

版本控制系统：就是管理一个目录里面的文件，记录他们改了什么，谁改的，什么时候改的，还支持多人同时开发等等。

一：创建版本库

版本库又名仓库，英文名 **repository**，你可以简单理解成一个目录，这个目录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，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

1. 创建一个空目录，然后初始化

```
mkdir learngit
cd learngit
git init
```

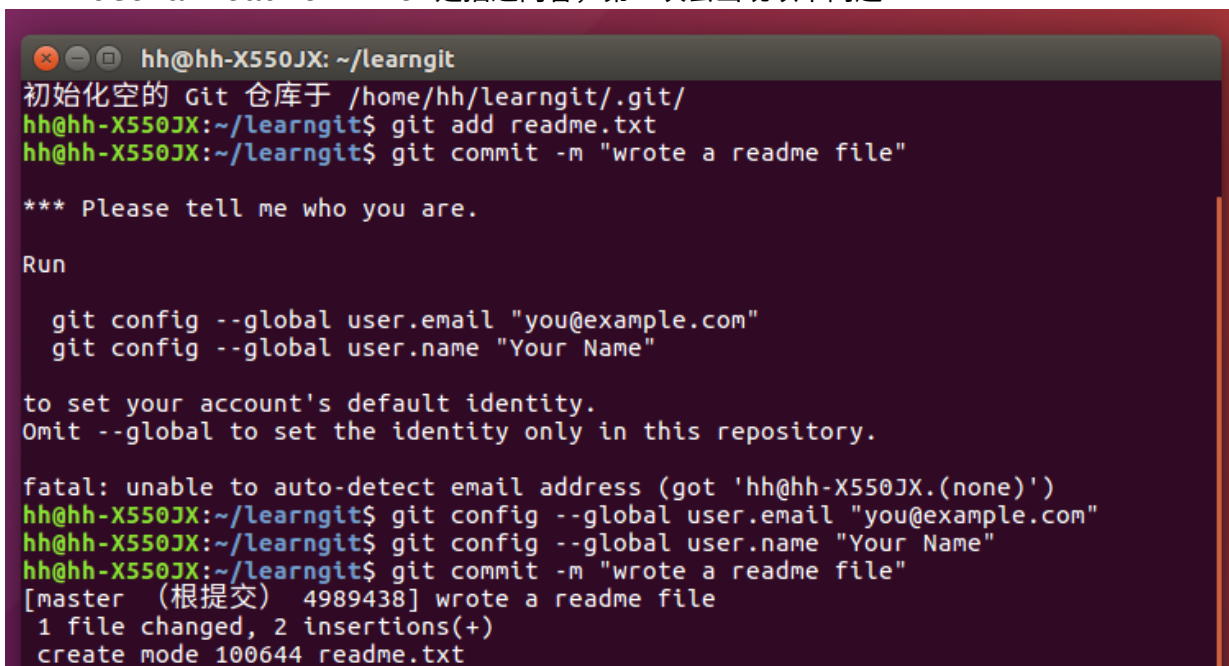
2. 添加文件：先写一个 txt 格式的文件 readme.txt，例如

```
Git is a version control system.
Git is free software.
```

然后输入命令行：

```
git add readme.txt
git commit -m "wrote a readme file"
```

"wrote a readme file" 是描述内容，第一次会出现以下问题

A terminal window with a dark background and light-colored text. The prompt is 'hh@hh-X550JX: ~/learngit'. The user runs 'git init', which outputs '初始化空的 git 仓库于 /home/hh/learngit/.git/'. Then they run 'git add readme.txt' and 'git commit -m "wrote a readme file"'. This leads to a prompt '*** Please tell me who you are.' followed by 'Run' and instructions to configure user identity. The user runs 'git config --global user.email "you@example.com"' and 'git config --global user.name "Your Name"'. Finally, they run 'git commit -m "wrote a readme file"' again, which successfully commits the file, showing '[master (根提交) 4989438] wrote a readme file' and '1 file changed, 2 insertions(+)'.

二：时光穿梭机

改变 readme.txt 里面的内容，如下

```
Git is a distributed version control system.
Git is free software
```

然后运行

```
git status
git diff
```

查看仓库的状态和具体修改的内容

```

hh@hh-X550JX: ~/learngit
hh@hh-X550JX:~$ git status
fatal: Not a git repository (or any of the parent directories): .git
hh@hh-X550JX:~$ cd learngit
hh@hh-X550JX:~/learngit$ git status
位于分支 master
尚未暂存以备提交的变更:
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改:      readme.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
hh@hh-X550JX:~/learngit$ git diff
diff --git a/readme.txt b/readme.txt
index 46d49bf..9247db6 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,2 @@
-Git is a version control system.
+Git is a distributed version control system.
 Git is free software.
hh@hh-X550JX:~/learngit$

```

然后再提交一次

```
git add readme.txt
```

```
git commit -m "add distributed"
```

```

hh@hh-X550JX: ~/learngit
hh@hh-X550JX:~/learngit$ git diff
diff --git a/readme.txt b/readme.txt
index 46d49bf..9247db6 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,2 @@
-Git is a version control system.
+Git is a distributed version control system.
 Git is free software.
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git status
位于分支 master
要提交的变更:
  (使用 "git reset HEAD <文件>..." 以取消暂存)

    修改:      readme.txt

hh@hh-X550JX:~/learngit$ git commit -m "add distributed"
[master 290726f] add distributed
 1 file changed, 1 insertion(+), 1 deletion(-)
hh@hh-X550JX:~/learngit$ git status
位于分支 master
无文件要提交, 干净的工作区
hh@hh-X550JX:~/learngit$

```

可穿插 git status 查看状态

1. 版本回退, 再改一次, 改成

Git is a distributed version control system.

Git is free software distributed under the GPL.

