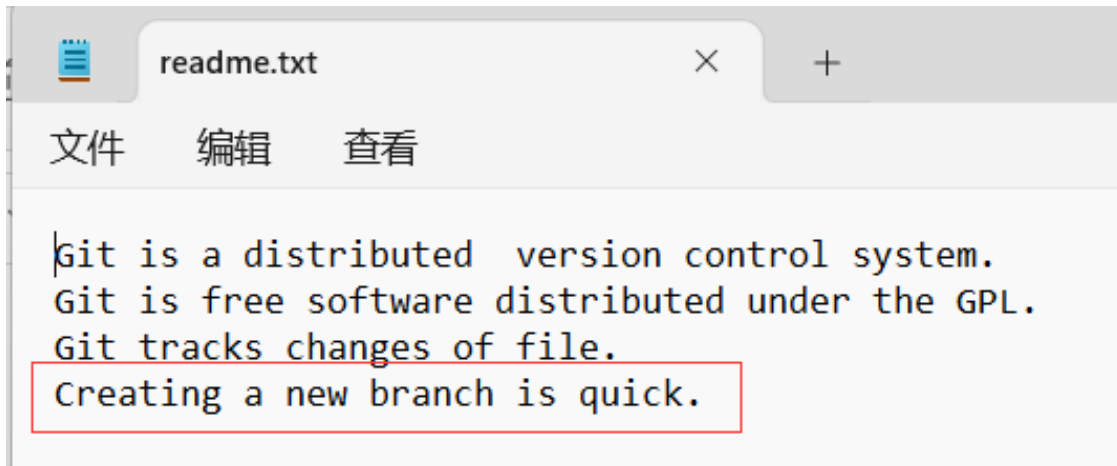


那么怎么才能让主分支上拥有 `dev` 分支上刚刚提交的内容呢？

我们需要把 `dev` 分支上的工作成果合并到 `master` 分支上去。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git merge dev
Updating 6c31f98..2f974f1
Fast-forward
 README.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

`git merge` 命令用于合并指定分支到当前分支，我们可以知道的是，当前分支是 `master` 分支，我们在合并后可以去看一下 `readme.txt` 文件中新添加的内容就有了。



我们可以注意到这个 Fast-forward 信息，Git 告诉我们，这次的合并是“快进模式”，也就是直接把 `master` 指向 `dev` 的当前提交，所以合并的速度是非常快的。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git merge dev
Updating 6c31f98..2f974f1
Fast-forward
 README.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

在分支合并完成后，我们就可以放心的删除 `dev` 分支了

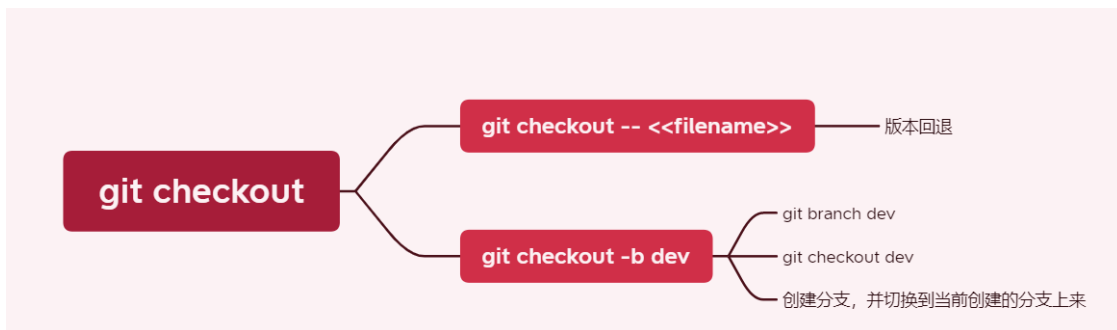
```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git branch -d dev
Deleted branch dev (was 2f974f1).
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git branch
* master
```

`git branch -d` 可以用来删除分支，我们可以看到，删除后就只剩下主分支。

在 Git 中，创建和删除分支的速度都很快，所以 Git 鼓励你使用分支完成某个任务，合并后再删除分支，这和直接在 `master` 分支上工作的效果是一样的，但是过程会更加的安全。

分支的切换



`git checkout` 既可以用来进行版本回退，也可以用来进行分支的创建和切换，实际上最新版本的 Git 提供了 `git switch` 命令来切换分支。

```
MINGW64:/c/Users/86150/Desktop/learnGit
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git switch -c dev
Switched to a new branch 'dev'
```

