Git 学习

概述

- Git 是目前世界上最先进的分布式版本控制系统。
- Linus 在 1991 年创建了开源的 Linux，目前已经成为最大的服务器系统软件了。
- GitHub 使用 C 语言开发
- 集中式版本控制：欲使用一个项目，先从服务器中取出来，修改，然后发到服务器。服务器集中管理控制。
- 分布式**版本控制**：没有中央控制器，每个人的电脑就是一个完整的版本库，工作的时候就不要联网。分布式也有一台中央服务器，用于交换不同个体的不同版本库的修改，同步项目。
- 分布式　+　版本控制

安装

Windows 上：

- 下载安装
- 自报家门：

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

Global 参数表示本机上所有的 git 仓库都会使用这个配置

创建版本库

版本库：即仓库：repository，可以理解为一个目录，该目录里面所有的文件都被 Git 管理，每个文件的修改都可以被跟踪
所以创建版本库：

- 选一个合适的文件目录，创建一个空文件夹（在 Git bush 中使用 mkdir 可以创建新文件夹，DOS 命令也是部分可用的）
  注意：使用含有中文的目录可能会错误，尽量避免。

- 初始化该仓库，将其变成 GitHub 可以管理的仓库
  命令：

```
$ git init
Initialized empty Git repository in /Users/michael/learngit/.git/
```

文件夹内多出了一个.git 文件：用于跟踪管理仓库

（注意：所有的版本控制系统，其实只能跟踪文本文件的改动，比如 TXT 文件，网页，所有的程序代码等等。Microsoft 的 Word 格式是二进制格式，因此，版本控制系统是没法跟踪 Word 文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。）

千万不要使用Windows自带的记事本编辑任何文本文件。原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Notepad++代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可：

- 把目录下的文件添加到仓库
  - Git add */filename1 filename2 …
  - Git commit -m “message”

查看状态

```
$ git status
```

如果文件被修改，会显示出来

使用 `$ git diff readme.txt` 或者

$ git diff 查看具体修改了什么内容。

版本回退与前进

如果文件出错，使用最近更新的文件来还原

Git log：查看修改历史

```
$ git log
commit 4a3b753e13234e01446073a4e8bacd919afd2c10 (HEAD -> master)
Author: FighterXutao <1603837506qq.com>
Date:   Thu Apr 25 19:25:16 2019 +0800

    Edwin's github learning
```