现在一共经历了三个版本，用

`git log`

查看记录

`git log --pretty=oneline`

查看简约版记录

```
hh@hh-X550JX: ~/learngit
1 file changed, 1 insertion(+), 1 deletion(-)
hh@hh-X550JX:~/learngit$ git log
commit 9aed6ca4db1dfad9869c2d6c5cfa57957fe00db3
Author: Your Name <you@example.com>
Date:   Fri May 22 11:42:00 2020 +0800

    append GPL

commit 290726ffe0b464dac70d9302e7be67a09b182dc9
Author: Your Name <you@example.com>
Date:   Fri May 22 11:36:29 2020 +0800

    add distributed

commit 4989438c34460fe6f42882518373d1960f4e11ff
Author: Your Name <you@example.com>
Date:   Fri May 22 10:36:24 2020 +0800

    wrote a readme file
hh@hh-X550JX:~/learngit$ git log --pretty=oneline
9aed6ca4db1dfad9869c2d6c5cfa57957fe00db3 append GPL
290726ffe0b464dac70d9302e7be67a09b182dc9 add distributed
4989438c34460fe6f42882518373d1960f4e11ff wrote a readme file
hh@hh-X550JX:~/learngit$
```

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 1094adb...（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^`

就可以改变 readme.txt 的内容到前一个版本

这时候输入 `git reset --hard 9aed6c...` 只要再本终端没有关闭的情况下，如果回退到上个版本，就可用此命令行再次来到最新的版本。9aed6c 是我最新版本号的前几位

```
hh@hh-X550JX:~/learngit$ git log --pretty=oneline
9aed6ca4db1dfad9869c2d6c5cfa57957fe00db3 append GPL
290726ffe0b464dac70d9302e7be67a09b182dc9 add distributed
4989438c34460fe6f42882518373d1960f4e11ff wrote a readme file
hh@hh-X550JX:~/learngit$ git reset --hard HEAD^
HEAD 现在位于 290726f add distributed
hh@hh-X550JX:~/learngit$ git reset --hard 9aed6c
HEAD 现在位于 9aed6ca append GPL
hh@hh-X550JX:~/learngit$
```

如果不小心关掉了终端，找不到版本号，用

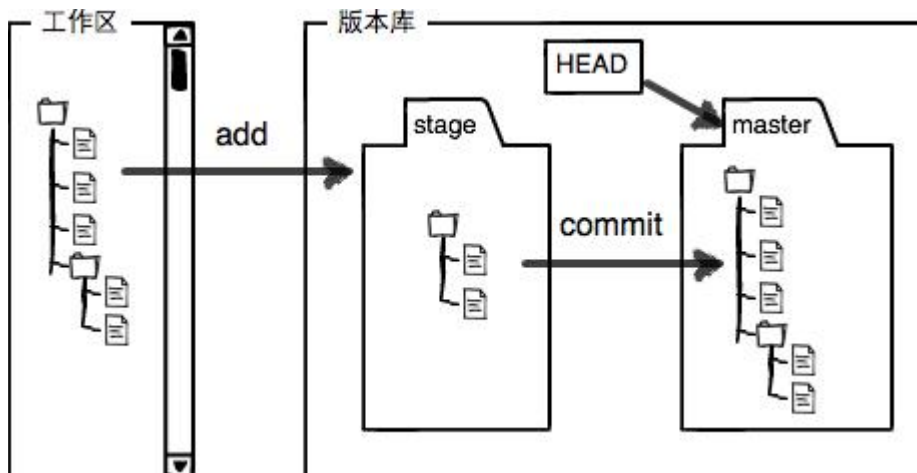
`git reflog`

就可以找到之前的版本号对应的操作

```
hh@hh-X550JX:~/learngit$ git reflog
HEAD 现在位于 9aed6ca append GPL
hh@hh-X550JX:~/learngit$ git reflog
9aed6ca HEAD@{0}: reset: moving to 9aed6c
290726f HEAD@{1}: reset: moving to HEAD^
9aed6ca HEAD@{2}: commit: append GPL
290726f HEAD@{3}: commit: add distributed
4989438 HEAD@{4}: commit (initial): wrote a readme file
hh@hh-X550JX:~/learngit$
```

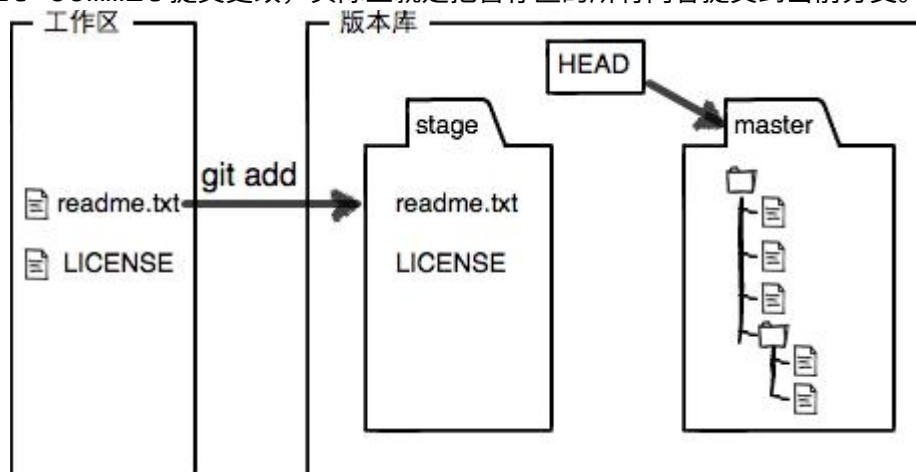
2. 工作区和暂存区

电脑上能看见的目录叫做工作区，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

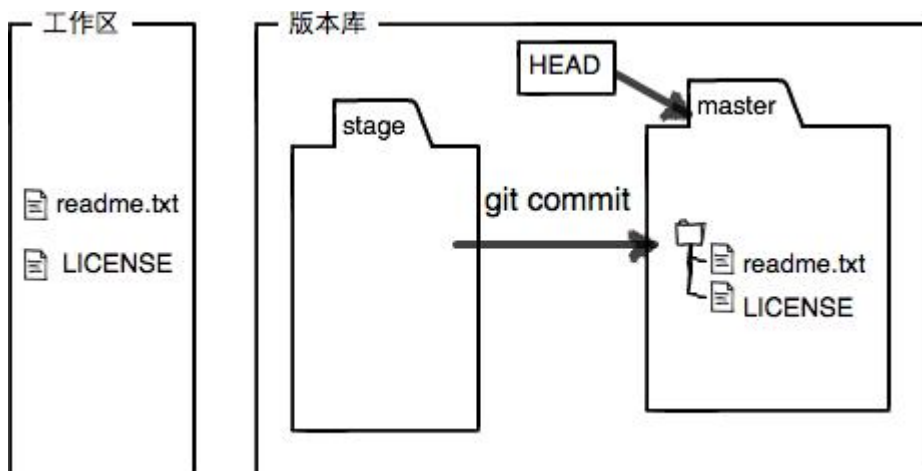


第一步是用 `git add` 把文件添加进去，实际上就是把文件修改添加到暂存区；

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。



commit 后



3. 管理修改

为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

例如，连续改一个文件两次：

第一次修改 -> git add -> 第二次修改 -> git commit

现在查看状态，只会提交第一次的修改，当你用 git add 命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit 只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