创建并切换新的 `dev` 分支，可以使用

```
git Switch -c dev
```

直接切换到已有分支

```
git switch master
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git branch
  dev
* master
```

2. 合并冲突的解决

准备一个新的分支 feature1 分支，继续我们新的分支开发：

```
MINGW64:/c:/Users/86150/Desktop/learnGit

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git switch -c feature1
Switched to a new branch 'feature1'
```

然后我们将 read.txt 文件的最后一行改为 Creating a new branch is quick AND simple.然后提交

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (feature1)
$ git add readme.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (feature1)
$ git commit -m"AND simple"
[feature1 29eb293] AND simple
1 file changed, 2 insertions(+), 1 deletion(-)
```

现在我们切换到 master 分支

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (feature1)
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
```

Git 会提示我们当前的 master 分支比远程的 master 分支要超前一个提交。

我们现在在 master 分支上把 readme.txt 文件的最后一行改为

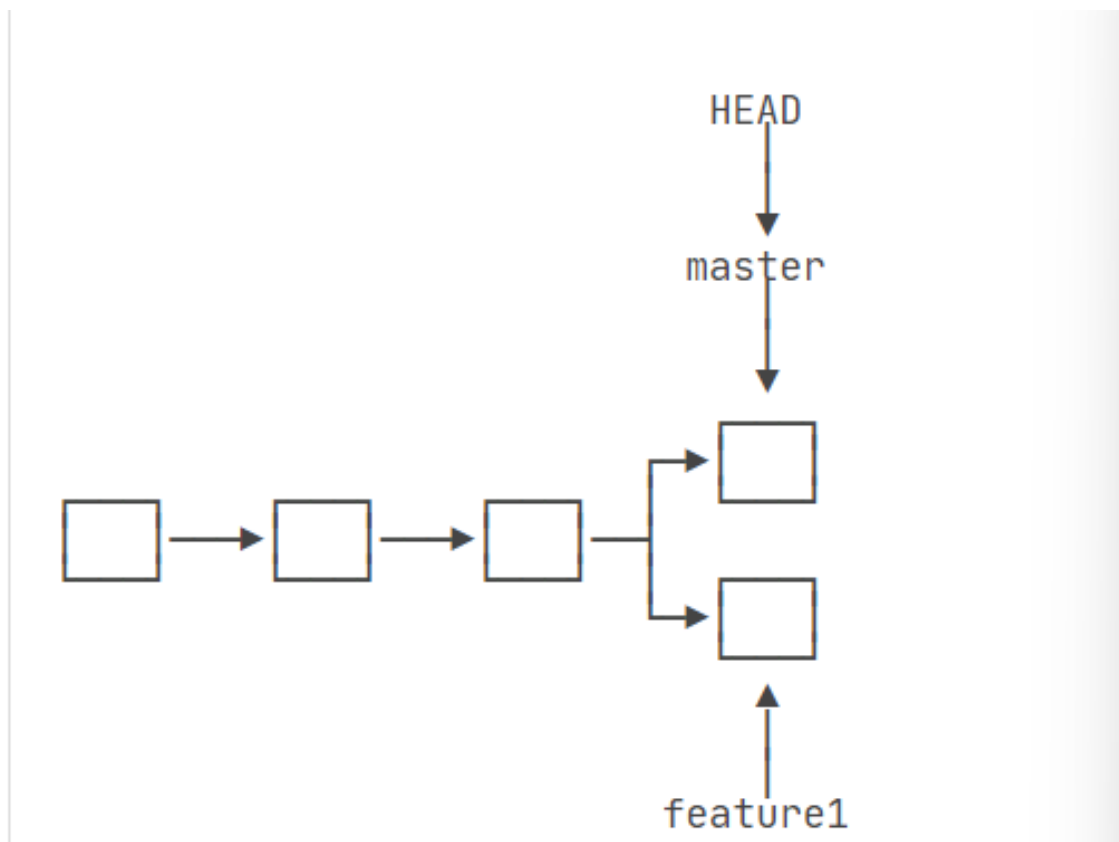
Creating a new branch is quick & simple.