如果回退 n 个版本，使用：

git reset HEAD~n

Cat filename：查看文本内容

回退：

```
$ git reset --hard HEAD^
HEAD is now at e475afc add distributed
```

前进：

git reflog：查看命令历史，以便确定要回到未来的哪个版本，获取版本号

然后使用

git reset --hard 版本号

工作区&版本库

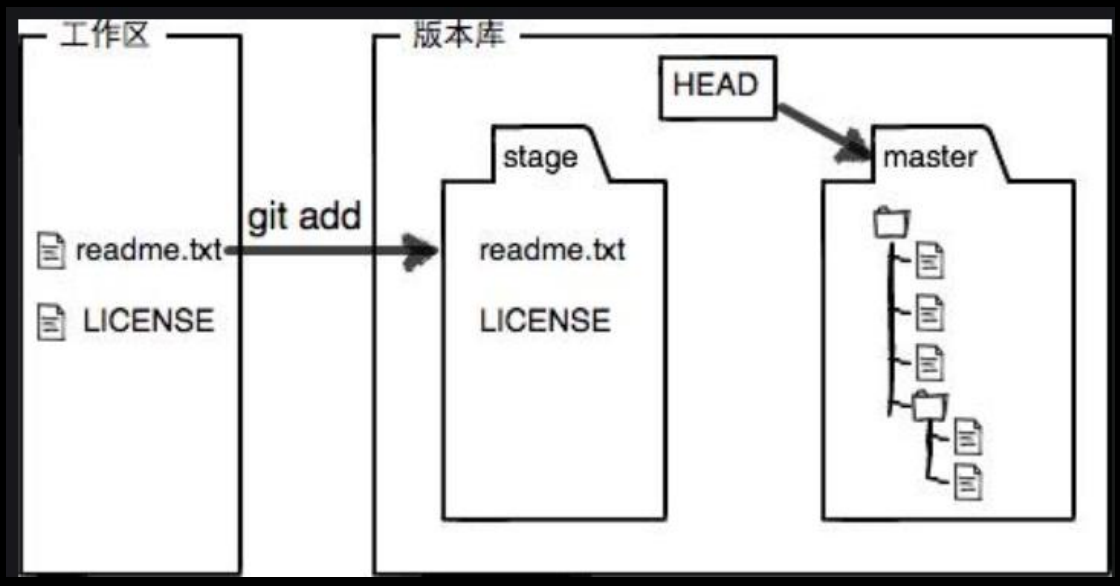
工作区：即电脑上的目录

版本库 Repository：.git 就是版本库

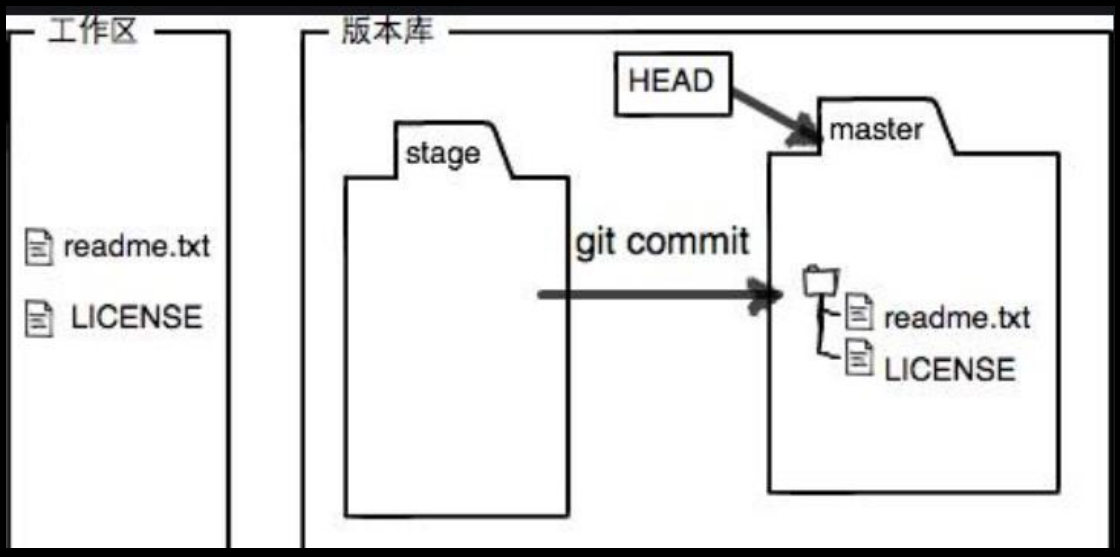
> 前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：
>
> 第一步是用`git add`把文件添加进去，实际上就是把文件修改添加到暂存区；
>
> 第二步是用`git commit`提交更改，实际上就是把暂存区的所有内容提交到当前分支。

> 因为我们创建Git版本库时，Git自动为我们创建了唯一一个`master`分支，所以，现在，`git commit`就是往`master`分支上提交更改。

工作区和版本库：



Git add git commit 更新文件到仓库
如果文件没改动，那么 git commit 就会显示工作区是“干净的”
同时，git commit 后，暂存区中的文件到了仓库，暂存区就空了



管理修改

Git 跟踪并管理的是修改，而非文件。
git diff HEAD -- readme.txt: 查看工作区和版本库里面最新版本的区别:

撤销修改

git checkout -- readme.txt: readme.txt 文件在工作区的修改全部撤销
git reset HEAD <file>可以把暂存区的修改撤销掉（unstage），重新放回工作区:

删除文件

git rm 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”。

（注意：git checkout vs git checkout --）

Origin

在默认情况下，origin 指向的就是你本地的代码库托管在 Github 上的版本。

远程仓库名字 “origin” 与分支名字 “master” 一样，在 Git 中并没有任何特别的含义一样。 同时 “master” 是当你运行 git init 时默认的起始分支名字，原因仅仅是它的广泛使用，**“origin” 是当你运行 git clone 时默认的远程仓库名字**。 如果你运行 git clone -o booyah，那么你默认的远程分支名字将会是 booyah/master。

远程仓库

使用 GitHub 网站—提供 Git 仓库托管服务

Step1创建 SSH Key

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有 id_rsa 和 id_rsa.pub 这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

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

如果一切顺利的话，可以在用户主目录里找到 .ssh 目录，里面有 id_rsa 和 id_rsa.pub 两个文件，这两个就是SSH Key的秘钥对，id_rsa 是私钥，不能泄露出去，id_rsa.pub 是公钥，可以放心地告诉任何人。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴 id_rsa.pub 文件的内容：

