

# Git简介

## 概述

Git是一个分布式的版本控制系统，速度快，体积小。

## 集中式与分布式

集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟。

分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

## Git安装

使用如下命令即可

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

安装完成之后，进行设置即可使用git

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

备注：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。

## Git 版本库创建

经过安装配置后，在Linux系统就可以使用Git了。接下来就是创建一个版本库。

1、首先创建一个目录，用于作为工作目录

```
kiko@hkiko:~$ mkdir gitrepository
kiko@hkiko:~$ cd gitrepository/
kiko@hkiko:~/gitrepository$
```

2、然后就是初始化版本库

通过git init命令，会把gitrepository这个目录变成一个git的仓库。

```
kiko@hkiko:~/gitrepository$ git init
已初始化空的 Git 仓库于 /home/kiko/gitrepository/.git/
```

通过查看当前目录中的所有文件，发现增加了一个.git，这个就是版本库。

```
kiko@Hkiko:~/gitrepository$ ls -al
总用量 12
drwxrwxr-x  3 kiko kiko 4096 7月  19 07:18 .
drwxr-xr-x 30 kiko kiko 4096 7月  19 07:14 ..
drwxrwxr-x  7 kiko kiko 4096 7月  19 07:18 .git
```

3、经过上面的步骤，已经有了一个可以工作的目录。接下来在/gitrepository中添加文件并且提交。

```
kiko@Hkiko:~/gitrepository$ vim readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a version control system.
Git is free software.
kiko@Hkiko:~/gitrepository$
kiko@Hkiko:~/gitrepository$ git add readme.txt
kiko@Hkiko:~/gitrepository$ git commit -m "add a file"
[master (根提交) 0d3265d] add a file
 1 file changed, 2 insertions(+)
 create mode 100644 readme.txt
kiko@Hkiko:~/gitrepository$
```

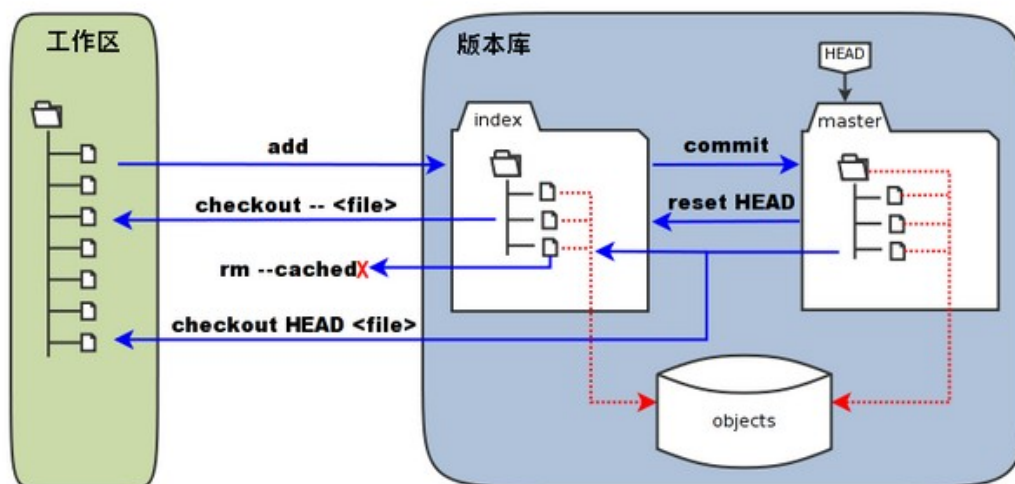
备注：commit命令的 -m 参数表示对本次提交进行说明，后面就是具体内容。  
其次，git add 一次也可以添加多个文件，在add后面写上文件名即可，空格空开。

## 暂存区、工作区和版本库

工作区：就是你在电脑里能看到的目录。如上面的gitrepository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，所以我们把暂存区有时也叫作索引（index）。

版本库：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。



CSDN @C学习者n号

## 文件的四种状态

版本控制就是对文件的版本控制，要对文件进行修改、提交等操作，首先要知道文件当前在什么状态，不然可能会提交了现在还不想提交的文件，或者要提交的文件没提交上。

Untracked: 未跟踪，此文件在文件夹中，但并没有加入到git库，不参与版本控制。通过git add 状态变为 Staged。

Unmodify: 文件已经入库, 未修改, 即版本库中的文件快照内容与文件夹中完全一致. 这种类型的文件有两种去处, 如果它被修改, 而变为Modified. 如果使用git rm移出版本库, 则成为Untracked文件

Modified: 文件已修改, 仅仅是修改, 并没有进行其他的操作. 这个文件也有两个去处, 通过git add可进入暂存staged状态, 使用git checkout 则丢弃修改过, 返回到unmodify状态, 这个git checkout即从库中取出文件, 覆盖当前修改!

Staged: 暂存状态. 执行git commit则将修改同步到库中, 这时库中的文件和本地文件又变为一致, 文件为Unmodify状态. 执行git reset HEAD filename取消暂存, 文件状态为Modified

## 版本追溯

### 查看仓库状态的两个常用命令

#### 1、git status

首先, 对readme.txt文件进行修改, 然后不对其进行git add和commit的操作。

```
kiko@hkiko:~/gitrepository$ vim readme.txt
kiko@hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
kiko@hkiko:~/gitrepository$ 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 restore <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")
```

备注: git status 查看仓库当前的状态, 显示有变更的文件。并且说明了修改的文件, 以及改文件所在分支和改文件还没有上传到暂存区以备提交。

#### 2、git diff

```
kiko@hkiko:~/gitrepository$ 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.
```

备注: git diff 比较文件的不同, 即暂存区和工作区的差异

3、下面, 我们对修改的文件提交, 在git add和gitcommit之后都是用上面两个命令查看状态。

```
kiko@hkiko:~/gitrepository$ git add readme.txt
kiko@hkiko:~/gitrepository$ git status
位于分支 master
要提交的变更:
  (使用 "git restore --staged <文件>..." 以取消暂存)
修改:      readme.txt

kiko@hkiko:~/gitrepository$ git diff
```

```
kiko@Hkiko:~/gitrepository$ git commit readme.txt -m "add a word distributed"
[master f966818] add a word distributed
1 file changed, 1 insertion(+), 1 deletion(-)
kiko@Hkiko:~/gitrepository$ git status
位于分支 master
无文件要提交，干净的工作区
kiko@Hkiko:~/gitrepository$ git diff
```

## 版本回退

这里的版本回退会作用于仓库中的所有文件，这里只有一个文件，所以只回退了一个。

1、先对文件进行一次修改并且提交到版本库中。

```
kiko@Hkiko:~/gitrepository$ vim readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
add a new line
kiko@Hkiko:~/gitrepository$ git add readme.txt
kiko@Hkiko:~/gitrepository$ git commit -m "add a new line"
[master 3ac9b1a] add a new line
1 file changed, 1 insertion(+)
```

2、git log

git log命令显示从最近到最远的提交日志。

```
kiko@Hkiko:~/gitrepository$ git log
commit 3ac9b1acadb92c805b99ee5c9d6444087faf06b40 (HEAD -> master)
Author: vv <553354863@qq.com>
Date: Tue Jul 19 07:45:27 2022 +0800

    add a new line

commit f966818cbab15bf7d81904b39f133806eda78587
Author: vv <553354863@qq.com>
Date: Tue Jul 19 07:38:47 2022 +0800

    add a word distributed

commit 0d3265da15b5231a92f0b30829c0a2b1c682f38e
Author: vv <553354863@qq.com>
Date: Tue Jul 19 07:25:09 2022 +0800

    add a file
```

2、git log --pretty=oneline

显示简洁版本的日志信息

```
kiko@Hkiko:~/gitrepository$ git log --pretty=oneline
3ac9b1acadb92c805b99ee5c9d6444087faf06b40 (HEAD -> master) add a new line
f966818cbab15bf7d81904b39f133806eda78587 add a word distributed
0d3265da15b5231a92f0b30829c0a2b1c682f38e add a file
```

### 3、开始回退版本 git reset

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3ac9...（版本号），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。

现在回退到上一个版本

```
kiko@Hkiko:~/gitrepository$ git reset --hard HEAD^
HEAD 现在位于 f966818 add a word distributed
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
```

### 查看日志情况

```
kiko@Hkiko:~/gitrepository$ git log
commit f966818cbab15bf7d81904b39f133806eda78587 (HEAD -> master)
Author: vv <553354863@qq.com>
Date:   Tue Jul 19 07:38:47 2022 +0800

    add a word distributed

commit 0d3265da15b5231a92f0b30829c0a2b1c682f38e
Author: vv <553354863@qq.com>
Date:   Tue Jul 19 07:25:09 2022 +0800

    add a file
```

可以看到，之前的提交在日志中没有了，也就是说当你回到过去，你将来的东西还没有发生。

恢复回退之前的版本。

如果还在当前的shell会话中，那么可以通过翻找上面的命令，找到版本号。

```
kiko@Hkiko:~/gitrepository$ git reset --hard 3ac9b1
HEAD 现在位于 3ac9b1a add a new line
```

如果已经没有了上次的shell，那么可以通过命令来查找这个版本号。即 git reflog 找到版本号之后，就可以使用上面的方式回退（过去/未来）某一个版本。  
git reflog命令记录了每一次操作的情况。

```
kiko@Hkiko:~/gitrepository$ git reflog
3ac9b1a (HEAD -> master) HEAD@{0}: reset: moving to 3ac9b1
f966818 HEAD@{1}: reset: moving to HEAD^
3ac9b1a (HEAD -> master) HEAD@{2}: commit: add a new line
f966818 HEAD@{3}: commit: add a word distributed
0d3265d HEAD@{4}: commit (initial): add a file
```

## 撤销修改

首先对文件回退到第二个版本

```
kiko@Hkiko:~/gitrepository$ git reset --hard f9668
HEAD 现在位于 f966818 add a word distributed
```

然后对文件内容进行了修改

```
kiko@Hkiko:~/gitrepository$ vim readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
manage change
kiko@Hkiko:~/gitrepository$ 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 restore <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")
```

通过git status的提示可以看到 `git restore <文件>` 可以丢弃工作区的内容。

命令功能: 丢弃工作区的修改 (不包括对文件自身的操作, 如添加文件、删除文件)

`git restore --staged <file_name>` 将暂存区的修改重新放回工作区 (包括对文件自身的操作, 如添加文件、删除文件)

```
kiko@Hkiko:~/gitrepository$ git restore readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
```

git checkout命令

命令 `git checkout -- file` 意思就是, 把文件在工作区的修改全部撤销 (对文件中内容的操作, 无法对添加文件、删除文件起作用), 这里有两种情况:

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。

第一种情况:

```
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
kiko@Hkiko:~/gitrepository$ vim readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
append a line
kiko@Hkiko:~/gitrepository$ git checkout -- readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
kiko@Hkiko:~/gitrepository$
```

第二种情况

```
kiko@Hkiko:~/gitrepository$ vim readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
```

```

Git is free software.
appen a line
kiko@Hkiko:~/gitrepository$ git add readme.txt
kiko@Hkiko:~/gitrepository$ vim readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
appen a line
new line
kiko@Hkiko:~/gitrepository$ git checkout -- readme.txt
kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
appen a line

```

最后，暂存区的修改撤销

上面的文件修改中，将修改的文件提交到暂存区，现在可以将暂存区的修改也撤销掉。

`git reset HEAD <file_name>` 丢弃暂存区的修改，重新放回工作区，会将暂存区的内容和本地已提交的内容全部恢复到未暂存的状态，不影响原来本地文件(相当于撤销git add 操作，不影响上一次commit后对本地文件的修改)（包括对文件的操作，如添加文件、删除文件）

`git reset` 命令既可以回退版本，也可以把暂存区的修改回退到工作区。

```

kiko@Hkiko:~/gitrepository$ git status
位于分支 master
要提交的变更：
  （使用 "git restore --staged <文件>..." 以取消暂存）
    修改:      readme.txt

kiko@Hkiko:~/gitrepository$ cat readme.txt
Git is a distributed version control system.
Git is free software.
appen a line
kiko@Hkiko:~/gitrepository$ git reset HEAD readme.txt
重置后取消暂存的变更：
M       readme.txt

```

经过上面的回退，当前工作区的内容就是相当于没有git add操作的修改的内容。然后把工作区的内容也修改了那么久版本统一了。

```

kiko@Hkiko:~/gitrepository$ 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 restore <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"）
kiko@Hkiko:~/gitrepository$ git restore readme.txt
kiko@Hkiko:~/gitrepository$ git status
位于分支 master
无文件要提交，干净的工作区

```

## 文件删除

`git rm` 文件名 删除改文件，且提交到了暂存区，之后需要 `git commit` 生效。

```
kiko@Hkiko:~/gitrespository$ git rm v.txt
rm 'v.txt'
kiko@Hkiko:~/gitrespository$ git status
位于分支 master
要提交的变更:
  (使用 "git restore --staged <文件>..." 以取消暂存)
    删除:       v.txt
```

```
kiko@Hkiko:~/gitrespository$ git commit v.txt -m "delete v.txt"
[master 0776a53] delete v.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 v.txt
kiko@Hkiko:~/gitrespository$ git status
位于分支 master
无文件要提交，干净的工作区
```

如果执行了 `git rm v.txt`，还没有提交，此时想要撤销删除操作。

```
kiko@Hkiko:~/gitrespository$ git rm v.txt
rm 'v.txt'
kiko@Hkiko:~/gitrespository$ git status
位于分支 master
要提交的变更:
  (使用 "git restore --staged <文件>..." 以取消暂存)
    删除:       v.txt

kiko@Hkiko:~/gitrespository$ git restore --staged v.txt
kiko@Hkiko:~/gitrespository$ git status
位于分支 master
尚未暂存以备提交的变更:
  (使用 "git add/rm <文件>..." 更新要提交的内容)
  (使用 "git restore <文件>..." 丢弃工作区的改动)
    删除:       v.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
kiko@Hkiko:~/gitrespository$ git restore v.txt
kiko@Hkiko:~/gitrespository$ git status
位于分支 master
无文件要提交，干净的工作区
kiko@Hkiko:~/gitrespository$ ls -al
总用量 12
drwxrwxr-x  3 kiko kiko 4096 7月 19 12:33 .
drwxr-xr-x 30 kiko kiko 4096 7月 19 12:12 ..
drwxrwxr-x  8 kiko kiko 4096 7月 19 12:33 .git
-rw-rw-r--  1 kiko kiko   0 7月 19 12:33 v.txt
```



## 命令总结 (以上)

仓库初始化:

`git init` --> 将当前目录设置为git的工作目录, 可以被git所管理

设置用户和邮箱(全局, 在git中的所有操作都是该用户的操作)

`git config --global user.name "vv"`

`git config --global user.email "553354863@qq.com"`

添加和提交

`git add <directory> / <file>` # 将指定目录的所有修改加入到下一次commit中(暂存区)。把<directory>替换成<file>将添加指定文件的修改

`git commit -m "message"` #提交暂存区的修改, 使用指定的<message>作为提交信息

状态信息

`git status` # 显示哪些文件已被staged(暂存状态)、未被staged以及未跟踪(untracked)。

`git diff` # 比较工作区和暂存区的修改。

`git diff HEAD` # 比较工作区和上一次commit后的修改。

`git diff --cached` # 比较暂存区和上一次commit后的修改。

## 可以在命令后面添加 `-- readme.txt`表示只判断该文件

`git diff HEAD -- readme.txt` ## 比较工作区和版本库中的readme.txt文件

日志

`git log` # 显示详细的日志信息, 全部显示。

`git log -<limit>` # 显示log, 限制log的显示数量。例如: "git log -5"仅显示最新5条commit

`git log --oneline` # 每行显示一条commit。简洁版本

`git reflog` # 显示本地仓库的所有commit日志

版本回退 (针对所有修改的文件)

`git reset --hard` # 移除所有暂存区的修改, 并强制删除所有工作区的修改。此时工作区、暂存区和版本库中的版本会一致。(回退到最近的一次提交)

`git reset` # 移除所有暂存区的修改, 但不会修改工作区。此时的暂存区和版本库一致。(回退到最近的一次提交)

`git reset --hard <commit id>` # 移除所有暂存区的修改, 并强制删除所有工作区的修改。此时工作区、暂存区和版本库中的版本会一致。(回退到指定的一次提交)

`git reset <commit id>` # 移除所有暂存区的修改, 但不会修改工作区。此时的暂存区和版本库一致。(回退到指定的一次提交)

`git reset HEAD <file_name>` # 丢弃暂存区的修改, 重新放回工作区, 会将暂存区的内容和本地已提交的内容全部恢复到未暂存的状态, 不影响原来本地文件(相当于撤销git add 操作, 不影响上一次commit后对本地文件的修改。此时暂存区和版本库统一, 工作区还有修改后的内容) (包括对文件的操作, 如添加文件、删除文件)

`git reset --hard HEAD` # 清空暂存区, 将已提交的内容的版本恢复到本地, 本地的文件也将被恢复的版本替换(恢复到上一次commit后的状态, 上一次commit后的修改也丢弃, 此时三方版本统一)

撤销修改（文件内容 / 文件本身）--> 针对某一个文件

`git restore --staged <file_name>` # 将暂存区的修改重新放回工作区（包括对文件自身的操作，如添加文件、删除文件）

`git restore <file_name>` # 丢弃工作区的修改（不包括对文件自身的操作，如添加文件、删除文件）

`git checkout -- <file_name>` # 丢弃工作区的修改，并用最近一次的`git add` 或者 `git commit`状态时的状态，还原到当前工作区（对文件中内容的操作，无法对添加文件、删除文件起作用）

`git checkout HEAD^ -- <file_name>` # 将指定`commit`提交的内容(`HEAD^`表示上一个版本)还原到当前工作区

删除文件

`git rm file` # 删除指定文件，且提交到了暂存区。

## GitHub远程仓库

### 创建GitHub用户

### 本地生成ssh密钥

进入`~/.ssh`目录，输入横线处命令回车，下面也是一直回车即可。

```
kiko@kiko:~/.ssh$ ssh-keygen -t rsa -C "553354863@qq.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/kiko/.ssh/id_rsa): 
Enter passphrase (empty for no passphrase): 
Enter same passphrase again: 
Your identification has been saved in /home/kiko/.ssh/id_rsa
Your public key has been saved in /home/kiko/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:TVPGKum+Zr5gDNQfG7sqUX7dAucdxLH85wiMAZyjBFA 553354863@qq.com
The key's randomart image is:
+----[RSA 3072]-----+
| .oE. ... o+.
| .. +. ++.
| ....00+00
| . o.+Bo=..
| .o .S*000.. .
| .o. o.+ o. +
| .+o. . . .
| .. o+
| ..++o
+----[SHA256]-----+
```

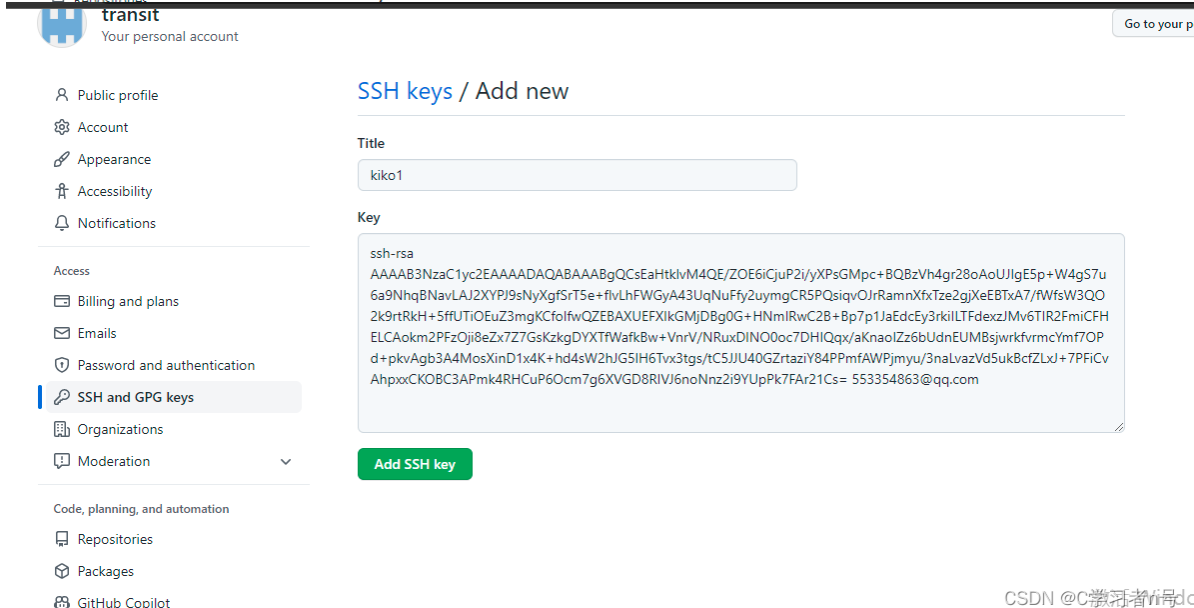
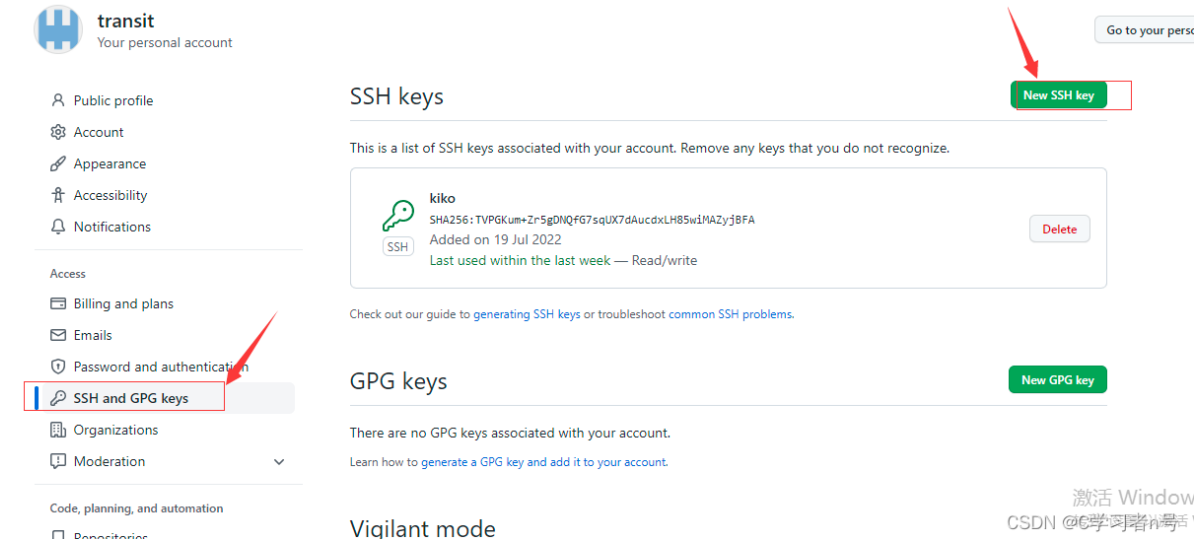
CSDN @C学习者n号