现在我们将进行一次提交

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git add readme.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git commit -m"& simple"
[master 247a2cf] & simple
1 file changed, 2 insertions(+), 1 deletion(-)
```

现在，master 分支和 feature1 分支各自都分别有新的提交，变成了这样：



在这种情况下，Git 无法执行“快速合并”，只能试图将各自的修改合并起来，但是这种合并可能会有冲突。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

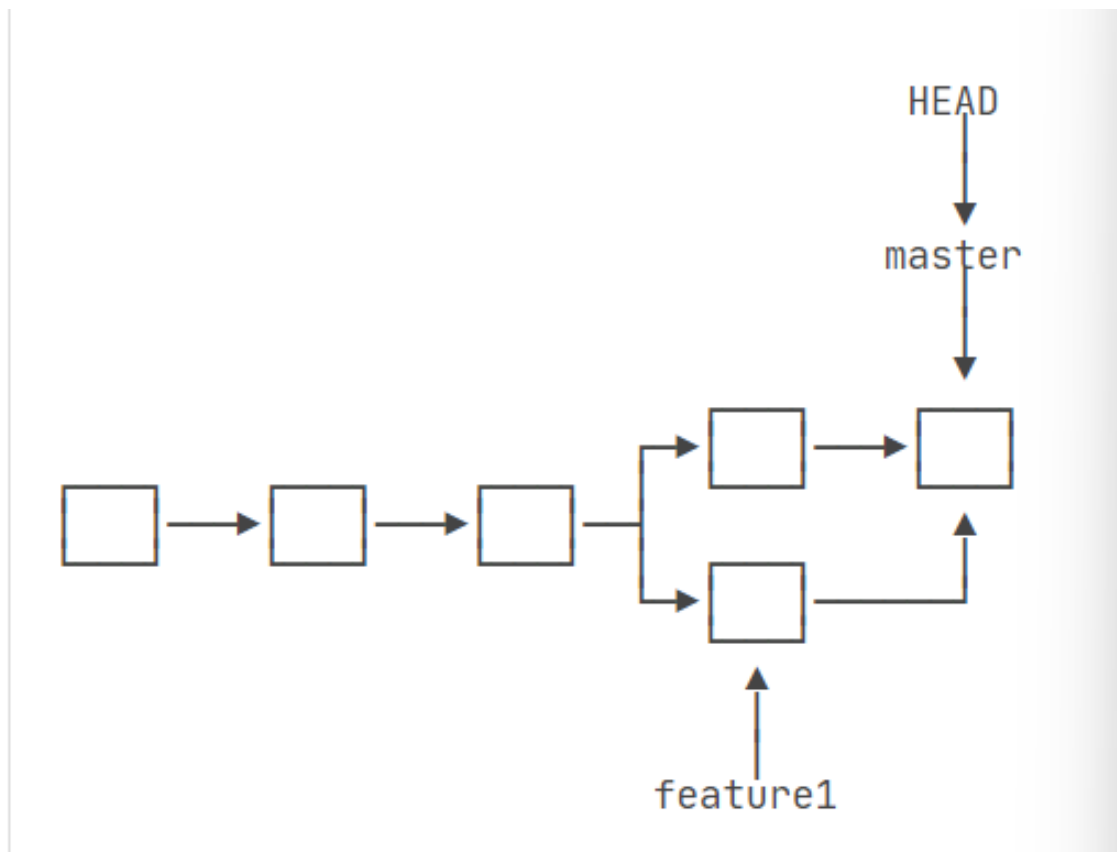
我们可以看到在合并的时候，readme.txt 文件存在冲突，我们需要将冲突手动解决后再提交。

我们在看一下 readme.txt 文件，可以看到冲突的内容。

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git tracks changes of file.
Creating a new branch is quick.
<<<<<<< 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，然后再进行提交。

现在 master 分支和 feature1 分支就变成了下图所示：



我们可以使用 `git log --graph` 可以看到分支的合并情况：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --graph --pretty=oneline --abbrev
* f2a6d88d5715ba26b3a426240dd2ddeb5c7d30d (HEAD -> master) conflict fixed
|
| * 29eb2934044d138fa0c2f288da4585db71e8750b (feature1) AND simple
| * | 247a2cfd2408b5cda864b21ece5e5c584e22a01c & simple
|/
* 2f974f1367a327b98e35f106cc817b838111542f branch commit
* 6c31f9864de9c0e8c5f637cbe369e670c8e37379 (origin/master) remove test.txt
* 8c1cd1b53488eb416986d19b95d8acee659cebc1 add test.txt
* 612c660d72e4b2e84b61da1bbf04f70728b415c9 2 commit
* 17b819c1fc5288fd11e6c5faf9b19c29820e775 git tracks changes
* b443727b5c5d41321c8c8a3bc259fd45cdef543b understand stage how work
* b5b7e39ea8780feeaf41d90f34e68fbb65d0993f append GPL
* c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
* ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file
```