第一次修改 -> git add -> 第二次修改 -> git add -> git commit

这样才能提交两次

如果使用回退效果的命令的话，会回到第一次修改前，而不是第二次修改前，这是因为提交 commit 才算版本更新。

4. 撤销修改

当你修改了工作区的文件，使用 git status 会出现如下：

```
hh@hh-X550JX:~/learngit$ git status
位于分支 master
尚未暂存以备提交的变更：
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改：      readme.txt

修改尚未加入提交（使用 "git add" 和/或 "git commit -a"）
hh@hh-X550JX:~/learngit$
```

如果输入 git checkout - readme.txt，会将工作区的所有修改撤销，恢复到修改之前的状态

一种是 readme.txt 自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；一种是 readme.txt 已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。总之，就是让这个文件回到最近一次 git commit 或 git add 时的状态。

另外一种，修改了，添加到暂存区了，使用 git status 会出现如下：

```
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git status
位于分支 master
要提交的变更：
  (使用 "git reset HEAD <文件>..." 以取消暂存)

    修改：      readme.txt

hh@hh-X550JX:~/learngit$
```

Git 同样告诉我们，用命令 git reset HEAD readme.txt 可以把暂存区的修改撤销掉

(unstage)，重新放回工作区。不过这样只会把暂存区里面的文件退出来，并不会撤销修改，如需要撤销修改，就还需要执行 git checkout - readme.txt

5. 删除文件

直接在工作区删除，或者 rm readme.txt 命令，运行 git status 命令会立刻告诉你哪些文件被删除了。

一是确实要从版本库中删除该文件，那就用命令 git rm 删掉，并且 git commit。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本。

git checkout -- test.txt

git checkout 其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

三. 远程仓库

请自行注册 GitHub 账号。由于你的本地 Git 仓库和 GitHub 仓库之间的传输是通过 SSH 加密的，所以，需要一点设置：

ssh-keygen -t rsa -C "youremail@example.com"


```

hh@hh-X550JX: ~
hh@hh-X550JX:~$ ssh-keygen -t rsa -C "1342363452@qq.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hh/.ssh/id_rsa): /home/hh/.ssh/id_rsa
/home/hh/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hh/.ssh/id_rsa.
Your public key has been saved in /home/hh/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:RuUbXEPyyIit9yby4H6hrgh0v7bro8xpU2eM1MVNUh4 1342363452@qq.com
The key's randomart image is:
+----[RSA 2048]-----+
|      ..+Eo.+      |
|      =oB.=      |
|      . o +.* .      |
|      . . o  o      |
|      . o . S .      |
|      o + o..      |
|    ... o o...o      |
|  =+o+ ..+.o      |
|  .****+.      |
+----[SHA256]-----+
hh@hh-X550JX:~$ cd ~/.ssh

```

这时候再主目录下是看不到 .ssh 文件夹的, 也看不到 id_rsa 和 id_rsa.pub 文件的

```

hh@hh-X550JX:~$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQD+W0JU3eP+2SmejYDOXSWNkzXpo08NxC0yGd8ete5D3oO
cnSiVnApzwOuc4Wn/freZWQn787pk3BxTBX4aGjNbGXvLyU1yUJdl5+JnGwViPDhJRV68sU94A+72Mrpw6HPi
ORydsPNaQR6QQPLq1Iym5lUQIFA04D103KeNi5RMcmUKPIf/Amkzgis72uSR9ML6XGL/QlZyK9jmHfoYesLtd
xjaNYDooa2zcRzHXBo6z57qITC8Aa8rNk3KHOU1ypEhKGoPSNQVS3F3U6VfOgj/0Ns/vRw4Z54dD0q6nTapRuJ
ydUtyPufKCLMJEFjDPAWGsPyyZq4pzZYGoZrx4yl 1342363452@qq.com
hh@hh-X550JX:~$

```

但是能直接在终端显示, 然后拷贝到网页 github 里面, 创建新的 ssh 就好

WUHAN5827 Personal settings

- Profile
- Account
- Security
- Security log
- Emails
- Notifications
- Billing
- SSH and GPG keys**
- Blocked users
- Repositories
- Organizations
- Saved replies
- Applications
- Developer settings

SSH keys / Add new

Title

My ssh

Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQD+W0JU3eP+2SmejYDOXSWNkzXpo08NxC0yGd8ete5D3oOcnSi
VnApzwOuc4Wn/freZWQn787pk3BxTBX4aGjNbGXvLyU1yUJdl5+JnGwViPDhJRV68sU94A+72Mrpw6HPi
ORydsPNaQR6QQPLq1Iym5lUQIFA04D103KeNi5RMcmUKPIf/Amkzgis72uSR9ML6XGL
/QlZyK9jmHfoYesLtdxjaNYDooa2zcRzHXBo6z57qITC8Aa8rNk3KHOU1ypEhKGoPSNQVS3F3U6VfOgj
/0Ns/vRw4Z54dD0q6nTapRuJydUtyPufKCLMJEFjDPAWGsPyyZq4pzZYGoZrx4yl 1342363452@qq.com
```

Add SSH key

© 2020 GitHub, Inc. Terms Privacy Security Status Help

Contact GitHub Pricing API Training Blog About

这样的话，就能实现数据传输了，不过是公开的。如果你不想让别人看到 Git 库，有两个办法，一个是交点保护费，让 GitHub 把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个 Git 服务器，因为是你自己的 Git 服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

1. 添加远程库

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，在 Repository name 填入 `learngit`（想要创建远端仓库的名字），其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的 Git 仓库：然后终端运行：