之后生成密钥文件`id_rsa`私钥，`id_rsa.pub`公钥。

```
kiko@kiko:~/ssh$ ls -al
总用量 20
drwx----- 2 kiko kiko 4096 7月 19 22:59 .
drwxr-xr-x 30 kiko kiko 4096 7月 19 22:54 ..
-rw----- 1 kiko kiko 2602 7月 19 22:59 id_rsa
-rw-r--r-- 1 kiko kiko 570 7月 19 22:59 id_rsa.pub
-rw-r--r-- 1 kiko kiko 444 7月 19 22:16 known_hosts
```

输出公钥内容，将其复制到GitHub网站中，如下：

```
kiko@kiko:~/ssh$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCsEaHtklvM4QE/ZOE6iCjuP2i/yXPsGMpc+BQBzVh4gr28oAoUJlE5p+W4gS7u6a9NhqBNNavLAJ2XYPJ9sNyXgfSrT5e+fLvLh
FWGyA43UqNuFfy2uymgCR5PQsiqvOJrRammXfxTze2gjXeEBTxA7/fWfswBQ02k9rtRkH+5ffUTiOEuZ3mgKCfoLfwQZEBAXUEFXIKGMjDBg0G+HNmIRwC2B+Bp7p1JaEdcEy3rk1
ILTFdexzJMv6TIR2Fm1CFHELCAokm2PFzOj18eZx7Z7GsKzkgDYXTfWafkBw+VnrV/NRuxDINO0oc7DHIQqx/aKnaoIZz6bUdnEUMBsjwrkfvrncYmf70Pd+pkvAgb3A4MosXinD1
x4K+hd4sW2hJG5IH6Tvx3tgs/tC5JJU40GZrtaziY84PPmFAWPjmyu/3naLvazVd5ukBcfZLxJ+7PFIcVAhpXXCKOBC3APmk4RHCU60cm7g6XVGD8RIV6noNnz2i9YUpPk7FAr21Cs= 553354863@qq.com
```



## GitHub创建仓库（上传本地仓库）

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

Repository name \*

java-sys

/


gitrepository


✓

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

Description (optional)

for study link remote repository

☒  **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:

Skip this step if you're importing an existing repository.

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)


**Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None

CSDN @C学习者n号

然后点击创建仓库，网页会跳转到这个仓库。

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

 Set up in Desktop

 or 

HTTPS

SSH

git@github.com:java-sys/gitrepository.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 "# gitrepository" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M 'main'
git remote add origin git@github.com:java-sys/gitrepository.git
git push -u origin 'main'
```

### ...or push an existing repository from the command line

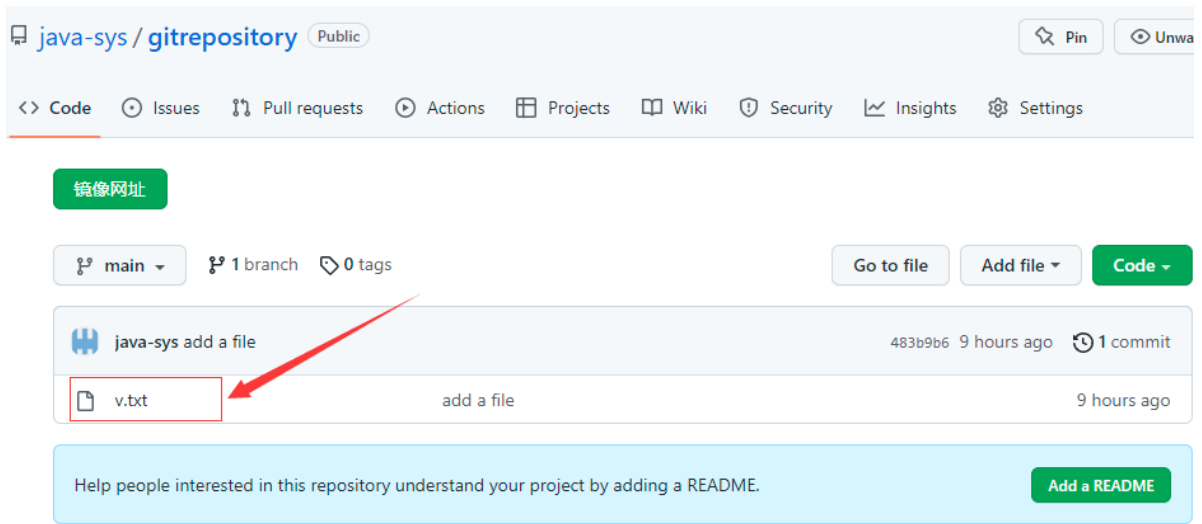
```
git remote add origin git@github.com:java-sys/gitrepository.git
git branch -M 'main'
git push -u origin 'main'
```

CSDN @C学习者n号

上面已经配置好了ssh秘钥，所以使用这种连接方式，将本地的库上传到这里。GitHub已经告诉了步骤命令。复制粘贴到shell中即可。

```
kiko@Hkiko:~/gitrespository$ git remote remove origin
kiko@Hkiko:~/gitrespository$ git remote add origin git@github.com:java-sys/gitrepository.git
kiko@Hkiko:~/gitrespository$ git branch -M 'main'
kiko@Hkiko:~/gitrespository$ git push -u origin 'main'
枚举对象中: 3, 完成.
对象计数中: 100% (3/3), 完成.
写入对象中: 100% (3/3), 201 字节 | 201.00 KiB/s, 完成.
总共 3 (差异 0), 复用 0 (差异 0)
To github.com:java-sys/gitrepository.git
 * [new branch]      main -> main
分支 'main' 设置为跟踪来自 'origin' 的远程分支 'main'。
kiko@Hkiko:~/gitrespository$
```

CSDN @C学习者n号



CSDN @C学习者n号

## GitHub创建仓库（本地克隆远程）

首先在GitHub上新建一个仓库，比如是一个c语言的项目仓库。

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

java-sys

Repository name \*

gitclone

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

Description (optional)

for clone a repository

☒ 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:

Skip this step if you're importing an existing repository.

☒ Add a README file

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

Add .gitignore

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

.gitignore template: C

Choose a license

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

License: GNU General Public ...

This will set `main` as the default branch. Change the default name in your [settings](#).

You are creating a public repository in your personal account.

Create repository

CSDN @C学习者n号

然后直接跳转进入仓库

java-sys / gitclone Public

[Pin](#) [Unwatch](#)

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

[镜像网址](#)

main

1 branch

0 tags

[Go to file](#) [Add file](#) [Code](#)

java-sys Initial commit 549431f 23 seconds ago 1 commit

.gitignore	Initial commit	23 seconds ago
LICENSE	Initial commit	23 seconds ago
README.md	Initial commit	23 seconds ago

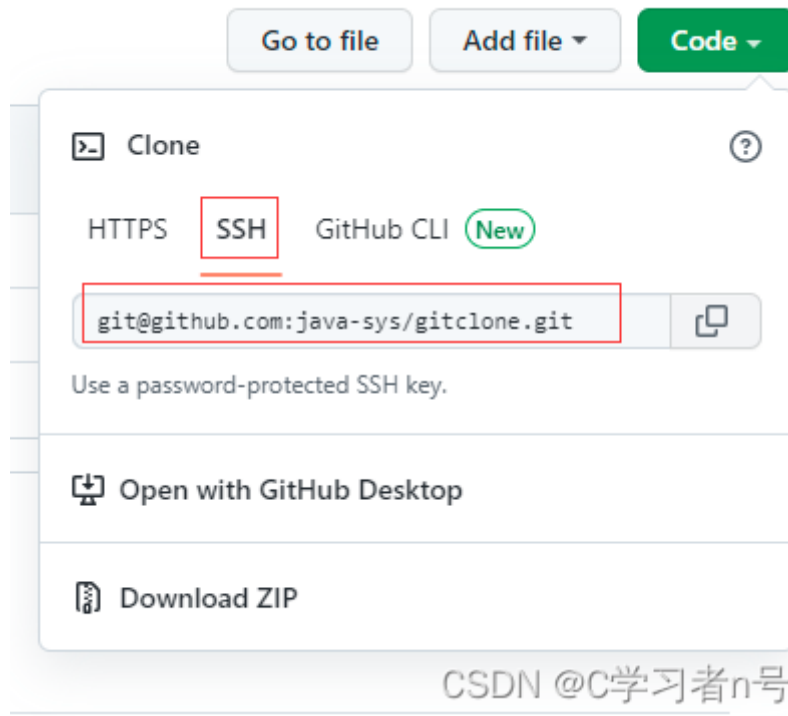
README.md

# gitclone

for clone a repository

CSDN @C学习者n号

直接复制图片中的地址。



现在在本地进行操作 (git clone)

首先创建一个目录，用于存放克隆过来的仓库。但是这个创建的目录并不是工作空间或者库，只是一个普通目录