最后删除 `feature1` 分支


```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git branch -d feature1
Deleted branch feature1 (was 29eb293).
```

3.分支策略管理

我们前面提到过，通常在分支合并的时候，采用的是 Fast-forward 模式，但是在这种模式下，删除分支后会丢掉分支的信息。

我们如果强制禁用 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的信息，

我们一般采用--no-ff 的方式 git merge 这样就可以强制禁用 Fast forward.

第一步：我们先创建一个 dev 分支并修改 readme.txt 文件，并提交。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git switch -c dev
Switched to a new branch 'dev'

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git add readme.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git commit -m"add merge"
[dev a3c399a] add merge
1 file changed, 1 insertion(+), 4 deletions(-)
```

第二步：现在我们切回到 master 分支，准备合并 dev 分支，请注意--no-ff 参数，表示禁用 Fast forward。

```

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git merge --no-ff -m"merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 5 +----
 1 file changed, 1 insertion(+), 4 deletions(-)

```

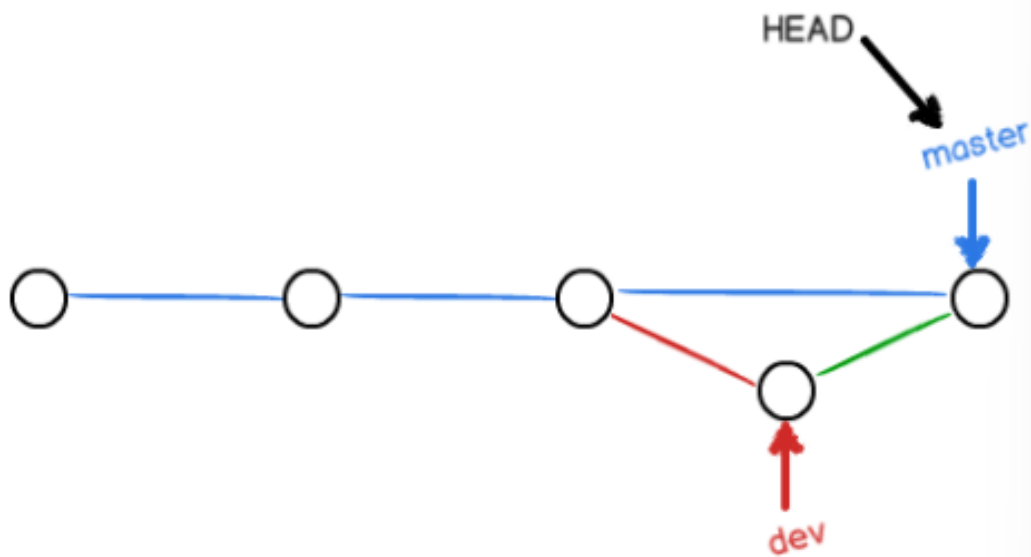
现在我们使用命令 `git log --graph` 命令查看一下分支历史

```

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --graph --pretty=oneline --abbrev
* db9ba552cc212bea1993654ed69cc796f80e2a24 (HEAD -> master) merge with no-ff
/ \
* a3c399a9498cb6a994f9c82fd8ffd1c550fa1b1c (dev) add merge
/ \
* f2a6d88d5715ba26b3a426240dd2ddeba5c7d30d conflict fixed
/ \
* 29eb2934044d138fa0c2f288da4585db71e8750b AND simple
* | 247a2cfd2408b5cda864b21ece5e5c584e22a01c & simple
/ \
* 2f974f1367a327b98e35f106cc817b838111542f branch commit
* 6c31f9864de9c0e8c5f637cbe369e670c8e37379 (origin/master) remove test.txt
* 8c1cd1b53488eb416986d19b95d8acee659cebc1 add test.txt
* 612c660d72e4b2e84b61da1bbf04f70728b415c9 2 commit
* 17b819c1fc5288fd11e6c5fafe9b19c29820e775 git tracks changes
* b443727b5c5d41321c8c8a3bc259fd45cdef543b understand stage how work
* b5b7e39ea8780feeaf41d90f34e68fbb65d0993f append GPL
* c9f383c36e2e2d35c6b3378f9a71f63827dc8103 add distributed
* ca5d95eca8d957b1247d71debe9e9a8df07eec70 wrote a readme file