Step2

添加远程库

现在本地已经有一个仓库了，需要在 GitHub 创建一个 Git 仓库，使二者同步。

这样，GitHub 上的仓库既可以作为备份，有可以让其他人来协作。

- 在 GitHub 创建新仓库：名字和本地的相同。

  创建后，可以选择

- ◆ 从该仓库克隆出新的仓库，
- ◆ 也可以选择把该仓库与本地仓库关联

- git remote add origin git@github.com:Edwin-Xu/TEST.git

  后面的.git 使用仓库名.git

- git push -u origin master 把当前分支的 master 推送到远程。

  （第一次需要使用 -u 关联，后面就不需要了）

  如果出错需要重新 Git remote

  则：

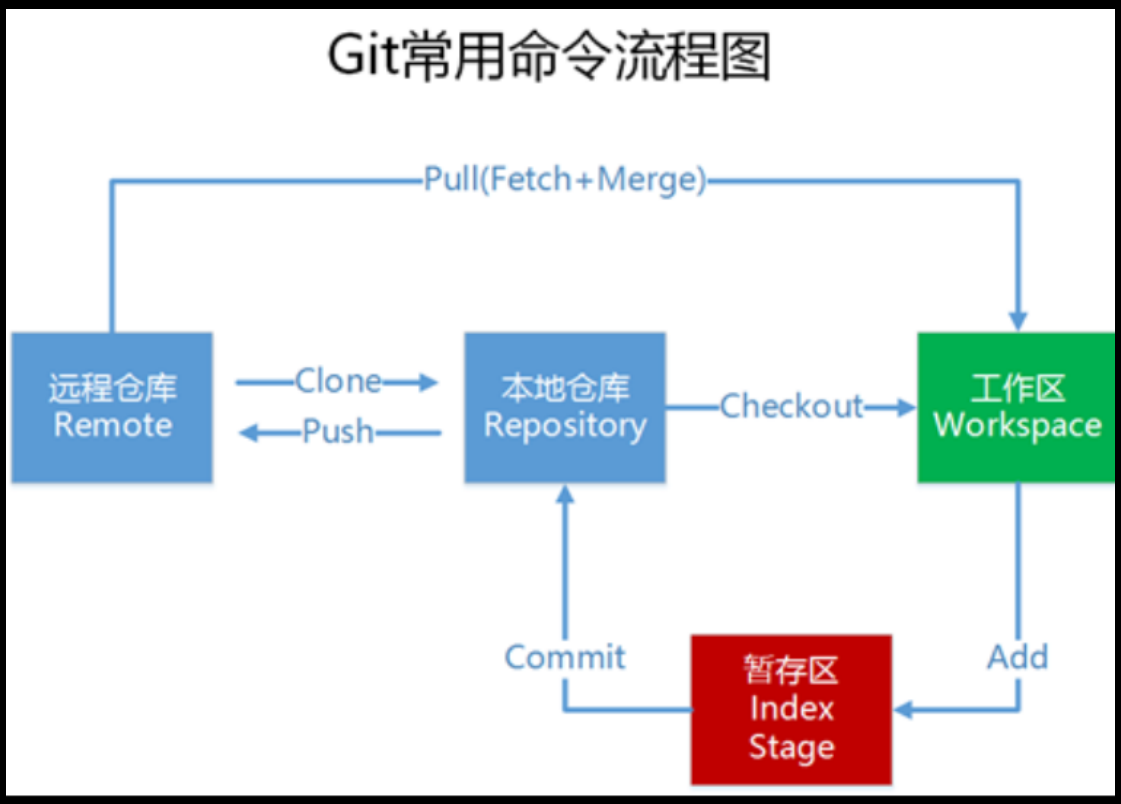
  ```
  git remote rm origin
  ```

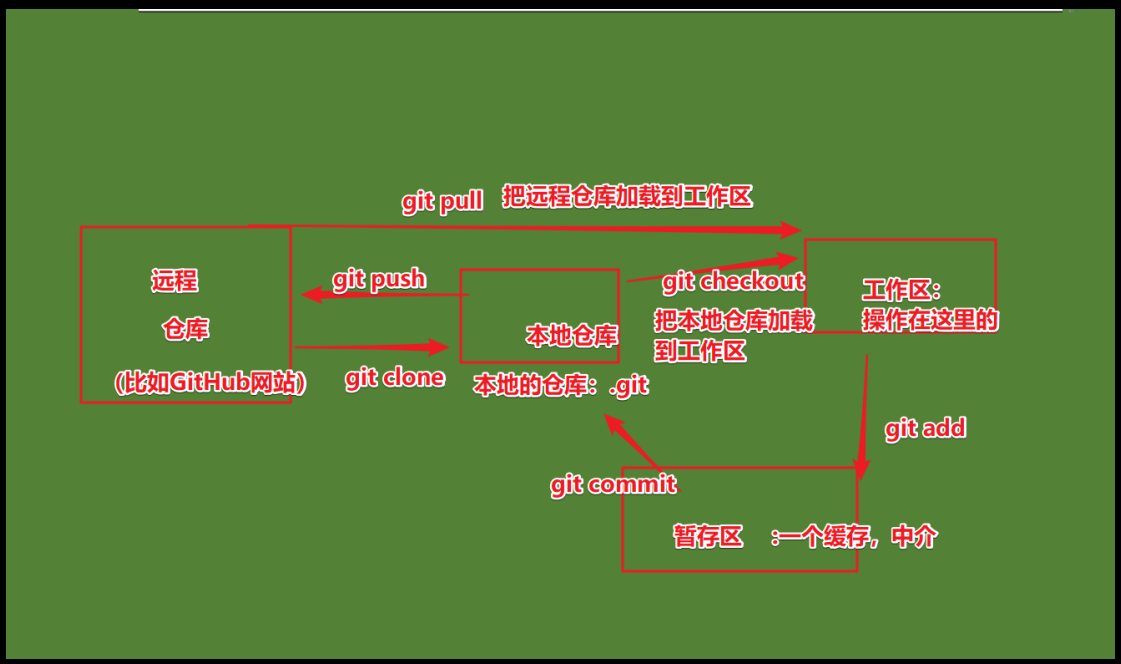