```
kiko@Hkiko:~/gitclone$ git clone git@github.com:java-sys/gitclone.git
正克隆到 'gitclone'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
接收对象中: 100% (5/5), 12.80 KiB | 2.56 MiB/s, 完成.
kiko@Hkiko:~/gitclone$ ls -al
总用量 12
drwxrwxr-x  3 kiko kiko 4096 7月  20 08:12 .
drwxr-xr-x 31 kiko kiko 4096 7月  20 08:11 ..
drwxrwxr-x  3 kiko kiko 4096 7月  20 08:12 gitclone
kiko@Hkiko:~/gitclone$ cd gitclone
kiko@Hkiko:~/gitclone/gitclone$ ls -al
总用量 56
drwxrwxr-x  3 kiko kiko  4096 7月  20 08:12 .
drwxrwxr-x  3 kiko kiko  4096 7月  20 08:12 ..
drwxrwxr-x  8 kiko kiko  4096 7月  20 08:12 .git
-rw-rw-r--  1 kiko kiko   430 7月  20 08:12 .gitignore
-rw-rw-r--  1 kiko kiko 35149 7月  20 08:12 LICENSE
-rw-rw-r--  1 kiko kiko    34 7月  20 08:12 README.md
```

为了方便操作可以先起个别名。在本地仓库添加了一个pull.c文件。

```
kiko@Hkiko:~/gitclone/gitclone$ git remote add origin git@github.com:java-sys/gitclone.git
kiko@Hkiko:~/gitclone/gitclone$ git push -u origin 'main'
枚举对象中: 4, 完成.
对象计数中: 100% (4/4), 完成.
使用 4 个线程进行压缩
压缩对象中: 100% (2/2), 完成.
写入对象中: 100% (3/3), 258 字节 | 51.00 KiB/s, 完成.
总共 3 (差异 1), 复用 0 (差异 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:java-sys/gitclone.git
 549431f..a2b9cb4  main -> main
分支 'main' 设置为跟踪来自 'origin' 的远程分支 'main'.
```

## 分支别名

无论是上传本地仓库还是本地克隆仓库，主分支都改名为main，git默认是master。可以通过 `git branch` 命令查看分支名称。本地上传时，是根据GitHub的提示指定操作的，其中第二句就是将主分支改名为main；而克隆远程仓库，GitHub已经默认改好了。所以上面操作时使用的分支名都是main。

## 命令总结

### 1、git remote

远程仓库

```
git remote add [-t <分支>] [-m <master>] [-f] [--tags | --no-tags] [--mirror=  
<fetch|push>] <名称> <地址>
```

# 起别名。比如 `git remote add origin git@github.com:java-sys/gitrepository.git` 就是把一串地址起别名为origin,方便操作。

```
git remote [-v | --verbose]
```

# 查看远程主机地址，别名 + 地址（有-v参数）；查看远程主机地址别名（没有-v参数）

```
git remote rename <旧名称> <新名称>
```

# 修改别名。

```
git remote remove <名称>
```

# 移除指定远程主机别名。之后将无法通过这个别名推送数据。

# 但是还是可以直接用远程主机地址来操作。别名只是为了方便记忆地址，并不会影响连接远程主机的问题。

```
kiko@Hkiko:~/gitrepository$ git remote remove original
kiko@Hkiko:~/gitrepository$ git remote
kiko@Hkiko:~/gitrepository$ vim v.txt
kiko@Hkiko:~/gitrepository$ git add v.txt
kiko@Hkiko:~/gitrepository$ git commit 'v.txt' -m "change file"
[main fc881cc] change file
1 file changed, 13 insertions(+), 585 deletions(-)
kiko@Hkiko:~/gitrepository$ git push -u git@github.com:java-sys/gitrepository.git 'main'
枚举对象中: 5, 完成.
对象计数中: 100% (5/5), 完成.
使用 4 个线程进行压缩
压缩对象中: 100% (2/2), 完成.
写入对象中: 100% (3/3), 308 字节 | 77.00 KiB/s, 完成.
总共 3 (差异 1), 复用 0 (差异 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:java-sys/gitrepository.git
ef33e04..fc881cc main -> main
分支 'main' 设置为跟踪来自 'git@github.com:java-sys/gitrepository.git' 的远程分支 'main'.
```

### 2、git clone

克隆远程仓库

```
git clone <版本库的网址> <本地目录名>
```

# `git clone`支持多种协议，除了HTTP(s)以外，还支持SSH、Git、本地文件协议等。

## 分支管理

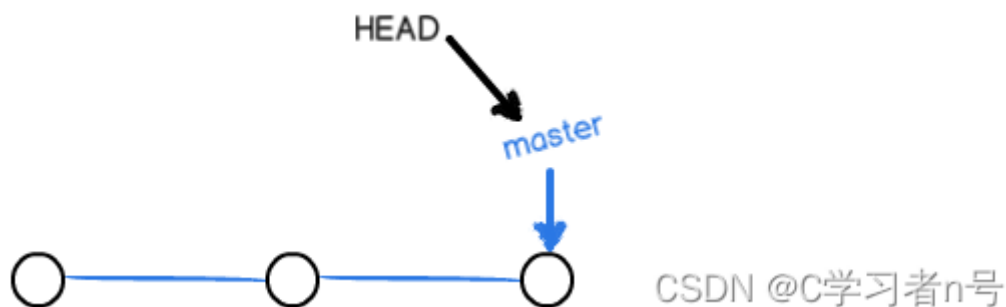


## 分支?

Git每次提交都会记录，串成一条时间线。一般情况下，都是在master主分支中上传提交数据，只有一条时间线。

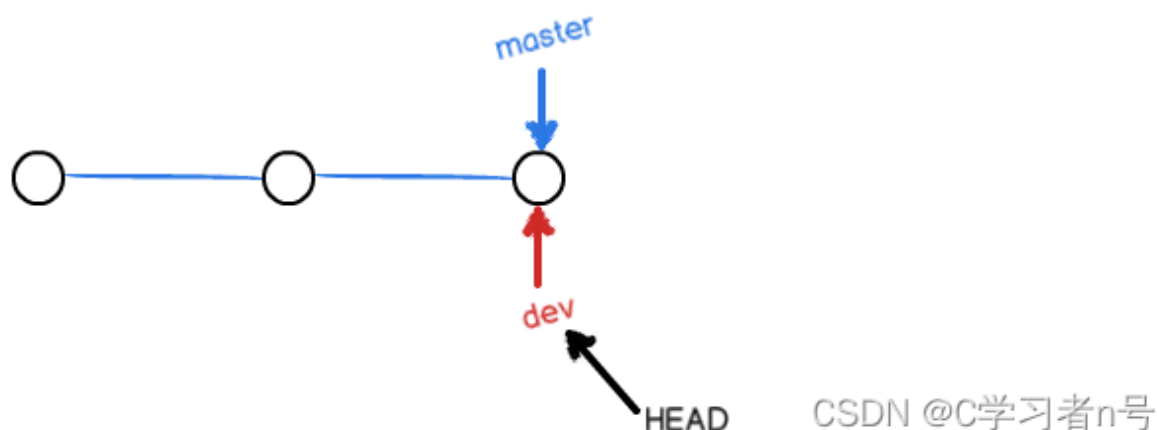
Git中的HEAD严格来说不是指向最新的提交的，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

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



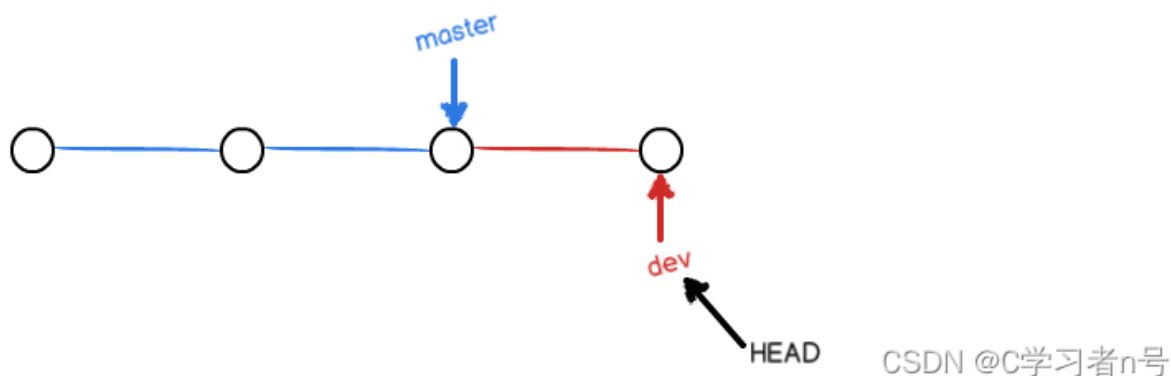
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

2、当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：

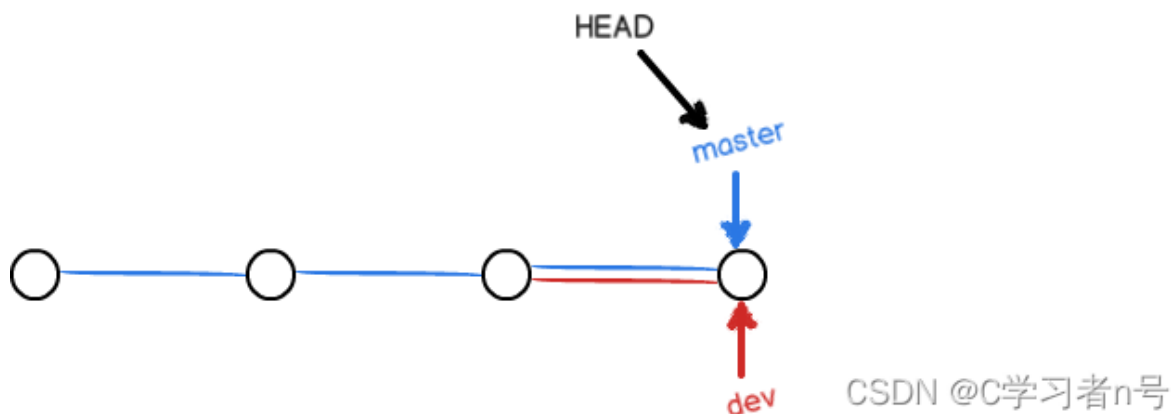


Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

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

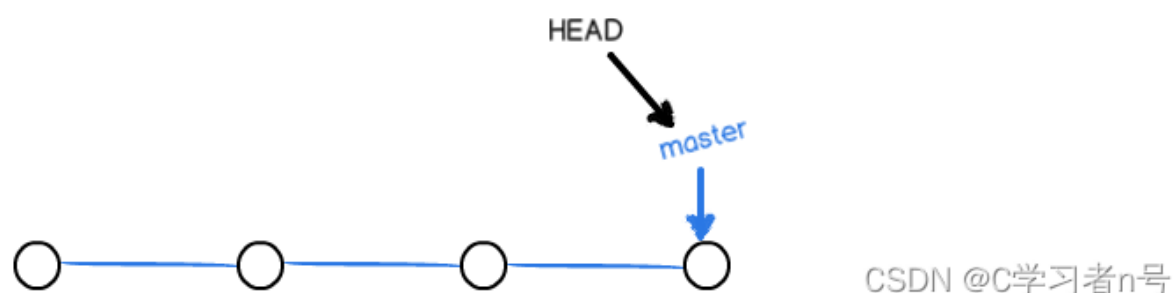


4、假如我们在dev上的工作完成了，就可以把dev合并到master上。Git合并最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

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



## 查看、创建、切换、合并、删除分支

### 1、创建 / 切换分支

首先创建一个文件夹gitbran，该文件夹作为工作空间，进入该文件夹初始化，然后创建一个文件，提交到版本库。此时还只有一个分支master。可以通过 `git branch` 查看。

之后，开始创建一个dev分支，`git branch dev` 负责创建这个分支，此时可以查分支，已经有了dev，然后通过命令 `git checkout dev` 切换当前分支到dev。现在再次查看分支情况，dev的前面就有了 \* 号。

说明：创建和切换分支分开进行和同时进行有两种方式：