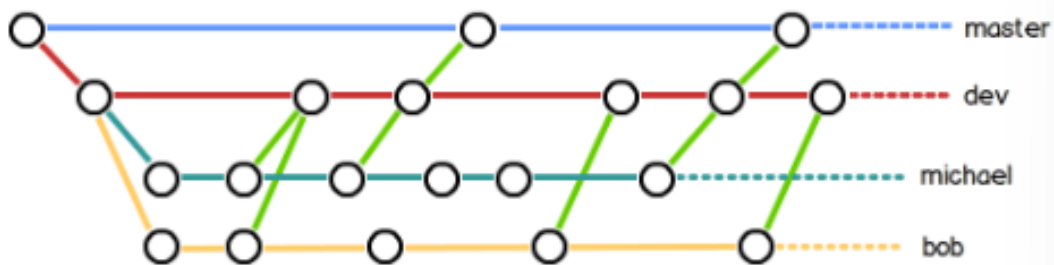
```

我们可以看到，不使用 Fast forward 模式，merge 后就像这样：



分支策略

1. **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。
2. 那我们在哪进行开发呢？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。
3. 项目的团队成员都在 **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，所以团队合作看起来就像是下图：



小结：

Git 分支是十分强大的，在团队开发中应该被充分的使用。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` 合并就看不出曾经做过合并。

4. Bug 分支

在日常开发中，我们难免会碰到 bug，有了 bug 我们就需要去修复，我们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。

当我们接到一个修复代号为 101 的 bug 时，我们可以创建一个 issue-101 来修复它，但是我们在 dev 上进行的工作还没有提交。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git status
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.txt

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

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git stash
Saved working directory and index state WIP on dev: a3c399a add merge

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git status
On branch dev
nothing to commit, working tree clean
```

我们可以使用 `git stash` 功能，把当前的工作现场给“储藏”起来，等以后恢复现场后继续工作。

现在假设我们需要在 master 分支上修复，并完成合并，最后删除 issue-101 分支：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git switch -c issue-101
Switched to a new branch 'issue-101'

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (issue-101)
$ git add readme.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (issue-101)
$ git commit -m"fix bug 101"
[issue-101 e04e60e] fix bug 101
1 file changed, 1 insertion(+), 1 deletion(-)

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (issue-101)
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
(use "git push" to publish your local commits)
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (issue-101)
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
(use "git push" to publish your local commits)

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git merge --no-ff -m"merged bug fix 101" issue-101
Merge made by the 'recursive' strategy.
 readme.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

现在 bug 修复完成了，我们可以切回 dev 分支继续工作了，但是我们可以发现工作区是干净的，那么我们刚刚的工作现场存到哪去了呢？

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git switch dev
Switched to branch 'dev'

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git status
On branch dev
nothing to commit, working tree clean
```

我们可以使用下面的命令进行查看

```
git stash list
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git stash list
stash@{0}: WIP on dev: a3c399a add merge
```

我们可以看到，工作现场还在，Git 把 stash 内容存放在某个地方了，但是需要恢复一下，有两个办法：

- 1.用 **git stash apply** 恢复，但是恢复后，stash 内容并不删除，我们需要使用 **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。

3.我们可以用多次 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}
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git stash apply stash@{0}
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.txt

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

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git stash list
stash@{0}: WIP on dev: a3c399a add merge

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git stash drop stash@{0}
Dropped stash@{0} (0645cc51d6372620f27694b0fa00e296c9ec6db0)

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git stash list
```

我们刚刚是在 master 分支修复的 bug, 我们的 dev 分支也是在修复前的 master 分支上分出来的, 也就是说刚刚修复的 master 分支上的 bug 其实在 dev 分支上也是存在的?

那么怎么在 dev 分支上修复同样的 bug?