```
git remote add origin git@github.com:WUHAN5827/learngit.git
```

这一步一般不会有任何的终端回馈，相当于关联一个远程库，名为 `origin`，`WUHAN5827` 是 `git` 的账户名，可改。然后

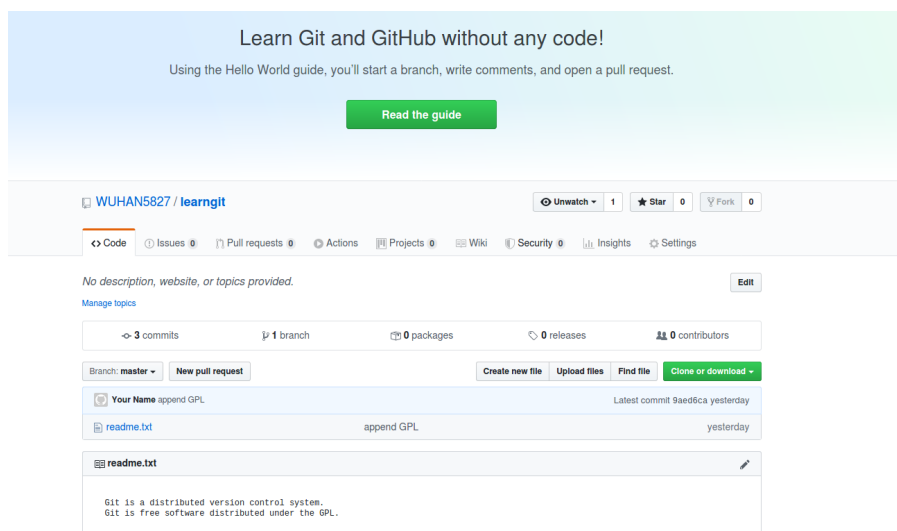
```
git push -u origin master
```

把本地库的内容推送到远程，用 `git push` 命令，实际上是把当前分支 `master` 推送到远程。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。之后的不用加 `-u`。

然后就可再 `github` 网页上找到现有的文件了。

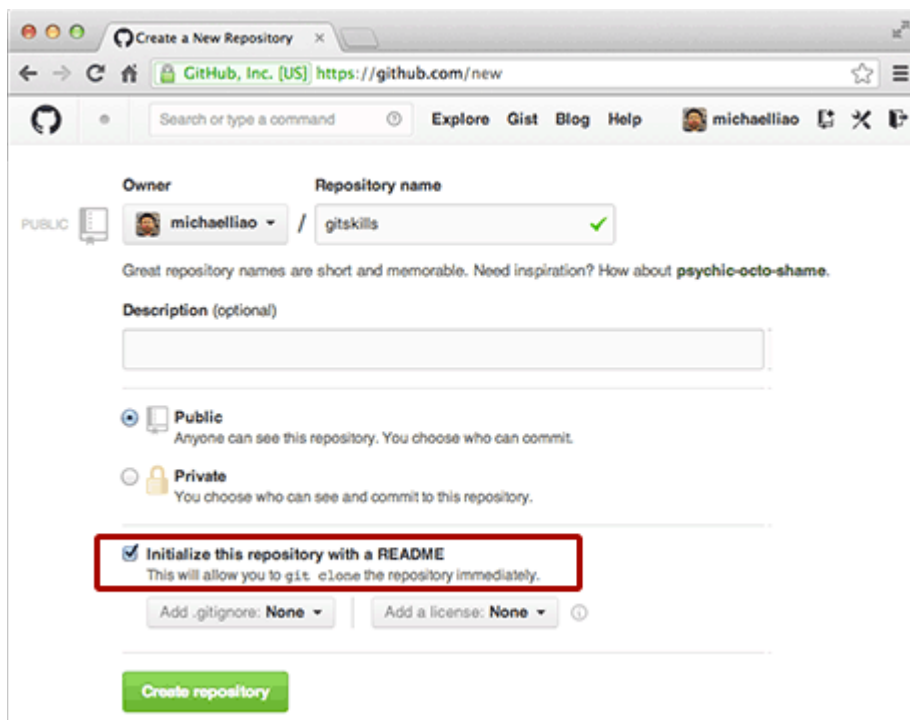