```
git branch dev          #创建分支

git checkout -b dev     # 创建并且切换分支
git checkout dev        # 仅切换分支

# switch为新方式
git switch -c dev       # 创建并且切换分支
git switch dev          # 切换分支
```

```
kiko@Hkiko:~/gitclone/gitbran$ git branch
kiko@Hkiko:~/gitclone/gitbran$ touch v
kiko@Hkiko:~/gitclone/gitbran$ git add v
kiko@Hkiko:~/gitclone/gitbran$ git commit v -m "add a file"
[master (根提交) c233449] add a file
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 v

kiko@Hkiko:~/gitclone/gitbran$ git branch
* master
kiko@Hkiko:~/gitclone/gitbran$ git branch dev
kiko@Hkiko:~/gitclone/gitbran$ git branch
```

```
dev
* master
kiko@Hkiko:~/gitclone/gitbran$ git checkout dev
切换到分支 'dev'
```

## 2、合并、删除分支

在切换到了dev分支之后，在此添加了一个文件dev；然后切换到git switch master 主分支。此时可以将子分支dev合并到master分支，通过git merge dev即可将dev合并到master分支。如果dev子分支没有了其他作用，可以通过git branch -d dev删除它。

```
kiko@Hkiko:~/gitclone/gitbran$ touch dev
kiko@Hkiko:~/gitclone/gitbran$ git add dev
kiko@Hkiko:~/gitclone/gitbran$ git commit dev -m "add a file"
[dev 983b147] add a file
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dev

kiko@Hkiko:~/gitclone/gitbran$ git switch master
切换到分支 'master'
kiko@Hkiko:~/gitclone/gitbran$ git merge dev
更新 c233449..983b147
Fast-forward
 dev | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 dev

kiko@Hkiko:~/gitclone/gitbran$ git branch -d dev
已删除分支 dev（曾为 983b147）。
kiko@Hkiko:~/gitclone/gitbran$ git branch
* master
```

## 合并分支的冲突

1、首先在一个空仓库中创建一个文件readme.txt，然后写上依据内容：Creating a new branch is quick。然后git add readme.txt / git commit readme.txt -m "add a file" 提交到版本库中。

```
kiko@Hkiko:~/gitclone$ mkdir gitconfit
kiko@Hkiko:~/gitclone$ cd gitconfit/
kiko@Hkiko:~/gitclone/gitconfit$ git init
已初始化空的 Git 仓库于 /home/kiko/gitclone/gitconfit/.git/
kiko@Hkiko:~/gitclone/gitconfit$ vim readme.txt
kiko@Hkiko:~/gitclone/gitconfit$ git add readme.txt
kiko@Hkiko:~/gitclone/gitconfit$ git commit readme.txt -m "add a file"
[master (根提交) a2609c7] add a file
1 file changed, 1 insertion(+)
create mode 100644 readme.txt
```

2、创建一个分支feature，并且切换到到该分支：git switch -c feature

```
kiko@Hkiko:~/gitclone/gitconfit$ git switch -c feature
切换到一个新分支 'feature'
kiko@Hkiko:~/gitclone/gitconfit$ git branch
* feature
  master
```

3、在feature中，修改readme.txt，内容修改为：Creating a new branch is quick And Quick。然后提交：`git add readme.txt / git commit readme.txt -m "change a file"`

```
kiko@Hkiko:~/gitclone/gitconfit$ vim readme.txt
kiko@Hkiko:~/gitclone/gitconfit$ cat readme.txt
Creating a new branch is quick And Quick.
kiko@Hkiko:~/gitclone/gitconfit$ git add readme.txt
kiko@Hkiko:~/gitclone/gitconfit$ git commit readme.txt -m "change a file"
[feature 7ab84fb] change a file
1 file changed, 1 insertion(+), 1 deletion(-)
```

4、切换到主分支master，此时，主分支继续对readme.txt进行修改，内容修改为：Creating a new branch is quick & Quick。然后提交：`git add readme.txt / git commit readme.txt -m "change a file"`

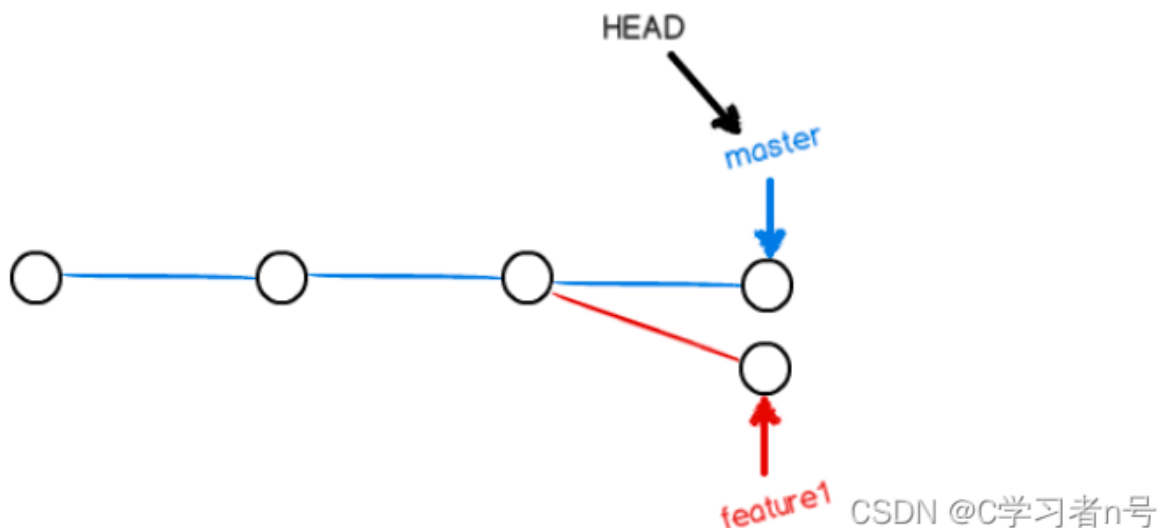
```
kiko@Hkiko:~/gitclone/gitconfit$ git switch master
切换到分支 'master'
kiko@Hkiko:~/gitclone/gitconfit$ vim readme.txt
kiko@Hkiko:~/gitclone/gitconfit$ cat readme.txt
Creating a new branch is quick & Quick.
kiko@Hkiko:~/gitclone/gitconfit$ git add readme.txt
kiko@Hkiko:~/gitclone/gitconfit$ git commit readme.txt -m "change a file"
[master be8224b] change a file
1 file changed, 1 insertion(+), 1 deletion(-)
```

5、此时，主分支和子分支都有了各自单独的提交。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。

以上操作的修改必然会造成冲突，因为修改的内容是一样的，现在各自都有不一样的版本。

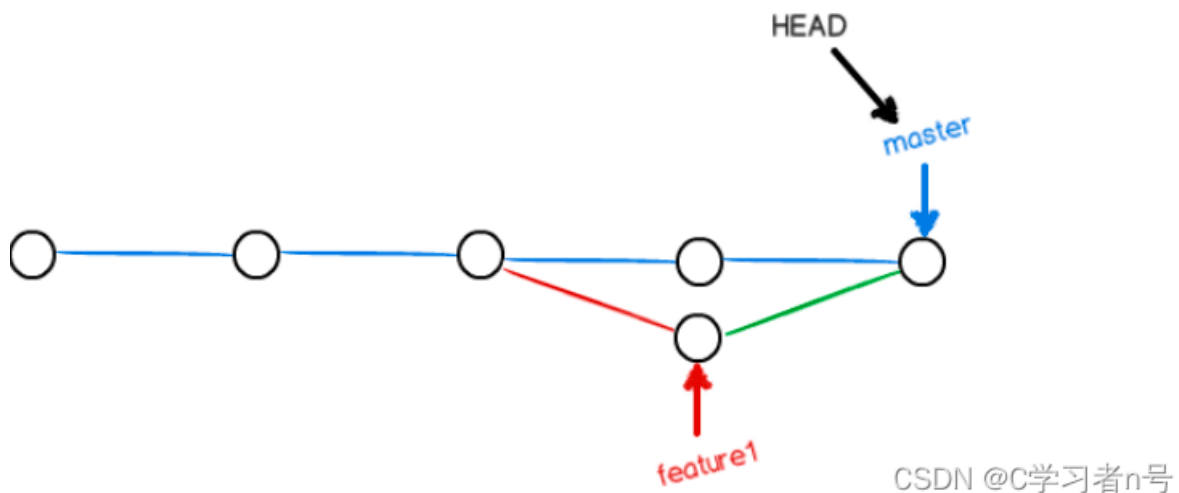
现在通过 `git merge feature` 合并，会报冲突。现在可以通过 `git status` 查看冲突在哪。

虽然合并操作出现了冲突，但是他还是执行了操作，已经将两者文件的内容也合并了（冲突的内容位置有类似乱码的冲突标记）