重复操作一次还是又更加简便的方法, 当然是有更简单的方法。

同样的 bug, 要在 dev 上修复, 我们只需要把刚刚的提交修改“复制”到 dev 分支。注意: 我们只需要复制刚刚修复 bug 的那个提交, 并不是把整个 master 分支 merge 过来。

为了操作方便, Git 专门提供了一个 **git cherry-pick** 命令, 让我们能复制一个特定的提交到当前的分支: e04e60e2

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git cherry-pick e04e60e2
Auto-merging readme.txt
[dev 7044d7e] fix bug 101
Date: Mon May 22 16:21:00 2023 +0800
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
git cherry-pick 4c805e2
```

5.Feature 分支

在正式的项目开发中，我们总会需要添加很多新功能进去，我们肯定不能因为开发一个新分支而把主分支给搞坏了，所以我们现在一般都会新建一个 feature 分支，在这个上面进行开发，完成后合并，最后删除 feature 分支。

现在我们有了一个新的开发任务，代号为 Vulcan 的新功能，该计划用于下一代的星际飞船，所以我们可以创建一个分支用于开发了：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git switch -c feature-Vulcan
Switched to a new branch 'feature-Vulcan'
```

现在开发完毕，已经添加完成也提交了。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (feature-Vulcan)
$ git add Vulcan.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (feature-Vulcan)
$ git status
On branch feature-Vulcan
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   Vulcan.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (feature-Vulcan)
$ git commit -m"add feature vulcan"
[feature-Vulcan 8798840] add feature vulcan
1 file changed, 1 insertion(+)
create mode 100644 Vulcan.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (feature-Vulcan)
$
```

现在准备切回 dev，准备合并，但在这时候，由于预算不足，新功能需要被取消掉。

如果 `git branch -d feature-vulcan` 可以删除分支，如果删不掉可以强制进行删除 `git branch -D feature-vulcan` 来强删除。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ 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'.

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git branch -D feature-Vulcan
Deleted branch feature-Vulcan (was 51d322f).
```

6.多人协助

当我们从远程仓库 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，远程仓库的默认地址是 origin。要查看远程信息库的信息，使用 **git remote**：

 MINGW64:/c/Users/86150/Desktop/learnGit

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git remote
origin

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ |
```

或者我们可以使用 **git remote -v** 显示更加详细的信息：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git remote -v
origin https://github.com/SundayYoung5/learngit.git (fetch)
origin https://github.com/SundayYoung5/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。

（1）推送分支

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

推送主分支

```
git push origin master
```

推送其他分支

```
git push origin dev
```


但是并不是所有的分支都需要往远程推送，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，所以需要远程同步。
- **bug** 分支用于在本地修复 **bug**,就没必要推到远程了，除非老板要看你每周修复了几个 **bug**。
- **feature** 分支是否推送到远程，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操作，是否需要推送，完全取决于我们自己。

（2）抓取分支

现在我们在另一个文件夹中模拟，其他人克隆工作。

```
MINGW64:/c/Users/86150/Desktop/learn2
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learn2
$ git clone https://github.com/SundayYoung5/learngit.git
Cloning into 'learngit'...
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 22 (delta 5), reused 22 (delta 5), pack-reused 0
Unpacking objects: 100% (22/22), 1.78 KiB | 32.00 KiB/s, done.
```

当我们从远程 clone 时，默认情况下，我们只能看到 **master** 分支。不信我们可以使用 **git branch** 命令进行查看。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learn2/learngit (master)
$ git branch
* master
```

现在，大家需要在 **dev** 分支上开发，我们就必须创建远程的 **origin** 的 **dev** 分支到本地，于是可以用下面的这个命令来创建本地 **dev** 分支。

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

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/leanr2/learngit (master)
$ git checkout -b dev origin/dev
Switched to a new branch 'dev'
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

现在你的同事在 dev 上修改，然后还把 dev 分支 push 到远程：

```
MINGW64:/c/Users/86150/Desktop/leanr2/learngit
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/leanr2/learngit (dev)
$ git add env.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/leanr2/learngit (dev)
$ git commit -m"add env"
[dev 4768d14] add env
1 file changed, 1 insertion(+)
create mode 100644 env.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/leanr2/learngit (dev)
$ git push origin dev
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 297 bytes | 297.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SundayYoung5/learngit.git
7044d7e..4768d14 dev -> dev
```

现在你的同事已经向 origin/dev 分支推送了他的提交，而我們也需要对这个文件做出修改，并试图推送：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git add env.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git commit -m"add new env"
[dev c078a25] add new env
1 file changed, 1 insertion(+)
create mode 100644 env.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git push origin dev