```
hh@hh-X550JX: ~/learngit
hh@hh-X550JX:~/learngit$ git remote add origin git@github.com:WUHAN5827/learngit.git
fatal: 远程 origin 已经存在。
hh@hh-X550JX:~/learngit$ git push -u origin master
The authenticity of host 'github.com (52.74.223.119)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? y
Please type 'yes' or 'no': yes
Warning: Permanently added 'github.com,52.74.223.119' (RSA) to the list of known hosts.
对象计数中: 9, 完成.
Delta compression using up to 8 threads.
压缩对象中: 100% (6/6), 完成.
写入对象中: 100% (9/9), 771 bytes | 0 bytes/s, 完成.
Total 9 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To git@github.com:WUHAN5827/learngit.git
 * [new branch]      master -> master
分支 master 设置为跟踪来自 origin 的远程分支 master。
hh@hh-X550JX:~/learngit$ git push origin master
Warning: Permanently added the RSA host key for IP address '13.229.188.59' to the list of known hosts.
Everything up-to-date
hh@hh-X550JX:~/learngit$
```



2. 克隆远程库

与一般的工作思路是现在远程创建库，然后再下载到本地，克隆远程库：先创建名为 gitskills 的库，

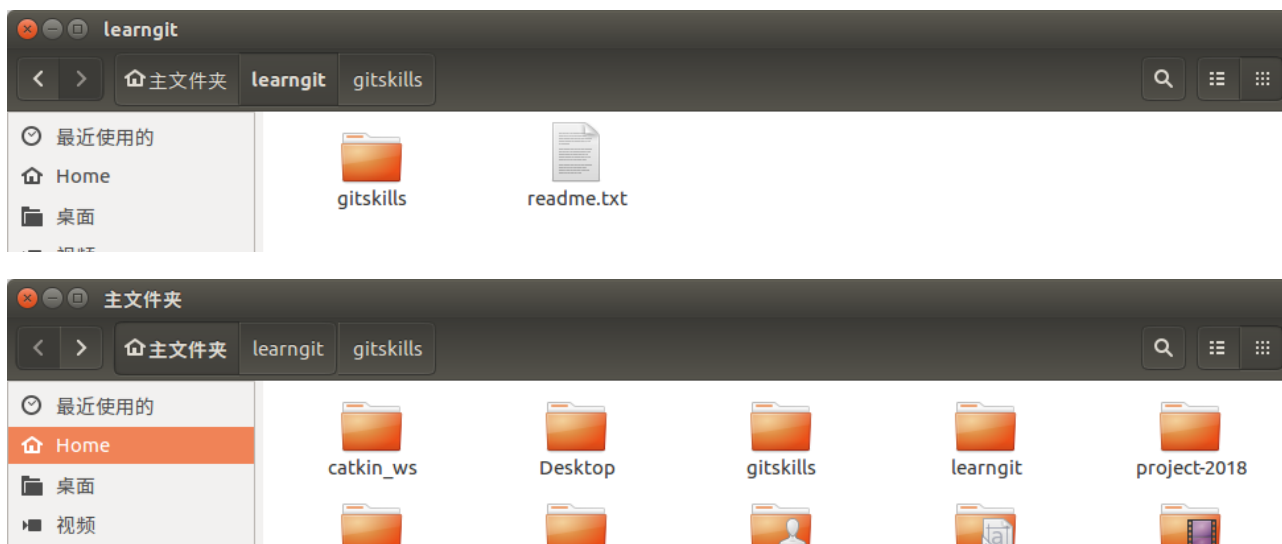


我们勾选 Initialize this repository with a README，这样 GitHub 会自动为我们创建一个 README.md 文件。创建完毕后，可以看到 README.md 文件

在主目录下，运行：

```
git clone git@github.com:WUHAN5827/gitskills.git
```

这样就会直接下载下来，注意终端的运行目录，在主目录下运行就会下载 gitskills 的文件夹，在 learngit 文件夹下运行，就会在 learngit 文件夹下下载此文件夹，结果如下：

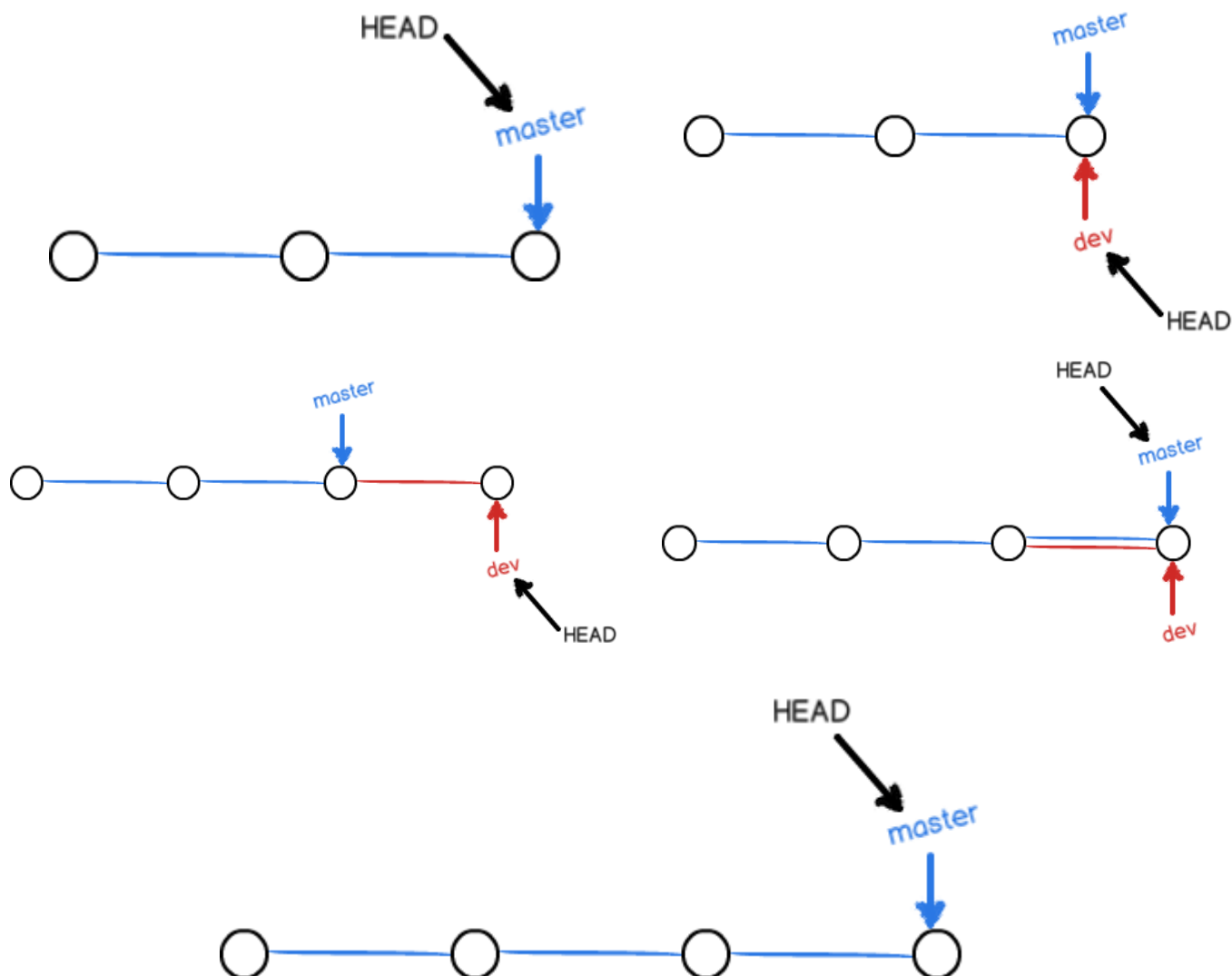


四. 分支管理

1. 创建和合并分支

现已存在主分支，master 分支，它指向最新的提交。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。现在创建新的分支 dev，再把 HEAD 指向 dev，就表示当前分支再 dev 上。然后在分支 dev 上面提交，dev 指针就会往前一步，而 master 指针还是指向老地方。然后合并两个指针，然后删除 dev 指针。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。



在工作目录下打开终端，不然没用