```
kiko@Hkiko:~/gitclone/gitconfit$ git merge feature
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。
kiko@Hkiko:~/gitclone/gitconfit$ cat readme.txt
<<<<<<< HEAD
Creating a new branch is quick & Quick.
=====
Creating a new branch is quick And Quick.
>>>>>> feature
```

6、因为造成了冲突就要解决，现在只能通过手动修改本地冲突位置的错误，然后上传提交。  
修改之后然后提交：`git add readme.txt / git commit -m "change a file"`。  
这里的commit提交不能指定文件名，否则会报：



```
kiko@Hkiko:~/gitclone/gitconfit$ git commit readme.txt -m "change a file"
fatal: 在合并过程中不能做部分提交。
```

```
kiko@Hkiko:~/gitclone/gitconfit$ cat readme.txt
Creating a new branch is quick and Quick.
kiko@Hkiko:~/gitclone/gitconfit$ git add readme.txt
kiko@Hkiko:~/gitclone/gitconfit$ git commit readme.txt -m "change a file"
fatal: 在合并过程中不能做部分提交。
kiko@Hkiko:~/gitclone/gitconfit$ git commit -m "change a file"
[master 10b2006] change a file
```

7、`git log --graph --pretty=oneline --abbrev-commit` 该命令可以查看分支合并情况。

8、最后，feature分支没有作用了可以删除 `git branch -d feature`

## 分支管理策略

1、正常情况下，合并分支的时候，如果可以，Git会用Fast forward模式；该模式虽然简单快速，但是在这种模式下，删除分支后，会丢掉分支信息（在分支删除之前查看日志信息，删除之后查看日志信息）

分支删除之前：在分支删除之前可以看到，通过日志还是可以看到有日志分支的信息的

```
kiko@Hkiko:~/gitclone/gitsto$ git init
已初始化空的 Git 仓库于 /home/kiko/gitclone/gitsto/.git/
kiko@Hkiko:~/gitclone/gitsto$ vim a.txt
```

```

kiko@Hkiko:~/gitclone/gitsto$ cat a.txt
master
kiko@Hkiko:~/gitclone/gitsto$ git add a.txt
kiko@Hkiko:~/gitclone/gitsto$ git commit -m "add a file"
[master (根提交) ba1e82b] add a file
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt
kiko@Hkiko:~/gitclone/gitsto$ git status
位于分支 master
无文件要提交，干净的工作区

kiko@Hkiko:~/gitclone/gitsto$ git switch -c feature1
切换到一个新分支 'feature1'
kiko@Hkiko:~/gitclone/gitsto$ vim a.txt
kiko@Hkiko:~/gitclone/gitsto$ cat a.txt
master - feature1
kiko@Hkiko:~/gitclone/gitsto$ git log
commit ba1e82b3c0ce185fa4e595f8fe3ee8acfee1e586 (HEAD -> feature1, master)
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:11:59 2022 +0800

    add a file
kiko@Hkiko:~/gitclone/gitsto$ git switch master
M      a.txt
切换到分支 'master'
kiko@Hkiko:~/gitclone/gitsto$ git merge feature1
已经是最新的。
kiko@Hkiko:~/gitclone/gitsto$ cat a.txt
master - feature1
kiko@Hkiko:~/gitclone/gitsto$ git log
commit ba1e82b3c0ce185fa4e595f8fe3ee8acfee1e586 (HEAD -> master, feature1)
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:11:59 2022 +0800

    add a file
kiko@Hkiko:~/gitclone/gitsto$ git log --graph --pretty=oneline --abbrev-commit
* ba1e82b (HEAD -> master, feature1) add a file

```

分支删除之后，可以看到日志信息中没有了分支的信息，看不出来是经历了分支合并：

```

kiko@Hkiko:~/gitclone/gitsto$ git branch -d feature1
已删除分支 feature1（曾为 ba1e82b）。
kiko@Hkiko:~/gitclone/gitsto$ git log --graph --pretty=oneline --abbrev-commit
* ba1e82b (HEAD -> master) add a file
kiko@Hkiko:~/gitclone/gitsto$ git log
commit ba1e82b3c0ce185fa4e595f8fe3ee8acfee1e586 (HEAD -> master)
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:11:59 2022 +0800

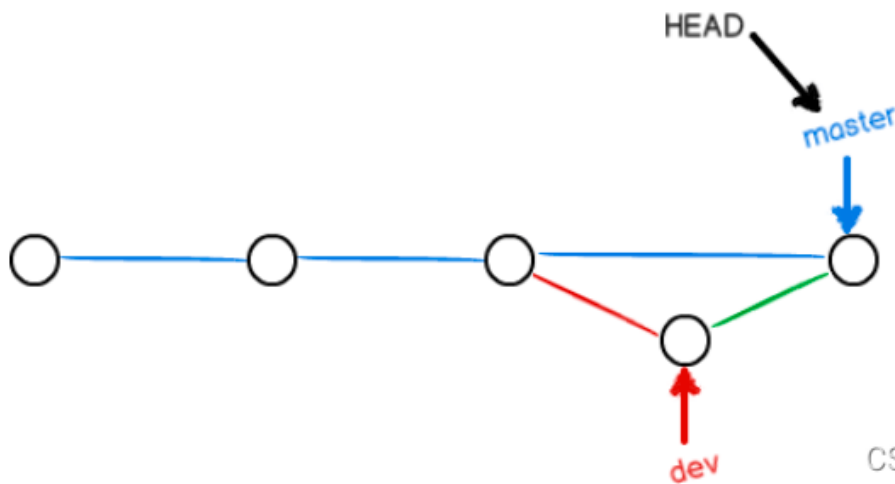
    add a file

```

2、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信息。总之，Fast forward模式看不出之前做过合并；而禁用之后的普通模式可以看出来以前做过合并。

不使用Fast forward模式的合并示意图（如上节的合并冲突，走的就是这种，所以合并冲突的情况是默

认不使用Fast forward模式的)：



CSDN @C学习者n号

```
kiko@Hkiko:~/gitclone$ mkdir gitff
kiko@Hkiko:~/gitclone$ cd gitff
kiko@Hkiko:~/gitclone/gitff$ git init
已初始化空的 Git 仓库于 /home/kiko/gitclone/gitff/.git/
kiko@Hkiko:~/gitclone/gitff$ vim readme.txt
kiko@Hkiko:~/gitclone/gitff$ git add readme.txt
kiko@Hkiko:~/gitclone/gitff$ git commit -m "add a file"
[master (根提交) 8ef8ad7] add a file
 1 file changed, 1 insertion(+)
 create mode 100644 readme.txt

kiko@Hkiko:~/gitclone/gitff$ git switch -c dev
切换到一个新分支 'dev'
kiko@Hkiko:~/gitclone/gitff$ vim readme.txt
kiko@Hkiko:~/gitclone/gitff$ cat readme.txt
Fast forward!!!
分支添加在这第二行
kiko@Hkiko:~/gitclone/gitff$ git add readme.txt
kiko@Hkiko:~/gitclone/gitff$ git commit -m "change a file"
[dev b997659] change a file
 1 file changed, 1 insertion(+)

kiko@Hkiko:~/gitclone/gitff$ git switch master
切换到分支 'master'
```

3、合并后，可以通过git log查看分支信息，与使用Fast Forward模式进行对比。

git merge使用--no-ff参数，表示禁用Fast forward模式，`git merge --no-ff -m "merge with no-ff" dev` --> 合并时会创建一个新的commit，加上-m参数可以添加提交点的描述信息

```
kiko@Hkiko:~/gitclone/gitff$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
kiko@Hkiko:~/gitclone/gitff$ git log
commit d4a7d3c02bce7f31189af70e5a4131bd0e2f0dd1 (HEAD -> master)
Merge: 8ef8ad7 b997659
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:27:55 2022 +0800

    merge with no-ff
```

```

commit b997659a6509b806a7aee470d241fd1cb58bbf1a (dev)
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:27:05 2022 +0800

    change a file

commit 8ef8ad7b26d3ed2ca14555435967947197bb1c76
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:24:11 2022 +0800

    add a file
kiko@Hkiko:~/gitclone/gitff$ git log --graph --pretty=oneline --abbrev-commit
* d4a7d3c (HEAD -> master) merge with no-ff
|\
| * b997659 (dev) change a file
|/
* 8ef8ad7 add a file

```

#### 4、删除分支之后仍然可以看到分支的信息

```

kiko@Hkiko:~/gitclone/gitff$ git branch -d dev
已删除分支 dev (曾为 b997659)。
kiko@Hkiko:~/gitclone/gitff$ git log --graph --pretty=oneline --abbrev-commit
* d4a7d3c (HEAD -> master) merge with no-ff
|\
| * b997659 change a file
|/
* 8ef8ad7 add a file
kiko@Hkiko:~/gitclone/gitff$ git log
commit d4a7d3c02bce7f31189af70e5a4131bd0e2f0dd1 (HEAD -> master)
Merge: 8ef8ad7 b997659
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:27:55 2022 +0800

    merge with no-ff

commit b997659a6509b806a7aee470d241fd1cb58bbf1a
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:27:05 2022 +0800

    change a file

commit 8ef8ad7b26d3ed2ca14555435967947197bb1c76
Author: transit <553354863@qq.com>
Date: Thu Jul 21 08:24:11 2022 +0800