注意：前面的流程还需要走的：

1. **Git init**
2. **Git add**
3. **Git commit**
4. **Git remote add origin**
5. **Git push origin master**

从远程克隆

Git clone





分支管理

分支就是科幻电影里面的平行宇宙，互不影响。你可以开一个分支，在上面提交你未完成的作品，别人不可见，又可以继续在原来的分支工作。

HEAD

HEAD 表示当前版本，也就是**最新的提交**

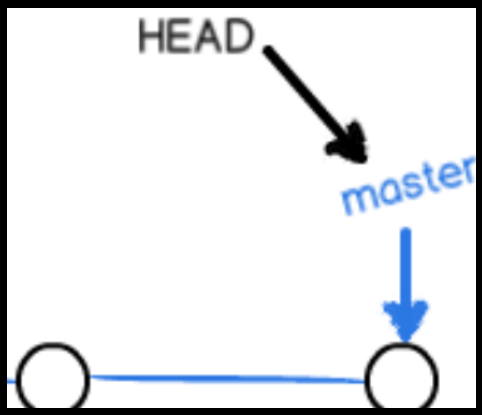
HEAD 指向的版本就是当前版本，因此，Git 允许我们在版本的历史之间穿梭，

使用命令 git reset --hard commit_id

创建和合并分支

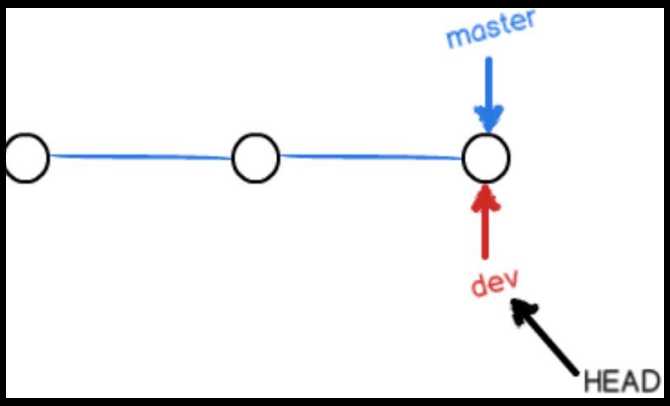
Master 分支是主分支，master 指向提交，而 HEAD 指向 master，即指向当前分支