To https://github.com/SundayYoung5/learngit.git
! [rejected]        dev -> dev (fetch first)
error: failed to push some refs to 'https://github.com/SundayYoung5/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.
```

推送发生了失败，Git 已经提示了我们，先用 git pull 把最新的提交从 origin/dev 抓下来，然后在本地合并，冲突解决，再推送：

```
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 277 bytes | 46.00 KiB/s, done.
From https://github.com/SundayYoung5/learngit
  7044d7e..4768d14  dev      -> origin/dev
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
```

我们发现 git pull 也失败了，根据提示，原因是没有指定 dev 与远程 origin/dev 分支的链接，根据提示，设置 dev 和 origin/dev 的链接：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git branch --set-upstream-to=origin/dev dev
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

我们在 pull 一下

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git pull
CONFLICT (add/add): Merge conflict in env.txt
Auto-merging env.txt
Automatic merge failed; fix conflicts and then commit the result.
```

这次 pull 成功，但是合并有冲突，需要手动解决一下，解决的方法和分支管理中解决冲突的方法一样。解决后，提交再 push:

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev|MERGING)
$ git add env.txt

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev|MERGING)
$ git commit -m"fix env conflict"
[dev e129967] fix env conflict

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git push origin dev
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 609 bytes | 609.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SundayYoung5/learngit.git
  4768d14..e129967  dev -> dev
```

提交成功。

因此，多人协助的工作模式通常是这样的：

- 1.首先，可以试图用 **git push origin** 推送自己的修改；
- 2.如果推送失败，因为远程分支比我们的本地要更新，需要先用 **git pull** 试图合并；
- 2.如果合并有冲突，则解决冲突，并在本地添加提交；
- 4.没有冲突或者解决掉冲突后，再用 **git push origin** 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没有创建，用命令：这就是多人协作的工作模式。

git branch --set-upstream-to origin/

7.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，最后 push 的童鞋不得不先 pull，在本地合并，然后才能 push 成功。

```

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master)
$ git log --graph --pretty=oneline --abbrev-commit
*   b81d373 (HEAD -> master) merged bug fix 101
|
| \
|  * e04e60e (issue-101) fix bug 101
| /
| *   db9ba55 merge with no-ff
| / \
| |  * a3c399a add merge
| | /
| | *   f2a6d88 conflict fixed
| | \
| |  * 29eb293 AND simple
| |  * | 247a2cf & simple
| | /
| * 2f974f1 branch commit
* 6c31f98 (origin/master) remove test.txt
* 8c1cd1b add test.txt
* 612c660 2 commit
* 17b819c git tracks changes
* b443727 understand stage how work
* b5b7e39 append GPL
* c9f383c add distributed
* ca5d95e wrote a readme file

```

如何才能让这个线变直一点呢？

我们可以使用 `git rebase` 命令。

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。

```

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master|REBASE 3/5)
$ git log --graph --pretty=oneline --abbrev-commit
* b806f6b (HEAD) fix
* 247a2cf & simple
* 2f974f1 branch commit
* 6c31f98 remove test.txt
* 8c1cd1b add test.txt
* 612c660 2 commit
* 17b819c git tracks changes
* b443727 understand stage how work
* b5b7e39 append GPL
* c9f383c add distributed
* ca5d95e wrote a readme file

```

可以看到的是远程的提交历史也是一条直线。

```
git rebase --quit
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (master|REBASE 3/5)
$ git rebase --quit
```

上面的命令可以退出 rebase。

五、标签管理

什么是标签管理？

发布一个版本的时候，我们通常会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。

Git 的标签虽然是版本库的快照，但其实它就是指向 commit 的指针（分支是可以移动的，但是标签是不会移动的），所以创建标签和删除标签都瞬间完成的。

创建标签

在 Git 中打标签是比较简单的，首先我们需要切换到要打标签的分支上。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git tag v1.0

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git tag
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，现在已经是星期五，但是周一的标签没有打，怎么办？

方法是找到历史提交的 commit id 然后打上标签就行了。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git tag v0.9 7044d7e
```

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` 查看标签信息。

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

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git tag -a v0.1 -m "version 0.1" 8c1cd1b