```
hh@hh-X550JX: ~/learngit
hh@hh-X550JX:~/learngit$ git checkout -b dev
切换到一个新分支 'dev'
hh@hh-X550JX:~/learngit$ git branch
* dev
  master
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git commit -m "branch test"
[dev be8b494] branch test
1 file changed, 1 insertion(+)
```

创建 dev 分支后先修改 readme.txt 文件，比如添加一行，然后在 dev 指针上进行一次提交，这时候查看工作区的 readme.txt 文件，是修改后的文件。如果这时候运行：

git checkout master

这样的话，切换回 master 分支，再查看 readme.TXT 文件，会发现它变成了修改之前的文件，没有修改痕迹。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

注意，由于上一步我们一已经把分支指向 master 了，所以 master 是当前分支，这一命令会把 dev 分支的工作合并到 master 上。合并之后的 readme.txt 文件为修改后的文件。

```
hh@hh-X550JX:~/learngit$ git checkout master
切换到分支 'master'
您的分支与上游分支 'origin/master' 一致。
hh@hh-X550JX:~/learngit$ git merge dev
更新 9aed6ca..be8b494
Fast-forward
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

合并模式为 Fast forward，然后就可以删除 dev 分支了：
git branch -d dev

```
hh@hh-X550JX:~/learngit$ git branch -d dev
已删除分支 dev (曾为 be8b494)。
hh@hh-X550JX:~/learngit$ git branch
* master
hh@hh-X550JX:~/learngit$
```

实际上，切换分支这个动作，用 switch 更科学。因此，最新版本的 Git 提供了新的 git switch 命令来切换分支：不过我现在电脑上的 git 版本不支持这个命令，哈哈哈

```
git switch -c dev
git switch master
```

2. 解决冲突

偶尔合并分支的时候会出现问题，这时候创建新分支，feature1。

```
git switch -c feature1
```

把 readme.txt 文件最后一行改为

Creating a new branch is quick AND simple.

然后提交，然后切换回 master，把最后一行改为：

Creating a new branch is quick & simple.

然后提交，这样，就有了两次不同的提交，

```
hh@hh-X550JX:~/learngit
hh@hh-X550JX:~/learngit$ git switch -c feature1
git: 'switch' 不是一个 git 命令。参见 'git --help'。
hh@hh-X550JX:~/learngit$ git checkout -b feature1
切换到一个新分支 'feature1'
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git commit -m "AND simple"
[feature1 265f00e] AND simple
 1 file changed, 1 insertion(+), 1 deletion(-)
hh@hh-X550JX:~/learngit$ git checkout master
切换到分支 'master'
您的分支领先 'origin/master' 共 1 个提交。
(使用 "git push" 来发布您的本地提交)
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git commit -m "& simple"
[master 9241cf5] & simple
 1 file changed, 1 insertion(+), 1 deletion(-)
hh@hh-X550JX:~/learngit$
```

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
```

```
hh@hh-X550JX: ~/learngit
1 file changed, 1 insertion(+), 1 deletion(-)
hh@hh-X550JX:~/learngit$ git merge feature1
自动合并 readme.txt
冲突 (内容): 合并冲突于 readme.txt
自动合并失败, 修正冲突然后提交修正的结果。
hh@hh-X550JX:~/learngit$ git status
位于分支 master
您的分支领先 'origin/master' 共 2 个提交。
(使用 "git push" 来发布您的本地提交)
您有尚未合并的路径。
(解决冲突并运行 "git commit")

未合并的路径:
(使用 "git add <文件>..." 标记解决方案)

    双方修改:    readme.txt

未跟踪的文件:
(使用 "git add <文件>..." 以包含要提交的内容)

    gitskills/

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
hh@hh-X550JX:~/learngit$
```

而 readme.txt 文件变成了下面这个样子:

```
readme.txt (~/learngit) - gedit
打开(O) 保存(S)

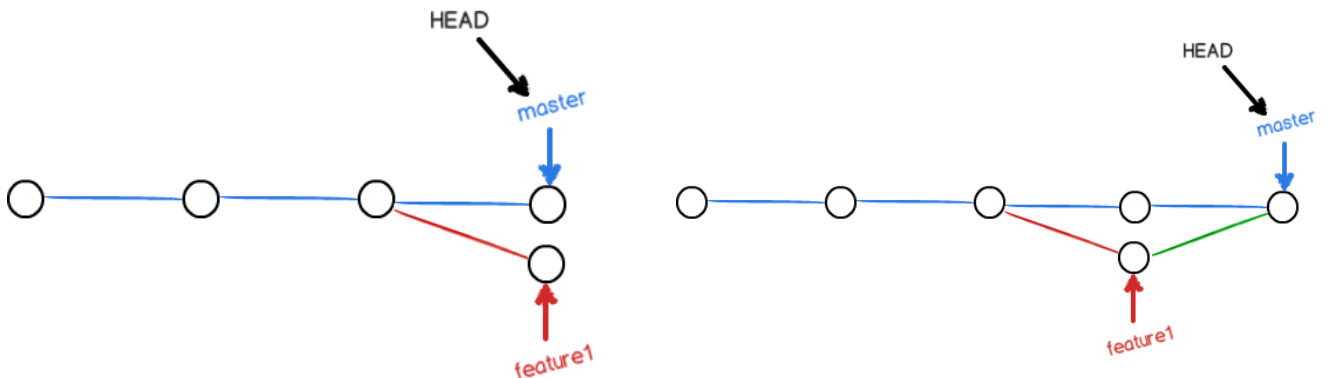
Git is a distributed version control system.
Git is free software distributed under the GPL.
<<<<<< HEAD
Creating a new branch is quick & simple.
=====
Creating a new branch is quick AND simple.
>>>>>> feature1|
```

这时候把 readme.txt 文件最后一行改为
Creating a new branch is quick and simple.

然后提交, 就可以解决问题。