    add a file

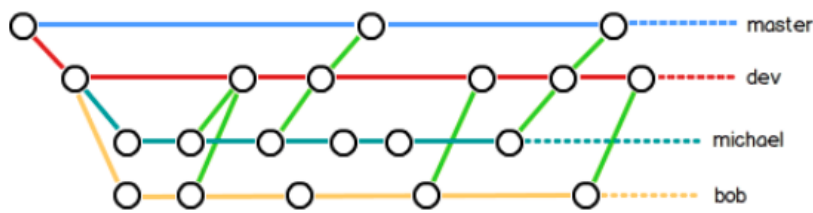
```

#### 5、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，每个人都有自己的其他分支，时不时地往dev分支上合并就可以了。



CSDN @C学习者n号

## bug分支

- 背景：软件开发中，bug经常出现。在Git中，由于分支的存在，所以，每个bug都可以通过一个临时分支来修复，修复后，合并分支，然后将临时分支删除即可。
- 出现如下情况：在开发过程中，接到了一个修复代号为100的bug任务，此时，就自然的想到通过创建一个临时分支（bug-100）来修复它。  
但是，此时正在dev分支上进行的工作还没有结束，工作进行到一半无法提交。  
git提供了一个stash功能，可以把当前的工作现场储藏起来，等以后恢复现场后继续工作：git stash。  
此时通过git status来查看工作区，它是干净的（除非有文件没有被git管理），此时可以去修复100bug
- 首先确定bug是在哪个分支，假设他是在master主分支，此时就从master创建一个临时分支bug-100用于修复这个bug  
假设这里的bug修复就是修改其中的某一个文件内容就完成了bug修复，然后提交即可：git add readme.txt git commit -m "fix bug 100"  
之后，切换到master分支，并完成合并，最后删除这个临时的bug-100分支：git switch master、git merge --no-ff -m "merge bug fix100" bug-100
- bug修复完成之后，就可以继续回到dev分支工作了，用git stash list可以查看储藏起来的工作现场。  
要想继续之前的工作就要将其恢复到原来状态，有两种方式恢复：git stash apply --> 恢复后，储藏起来的工作现场还存在，需要使用git stash drop 删除之  
git stash pop --> 恢复工作现场的同时将stash内容也删掉  
值得注意的是git stash pop恢复的是最近一次储存的工作现场，所以执行多次才能恢复所有的。  
git stash apply stash@{0}可以恢复指定的工作现场
- 此时就已经可以在恢复的工作现场工作了。现在思考dev分支是早期从master分支分支出来的，所以，这个bug其实在当前dev分支上也存在。  
那么如何修复dev分支的bug？重复修改一次，然后提交就可以了。  
同样的bug，要在dev上修复，我们只需要把4c805e2 fix bug 100这个提交所做的修改“复制”到dev分支。  
注意：我们只想复制4c805e2 fix bug 100这个提交所做的修改，并不是把整个master分支merge过来。  
为了方便操作，git专门提供了一个cherry-pick命令，让我们能复制一个特定的提交到当前分支（git cherry-pick 4c805e2）  
git自动给dev分支做了一次提交，注意这次提交的commit是1d4b803，它并不同于master的4c805e2，因为这两个commit只是改动相同确实是两个不同的commit。  
用git cherry-pick，我们就不需要在dev分支上手动再把修bug的过程重复一遍。

CSDN @VVPU

如下代码准备了一个开发环境进行测试：

现在，有主分支master中有my.txt，然后分支出dev，然后在dev上面工作，写了一个mydev.c文件。

```
kiko@Hkiko:~/gitclone/gitbug$ vim my.txt
kiko@Hkiko:~/gitclone/gitbug$ cat my.txt
Git is a Fast And Simple Distributed System!!!
kiko@Hkiko:~/gitclone/gitbug$ git add my.txt
kiko@Hkiko:~/gitclone/gitbug$ git commit my.txt -m "add a file"
[master (根提交) 28fc019] add a file
1 file changed, 1 insertion(+)
create mode 100644 my.txt

kiko@Hkiko:~/gitclone/gitbug$ git switch -c dev
切换到一个新分支 'dev'
kiko@Hkiko:~/gitclone/gitbug$ vim mydev.c
kiko@Hkiko:~/gitclone/gitbug$ git add mydev.c
kiko@Hkiko:~/gitclone/gitbug$ git status
位于分支 dev
要提交的变更：
  （使用 "git restore --staged <文件>..." 以取消暂存）
    新文件：    mydev.c
```

假设bug是出现在master分支上，就需要从master分支切出一条新的分支bug-100去解决这个bug，假设bug的解决就是在my.txt上添加了一行Bug-Fix内容就可以解决bug。

```
kiko@Hkiko:~/gitclone/gitbug$ git stash
保存工作目录和索引状态 WIP on dev: 28fc019 add a file
kiko@Hkiko:~/gitclone/gitbug$ git status
位于分支 dev
无文件要提交，干净的工作区
```

```
kiko@Hkiko:~/gitclone/gitbug$ git switch master
切换到分支 'master'
kiko@Hkiko:~/gitclone/gitbug$ git switch -c bug-100
切换到一个新分支 'bug-100'
kiko@Hkiko:~/gitclone/gitbug$ vim my.txt
kiko@Hkiko:~/gitclone/gitbug$ cat my.txt
Git is a Fast And Simple Distributed System!!!
Bug-Fix

kiko@Hkiko:~/gitclone/gitbug$ git add my.txt
kiko@Hkiko:~/gitclone/gitbug$ git commit -m "fix a bug-100"
[bug-100 606fcff] fix a bug-100
1 file changed, 1 insertion(+)
```

```
kiko@Hkiko:~/gitclone/gitbug$ git status
位于分支 bug-100
无文件要提交，干净的工作区
kiko@Hkiko:~/gitclone/gitbug$ git switch master
切换到分支 'master'
kiko@Hkiko:~/gitclone/gitbug$ git merge bug-100
更新 28fc019..606fcff
Fast-forward
 my.txt | 1 +
1 file changed, 1 insertion(+)

kiko@Hkiko:~/gitclone/gitbug$ cat my.txt
Git is a Fast And Simple Distributed System!!!
Bug-Fix

kiko@Hkiko:~/gitclone/gitbug$ git switch dev
切换到分支 'dev'
kiko@Hkiko:~/gitclone/gitbug$ git stash list
stash@{0}: WIP on dev: 28fc019 add a file
kiko@Hkiko:~/gitclone/gitbug$ git stash pop
位于分支 dev
要提交的变更:
  (使用 "git restore --staged <文件>..." 以取消暂存)
  新文件:   mydev.c

丢弃了 refs/stash@{0} (24ee31c184c0598808732f0bc66cef8da15d14ce)
kiko@Hkiko:~/gitclone/gitbug$ git stash list

kiko@Hkiko:~/gitclone/gitbug$ cat my.txt
Git is a Fast And Simple Distributed System!!!
kiko@Hkiko:~/gitclone/gitbug$ vim mydev.c
kiko@Hkiko:~/gitclone/gitbug$ git add mydev.c
kiko@Hkiko:~/gitclone/gitbug$ git commit -m "complete mydev.c program"
[dev 3f7bdef] complete mydev.c program
1 file changed, 19 insertions(+)
 create mode 100644 mydev.c
kiko@Hkiko:~/gitclone/gitbug$ git status
位于分支 dev
无文件要提交，干净的工作区

kiko@Hkiko:~/gitclone/gitbug$ git cherry-pick 606fcff
[dev 9cc25bc] fix a bug-100
Date: Thu Jul 21 22:20:53 2022 +0800
```

```
1 file changed, 1 insertion(+)
```

## feature分支

1、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，肯定不希望因为一些实验性质的代码，把主分支搞乱；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。

2、假设现在有一个代号为fun1的新功能要开发；

首先创建并且切换到新分支feature-fun1 --> `git switch -c feature-fun1`

开发完毕，并且提交了--> `git add feature-fun1 / git commit -m "add feature-fun1"`

3、现在切回dev，`git switch dev`

但是，就在合并之前，又接到要求，不需要该新功能；

此时，需要将这个新功能分支删除销毁

`git branch -d feature-fun1 -->` 销毁失败，该分支还没有被合并，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；`git branch -D feature-fun1`。

## 多人协作

- 1、当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的main分支对应起来，并且，远程仓库默认使用的名称是origin  
可以通过`git remote`查看远程库的信息，`git remote -v`可以显示更加详细的信息
- 2、推送分支 --> 就是把这个分支上的所有本地提交推送到远程仓库，推送时，需要指定本地分支，Git就会把该分支推送到远程仓库中对应的远程分支。  
`git push origin master`：推送主分支到远程的主分支（两者默认对应好了）  
`git push origin dev`：推送其它分支，比如dev

- 3、并不是一定要把本地分支往远程推送，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。  
总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

- 4、抓取分支  
多人协作的时候，通常会往master和dev分支上推送各自的修改  
现在，模拟一个新的用户（要把新用户电脑上的SSH添加到这个GitHub中）或者直接在另一个目录下克隆：  
`git clone git@github.com:michaelliao/learn-git.git` 克隆远程仓库。  
正常情况下，本地只可以看到master主分支。可以用`git branch`命令查看。

现在，新用户要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是使用如下命令：

```
git checkout -b dev origin/dev --> 在本地创建（切换）和远程分支对应的分支，
因为只是相当于把远程同名分支的内容复制到了当前分支，并没有建立关联
然后可以在dev分支上修改，并提交然后推送到远程：
git add env.txt git commit -m "add env"
git push origin dev --> 推送到远程分支dev上（指定推送的远程分支，没有加上-u参数，下次推送还要加上远程分支，不能简写为git push）
```

此时另一个用户也对该分支同一个文件做了修改，并且视图推送：

```
git push origin dev （指定推送的远程分支，没有加上-u参数，下次推送还要加上远程分支，不能简写为git push）
推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，
Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：
git pull （简写为git pull需要将本地和远程的分支关联起来，但是还没有关联）
git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：
git branch --set-upstream-to=origin/dev dev
再pull: git pull
这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后，提交，再push：
git commit -m "fix env conflict"
git push origin dev
```

注：`git push -u origin dev -->` 其中-u参数是在第一次推送的时候添加的，表示当前本地分支与指定的远程分支origin dev分支关联  
将本地的分支版本上传到远程合并，并且记录push到远程分支的默认值；  
当添加“-u”参数时，表示下次继续push的这个远端分支的时候推送命令就可以简写成“git push”。

将本地的分支版本上传到远程合并，并且记录push到远程分支的默认值；  
当添加“-u”参数时，表示下次继续push的这个远端分支的时候推送命令就可以简写成“git push”。

既然有了这个参数-u，那么为什么还要通过命令进行绑定，主要是为了应对一次推送都没有成功的情况，此时就需要通过命令来关联本地分支和远程分支

- 5、多人协作工作模式
  - 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，则因为远程分支也被人修改提交推送过，需要用`git pull`试图拉取合并（拉取之后主动合并）；
  - 如果合并有冲突，手动解决文件冲突，并在本地提交；
  - 没有冲突或者解决冲突后，再用`git push origin <branch-name>`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关系没有创建，用命令`git branch --set-upstream-to <branch-name> origin/<branch-name>`。

CSDN @VVPVU

## 标签管理

- 1、说明  
标签就是版本库的快照，指向某一次提交方便记录。
- 2、创建标签  
在Git中打标签首先要切换到需要打标签的分支上，通常是master主分支：  

```
git switch master  
git tag v1.0 # git tag <tag-name> 默认标签打在最新的提交上
```

  
如果最新的一次不是自己想要打标签的提交，那么可以先通过git log --pretty=oneline --abbrev-commit查看日志信息，找到想要的id  
然后 git tag v1.8 cf810da  
  
# 还可以创建带有说明的标签： git tag -a v2.0 -m "this is a tag" cf810da  
# git tag 可以查看所有的标签： git show <tag-name> 查看指定的标签的详细信息
- 3、操作标签  

```
git tag -d v1.8 # 删除指定的标签
```

  
创建的标签都只会存储在本地，不会自动推送到远程仓库，所以，打错的标签可以在本地安全删除  
  
如果要推送某一个标签到远程，使用命令git push origin <tag-name>;  
或者一次性推送全部尚未推送到远程的本地标签： git push origin --tags  
  
如果要删除远程标签： 第一步删除本地标签： git tag -d v1.9 ; 然后删除远程标签： git push origin :refs/tags/v1.9

CSDN @VVPU