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git show v0.1
tag v0.1
Tagger: yanghao <1741601421@qq.com>
Date: Tue May 23 11:15:36 2023 +0800

version 0.1

commit 8c1cd1b53488eb416986d19b95d8acee659cebc1 (tag: v0.1)
Author: yanghao <1741601421@qq.com>
Date: Fri May 19 14:42:41 2023 +0800

    add test.txt

diff --git a/test.txt b/test.txt
new file mode 100644
index 0000000..0eb7e4c
--- /dev/null
+++ b/test.txt
@@ -0,0 +1,3 @@
+123
+456
+789
\ No newline at end of file
```

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。

操作标签

如果标签打错了，也可以删除：

```
git tag -d v0.1
```

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git tag -d v0.1
Deleted tag 'v0.1' (was 49b65cf)
```

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`

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git push origin v1.0
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SundayYoung5/learngit.git
 * [new tag]          v1.0 -> v1.0
```

或者，一次性推送全部尚未推送到远程的标签：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git push origin --tags
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SundayYoung5/learngit.git
 * [new tag]          v0.9 -> v0.9
```

如果标签已经推送到了远程，要删除远程标签就麻烦了一点，先从本地删除。

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git tag -d v1.0
Deleted tag 'v1.0' (was e129967)
```

然后再从远程删除。删除命令也是 `push`，但是格式如下：

```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git push origin :refs/tags/v1.0
To https://github.com/SundayYoung5/learngit.git
 - [deleted]          v1.0
```

要看看远程是否删除了标签，可以登录 GitHub 进行查看。

六、使用 GitHub

GitHub 简介：

我们一直使用 GitHub 作为免费的远程仓库，如果是个人的开源项目。如果是个人开源项目，放到 GitHub 上是完全没有问题的，然后 GitHub 还是一个开源协作的社区，通过 GitHub，既可以让别人参与你的开源项目，也可以参与别人的开源项目。

- 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

七、使用 Gitee

在使用 GitHub 的时候我们最常遇到的情况就是访问的速度太慢，有时候还会出现无法连接等情况。那么我们就可以使用国内的 Git 托管服务----Gitee

八、自定义 Git

Git 有很多可配置项，比如让 Git 显示颜色，会让命令看起来更加的醒目。

```
git config --global color.ui true
```

1.忽略特殊文件

为什么要忽略特殊文件？

有些时候，我们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等等，每次 `git status` 都会显示 `Untracked files.....`，有强迫症的兄弟们肯定受不了。

在 Git 中我们可以在 Git 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。

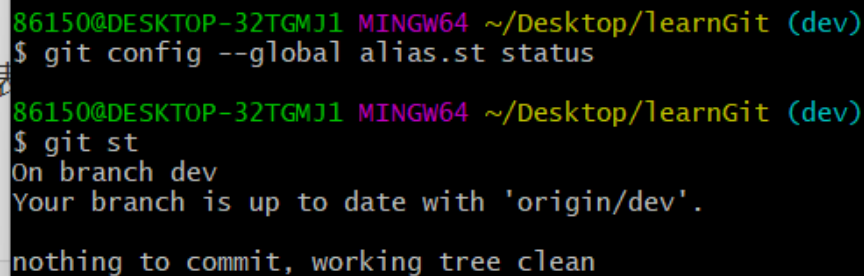
我们不需要从头写 `.gitignore` 文件，GitHub 已经为我们准备了各种配置文件，只需要组合一下就可以使用了。

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，那么自动生成生成的文件就没有必要放进版本库，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。

2.配置别名

在 Git 中我们可以简化指令的操作，我们可以设置别名简化命令的操作。



```
86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git config --global alias.st status

86150@DESKTOP-32TGMJ1 MINGW64 ~/Desktop/learnGit (dev)
$ git st
On branch dev
Your branch is up to date with 'origin/dev'.

nothing to commit, working tree clean
```

类似于这样的操作还有很多。

比如：

```
git config --global alias.co checkout  
git config --global alias.ci commit  
git config --global alias.br branch
```

3.搭建 Git 服务器

GitHub 就是一个免费托管开源代码的远程仓库，但是对于公司来说，产品的源码一般是商业机密，如果公司不想给 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 Linux 机器。

最后常用的 Git 命令清单可以点击这个链接下载：

<https://liao xuefeng.gitee.io/resource.liao xuefeng.com/git/git-cheat-sheet.pdf>

本文为廖雪峰老师 Git 教程的学习笔记，大家可以看廖雪峰老师写的 Git 教程通俗易懂，强烈推荐 <https://www.liao xuefeng.com>。