```
hh@hh-X550JX:~/learngit$ git branch
feature1
* master
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git commit -m "conflict fixed"
[master 73fe7a5] conflict fixed
hh@hh-X550JX:~/learngit$
```

下图为分支图过程, 最后删除分支 feature1。

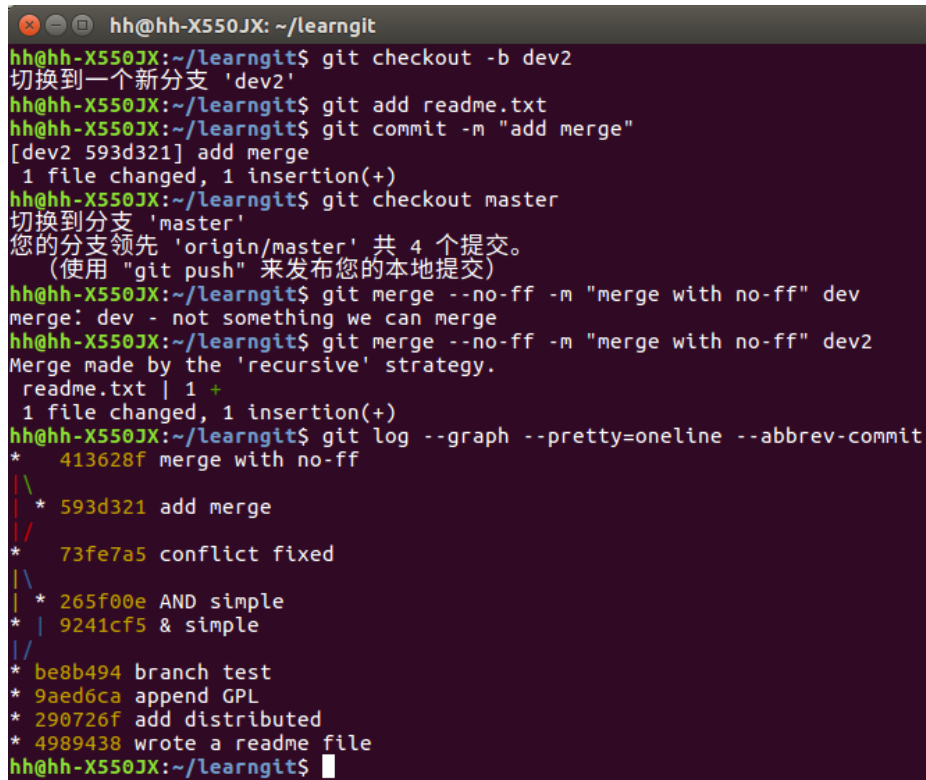


git log --graph --pretty=oneline --abbrev-commit
查看分支历史

3. 分支管理策略

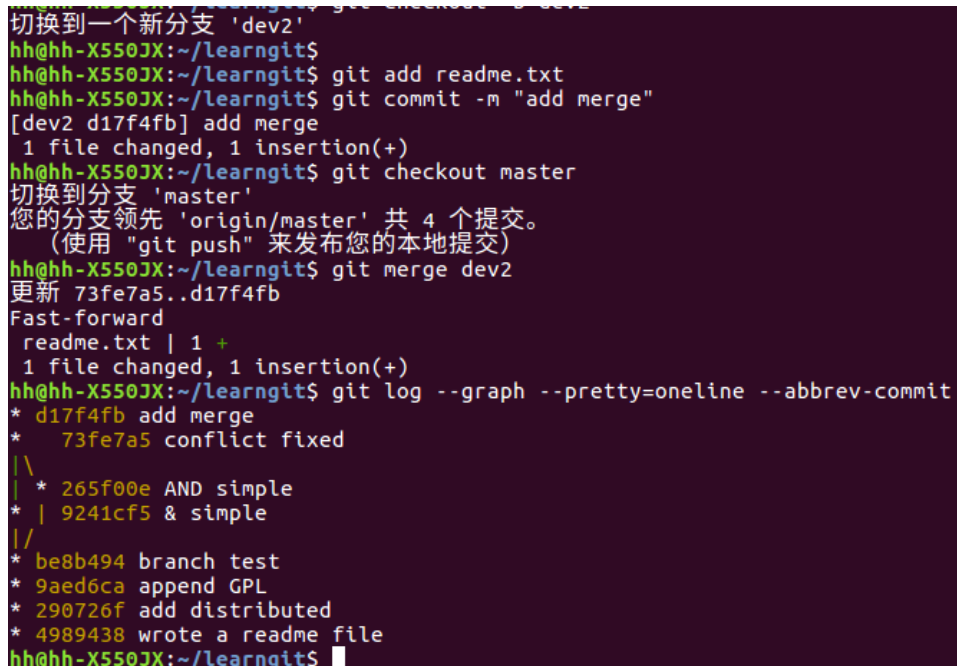
如果用 Fast forward 模式来合并分支，则会在分支历史上丢掉删除新分支的信息。如果要强制禁用 Fast forward 模式，Git 就会在 merge 时生成一个新的 commit，这样，从分支历史上就可以看出分支信息。先创建，然后修改并提交新的 commit：