```
iko@Hkiko:~/gitclone/gitbug$ git tag v1.8  
kiko@Hkiko:~/gitclone/gitbug$ git tag  
v1.8  
kiko@Hkiko:~/gitclone/gitbug$ git show v1.8  
commit 606fcff188dea1e4201b894540b387382c39a2c2 (HEAD -> master, tag: v1.8)  
Author: transit <553354863@qq.com>  
Date: Thu Jul 21 22:20:53 2022 +0800  
  
    fix a bug-100  
  
diff --git a/my.txt b/my.txt  
index bf96956..922c37a 100644  
--- a/my.txt  
+++ b/my.txt  
@@ -1,2 @@  
    Git is a Fast And Simple Distributed System!!!  
+Bug-Fix  
kiko@Hkiko:~/gitclone/gitbug$ git tag -d v1.8  
已删除标签 'v1.8' (曾为 606fcff)
```

## 忽略特殊文件

- 1、背景：有时候，某些文件放到了Git工作目录中，但是又不能提交它们，比如保存了数据库密码的配置文件等；如果不将这些文件交由Git管理，通过git status查看工作目录状态时，会出现一些提示，比较碍眼（没有错误）。  
  
此时，Git提供了一个特殊的名为.gitignore文件，在工作根目录下。创建这个文件，并且提交到仓库即可。  
文件里面填写一些需要忽略的文件名，Git就会自动忽略这些文件。  
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。  
所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>  
忽略文件规则：
  - 忽略操作系统自动生成的文件，比如缩略图等；
  - 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，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
  - 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、强制添加被忽略的文件  
比如有一个App.class想要提交，但是被忽略规则忽略了，此时可以通过：git add -f App.class 强制添加App.class文件  
如果不想破坏规则，那么可以在.gitignore文件里面添加上一条规则：!App.class，表示不忽略App.class文件  
此外，如果遇到了App.class被忽略的问题，可以通过git check-ignore -v App.class来忽略检查，哪一条规则将这个文件忽略  
  
如果.gitignore文件中有一条规则：.\*，那么他也会把这个忽略文件给忽略了，此时需要添加规则：!.gitignore

CSDN @VVPU

## 配置别名和配置文件

1、配置别名主要是为了好记命令，如下：

```
# --global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.br branch
git config --global alias.unstage 'reset HEAD'
git config --global alias.last 'log -1'
```

2、配置文件

配置文件放在.git/config文件中

别名配置就在[alias]后面，要删除别名，直接把对应的行删掉即可。

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

CSDN @VVP

## 命令总结（全部，功能分类）

仓库初始化：

`git init` --> 将当前目录设置为git的工作目录，可以被git所管理

设置用户和邮箱(全局，在git中的所有操作都是该用户的操作)

`git config --global user.name "vv"`

`git config --global user.email "553354863@qq.com"`

添加和提交

`git add <directory> / <file>` # 将指定目录的所有修改加入到下一次commit中(暂存区)。把<directory>替换成<file>将添加指定文件的修改

`git commit -m "message"` #提交暂存区的修改，使用指定的<message>作为提交信息

状态信息

`git status` # 显示哪些文件已被staged(暂存状态)、未被staged以及未跟踪(untracked)。

`git diff` # 比较工作区和暂存区的修改。

`git diff HEAD` # 比较工作区和上一次commit后的修改。

`git diff --cached` # 比较暂存区和上一次commit后的修改。

## 可以在命令后面添加 `-- read.txt`表示只判断该文件

`git diff HEAD -- readme.txt` ## 比较工作区和版本库中的readme.txt文件

日志

`git log` # 显示详细的日志信息，全部显示。

`git log -<limit>` # 显示log，限制log的显示数量。例如：“`git log -5`”仅显示最新5条commit

`git log --oneline` # 每行显示一条commit。简洁版本

`git reflog` # 显示本地仓库的所有commit日志

版本回退（针对所有修改的文件）

`git reset --hard #` 移除所有暂存区的修改，并强制删除所有工作区的修改。此时工作区、暂存区和版本库中的版本会一致。（回退到最近的一次提交）

`git reset #` 移除所有暂存区的修改，但不会修改工作区。此时的暂存区和版本库一致。（回退到最近的一次提交）

`git reset --hard <commit id>#` 移除所有暂存区的修改，并强制删除所有工作区的修改。此时工作区、暂存区和版本库中的版本会一致。（回退到指定的一次提交）

`git reset <commit id> #` 移除所有暂存区的修改，但不会修改工作区。此时的暂存区和版本库一致。（回退到指定的一次提交）

`git reset HEAD <file_name> #` 丢弃暂存区的修改，重新放回工作区，会将暂存区的内容和本地已提交的内容全部恢复到未暂存的状态，不影响原来本地文件（相当于撤销`git add`操作，不影响上一次`commit`后对本地文件的修改。此时暂存区和版本库统一，工作区还有修改后的内容）（包括对文件的操作，如添加文件、删除文件）

`git reset --hard HEAD #` 清空暂存区，将已提交的内容的版本恢复到本地，本地的文件也将被恢复的版本替换（恢复到上一次`commit`后的状态，上一次`commit`后的修改也丢弃，此时三方版本统一）

撤销修改（文件内容 / 文件本身）--> 针对某一个文件

`git restore --staged <file_name> #` 将暂存区的修改重新放回工作区（包括对文件自身的操作，如添加文件、删除文件）

`git restore <file_name> #` 丢弃工作区的修改（不包括对文件自身的操作，如添加文件、删除文件）

`git checkout -- <file_name> #` 丢弃工作区的修改，并用最近一次的`git add`或者`git commit`状态时的状态，还原到当前工作区（对文件中内容的操作，无法对添加文件、删除文件起作用）

`git checkout HEAD^ -- <file_name> #` 将指定`commit`提交的内容(`HEAD^`表示上一个版本)还原到当前工作区

删除文件

`git rm file #` 删除指定文件，且提交到了暂存区。

## 分支

# 创建分支

`git branch <branch-name> #` 创建分支

# 创建 / 切换分支

`git checkout -b <branch-name> #` 创建并且切换分支

`git checkout <branch-name> #` 仅切换分支

`git switch -c <branch-name> #` 创建并且切换分支

`git switch <branch-name> #` 切换分支

# 删除分支

`git branch -d <branch-name> #` 将指定的分支删除，但是不能此时就在要删除的分支中

`git branch -D <branch-name> #` 丢弃一个没有合并过的子分支或者一个没有完全合并的子分支，如果使用`-d`作为参数是失败操作，他只能删除合并过了的分支。

# 合并分支

`git merge <branch-name> #` 将指定的分支合并到当前所在的分支中

`git merge --no-ff -m "merge with no-ff" <branch-name> #` 合并分支，使用`--no-ff`参数，表示禁用Fast forward模式，合并时会创建一个新的`commit`，加上`-m`参数可以添加提交点的描述信息

# 查看分支日志情况

`git log --graph --pretty=oneline --abbrev-commit #` 可以查看分支合并情况（包括日志信息）。



## # Git储藏

**git stash** # 将未完成的工作空间临时储藏起来，通过**git status**查看工作空间，它是干净的（除了有没有被Git管理的文件）。

**git stash list** # 查看储藏区的被储藏起来的工作空间

**git stash apply stash@{0}** --> 恢复指定工作现场后，储藏起来的工作现场还存在储藏区，**stash@{0}**是储藏的编号（如果不指定，那么就默认最近一次的储藏）

**git stash drop stash@{0}** --> 删除储藏区的一个存储（和**git stash apply**搭配使用）

**git stash pop** --> 从git栈中获取到最近一次**stash**进去的内容，恢复工作区的内容。获取之后，会删除栈中对应的**stash**。

**git cherry-pick <commit id>**

#将指定分支的修改复制到当前所在分支，主要是针对bug的修复，**commit id**是另一个分支修复bug后提交的**id**，当前分支也有这个bug，可以这样解决当前分支bug

## 远程仓库

**git remote add <名称> <地址>**

# 起别名。比如**git remote add origin git@github.com:java-sys/gitrepository.git** 就是把一串地址起别名为**origin**,方便操作。

**git remote [-v | --verbose]**

# 查看远程仓库地址，别名 + 地址（有**-v**参数） + **fetch / push**：查看远程仓库地址别名（没有**-v**参数）

**git remote rename <旧名称> <新名称>**

# 修改别名。

**git remote remove <名称>**

# 移除指定远程主机别名。之后将无法通过这个别名推送数据。

# 但是还是可以直接用远程主机地址来操作。别名只是为了方便记忆地址，并不会影响连接远程主机的问题。

## 远程分支（多人协作部分）

**git branch -M <branch-name>** # 将当前分支改名为指定的名称。通常往github上传一个本地仓库会有**git branch -M 'main'**的提示，要求将本地的**master**分支改名为**main**

**git branch --set-upstream-to=origin/dev dev** # 将指定的两个分支关联

**git checkout -b dev origin/dev**

# 在本地创建并切换和远程分支对应的分支，因为只是相当于把远程同名分支的内容复制到了当前分支，并没有建立关联

## 克隆与推送

**git clone git@github.com:michaelliao/learngit.git** # 克隆远程仓库到本地。

**git push -u origin dev**

# 其中**-u**参数是在第一次推送的时候添加的，表示当前本地分支与指定的远程分支**origin dev**分支关联。

# 将本地的分支版本上传到远程合并，并且记录**push**到远程分支的默认值：

# 当添加“**-u**”参数时，表示下次继续**push**的这个远端分支的时候推送命令就可以简写成“**git push**”。

**git push origin dev** # 向指定远程分支推送当前分支数据

**git push** # 推送当前分支的数据到远程仓库对应的分支（要求两侧的分支已经关联）

**git pull** # 从远程分支拉取数据到本地当前分支并且进行分支合并。（要求两侧的分支已经关联）

## 标签

# 创建标签

**git tag <tag-name>** # 创建指定名称的的标签，该标签指向当前分支的最新一次提交

**git tag <tag-name> <commit id>** # 创建标签，指向指定的提交版本

**git tag -a <tag-name> -m <message> <commit id>** # 创建带有标签信息的标签

# 查看标签

`git tag` # 查看所有标签

`git show <tag-name>` # 查看指定标签的详细信息

# 本地与远程标签

`git tag -d <tag-name>` # 删除指定标签（本地）

`git push origin <tag-name>` # 推送指定标签

`git push origin --tags` # 推送本地全部标签

`git push origin :refs/tags/<branch-name>` # 删除指定的远程标签（需要先删除本地标签，然后使用该命令）