```
git checkout -b dev
git add readme.txt
git commit -m "add merge"
git checkout master
git merge --no-ff -m "merge with no-ff" dev2
git log --graph --pretty=oneline --abbrev-commit
```



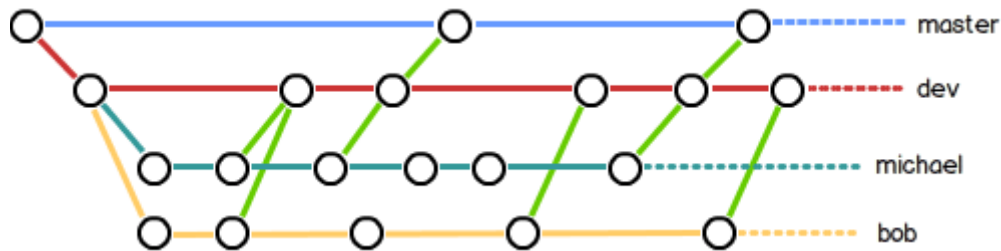
```
hh@hh-X550JX: ~/learngit
hh@hh-X550JX:~/learngit$ git checkout -b dev2
切换到一个新分支 'dev2'
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git commit -m "add merge"
[dev2 593d321] add merge
1 file changed, 1 insertion(+)
hh@hh-X550JX:~/learngit$ git checkout master
切换到分支 'master'
您的分支领先 'origin/master' 共 4 个提交。
(使用 "git push" 来发布您的本地提交)
hh@hh-X550JX:~/learngit$ git merge --no-ff -m "merge with no-ff" dev
merge: dev - not something we can merge
hh@hh-X550JX:~/learngit$ git merge --no-ff -m "merge with no-ff" dev2
Merge made by the 'recursive' strategy.
readme.txt | 1 +
1 file changed, 1 insertion(+)
hh@hh-X550JX:~/learngit$ git log --graph --pretty=oneline --abbrev-commit
* 413628f merge with no-ff
| \
| * 593d321 add merge
| /
* 73fe7a5 conflict fixed
| \
| * 265f00e AND simple
* | 9241cf5 & simple
| /
* be8b494 branch test
* 9aed6ca append GPL
* 290726f add distributed
* 4989438 wrote a readme file
hh@hh-X550JX:~/learngit$
```

下图是把版本退到上一个版本再来一次，采用 Fast forward 模式来做



```
hh@hh-X550JX:~/learngit$ git checkout -b dev2
切换到一个新分支 'dev2'
hh@hh-X550JX:~/learngit$ git add readme.txt
hh@hh-X550JX:~/learngit$ git commit -m "add merge"
[dev2 d17f4fb] add merge
1 file changed, 1 insertion(+)
hh@hh-X550JX:~/learngit$ git checkout master
切换到分支 'master'
您的分支领先 'origin/master' 共 4 个提交。
(使用 "git push" 来发布您的本地提交)
hh@hh-X550JX:~/learngit$ git merge dev2
更新 73fe7a5..d17f4fb
Fast-forward
readme.txt | 1 +
1 file changed, 1 insertion(+)
hh@hh-X550JX:~/learngit$ git log --graph --pretty=oneline --abbrev-commit
* d17f4fb add merge
* 73fe7a5 conflict fixed
| \
| * 265f00e AND simple
* | 9241cf5 & simple
| /
* be8b494 branch test
* 9aed6ca append GPL
* 290726f add distributed
* 4989438 wrote a readme file
hh@hh-X550JX:~/learngit$
```

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。



4. Bug 分支

如果此时正在进行一个项目的开发，但是突然进来了一个另外的项目需要改 bug，这时候先把开发的项目储藏起来，然后确定要在哪个分支上修复 bug，假定需要在 master 分支上修复，就从 master 创建临时分支，修改完后，切换提交合并，最后删除新建的分支，然后转到原来项目的分支上继续进行开发就好。

```
git stash —储藏
git checkout master
git checkout -b issue-101
git add readme.txt
git commit -m "fix bug 101"
git checkout master
git merge --no-ff -m "merged bug fix 101" issue-101 —合并数据
git checkout dev
git stash pop —恢复，之后就可以继续开发了
```

在 master 分支上修复了 bug 后，我们要想一想，dev 分支是早期从 master 分支分出来的，所以，这个 bug 其实在当前 dev 分支上也存在。同样的 bug，要在 dev 上修复，我们只需要把在 master 分支上新建的改 bug 分支 commit 的版本号提交所做的修改“复制”到 dev 分支。注意：我们只想复制这个提交所做的修改，并不是把整个 master 分支复制过来。用：

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

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
(use "git push" to publish your local commits)

$ git checkout -b issue-101
Switched to a new branch 'issue-101'
```

现在修复bug，需要把“Git is free software ...”改为“Git is a free software ...”，然后提交：

```
$ git add readme.txt
$ git commit -m "fix bug 101"
[issue-101 4c805e2] fix bug 101
1 file changed, 1 insertion(+), 1 deletion(-)
```

上图为改 bug 创建的新分支，和提交的版本号， cherry-pick 复制的就是这个版本号。

5. Feature 分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。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 feature 分支，在上面开发，完成后，合并，最后，删除该 feature 分支。

```
git checkout -b feature-vulcan
```

开发过程.....

```
git add vulcan.py
git commit -m "add feature vulcan"
git switch dev    -因为新功能是在 dev 分支上进行的，所以回到 dev 分支进行合并
```

如果这时候，老板说资金不足，项目取消，这时候就要删除 feature-vulcan 分支，因为是准备合并的但是还没有合并，如果删除，就会造成数据损失，所以终端回进行提醒如下：

```
$ 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'.
```

销毁失败。Git友情提醒，feature-vulcan 分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用大写的 -D 参数。。

现在我们强行删除：

```
$ git branch -D feature-vulcan
Deleted branch feature-vulcan (was 287773e).
```

终于删除成功！

6. 多人协作

查看远程库信息 `git remote -v`

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 master
git push origin dev
```

master 分支是主分支，因此要时刻与远程同步；dev 分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；bug 分支只用于在本地修复 bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个 bug；feature 分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

五. 标签管理

标签是一个让人容易记住的名字，等价于 commit 号。但是标签一旦指定就不可动，而分支是可动的。

默认标签是打在最新提交的 commit 上的，直接 checkout 到所需的分支，然后：

```
git tag <tagname>
```

git tag 可查看所有现有的标签

如果忘记给最新的 commit 提交打上标签，可以查看 commit 历史，找到对应的 id：

```
git tag <tagname> <commit id>
```

用 `git show <tagname>` 查看具体信息，也可用 `git tag -d <tagname>` 删除本地标签，如果要想把标签推送到远程，`git push origin <tagname>` 或者 `git push origin -tags`，如果已经推入远程，要想删除就麻烦一点，先删除本地的标签，然后 `git push origin :refs/tags/<tagname>`