Head 指向 master，master 指向最新提交，就能确定当前分支

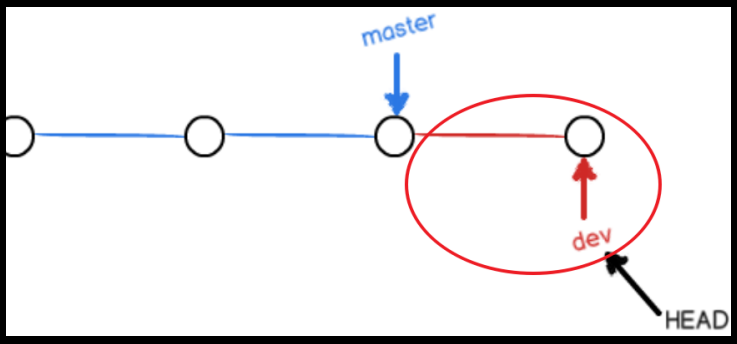


每次提交，master 主分支这条线越来越长

当创建新分支 Dev 时，git 新建一个指针较 dev 指向 master 相同的提交，再把 head 指向 dev（表示当前分支在 dev 上）

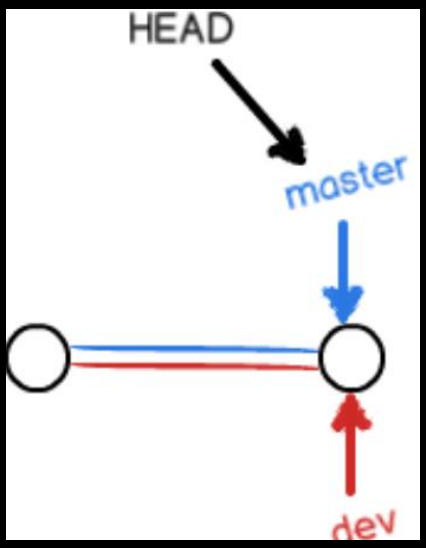


现在，对工作区的修改和提交就是针对 dev 分支了，master 指针不变



合并：

合并就直接把 master 指向 dev 当前提交：



合并后也可以把 dev 分支删除掉

实际操作：

- 创建分支，切换到 dev 上：

  Git checkout -b branchName

  (相当于：

  Git branch dev;

  Git checkout dev;

)

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

- 查看当前分支：

  Git branch

```
$ git branch   列出所有分支
* dev    *表示当前分组
  master
```

- 对分支进行操作：

  Git add

  Git commit

  注意：现在就算切换回 master（git checkout master），也不会显示刚才的修改。因为两个分支是独立的，只有合并后才在 master 可见。

- 合并：
  Git merge dev 的
  现在 master 可见在分支上的修改
- 删除分支：
  Git branch -d dev

注意：
Git checkout <branch>是撤销，而 git checkout --<file>是撤销修改。

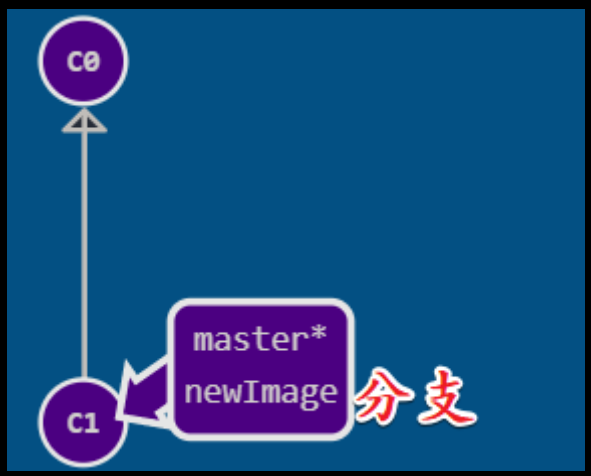
所有切换分支可以使用：git switch
Git switch -c dev
Git switch master

Git Banch Learning

Git 的提交是一个链式的父子关系的

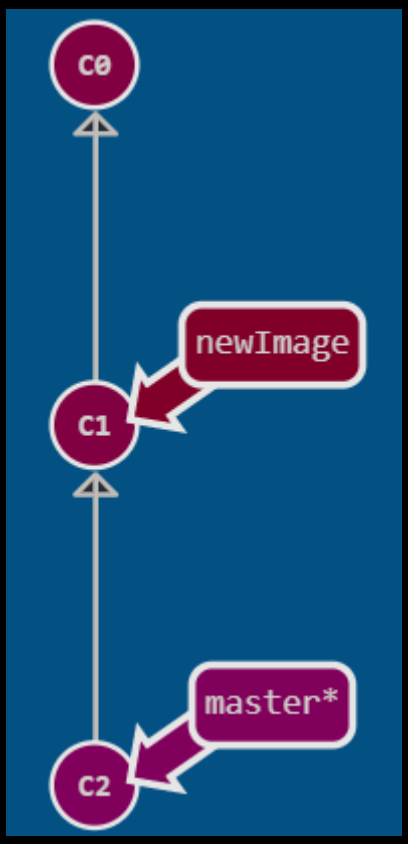
分支

Git 的分支也非常轻量。它们只是简单地指向某个提交纪录 —— 仅此而已。
所以：早建分支！多用分支！
即使创建再多分的支也不会造成储存或内存上的开销，并且按逻辑分解工作到不同的分支要比维护那些特别臃肿的分支简单多了。
分支其实就相当于在说："我想基于这个提交以及它所有的父提交进行新的工作。"



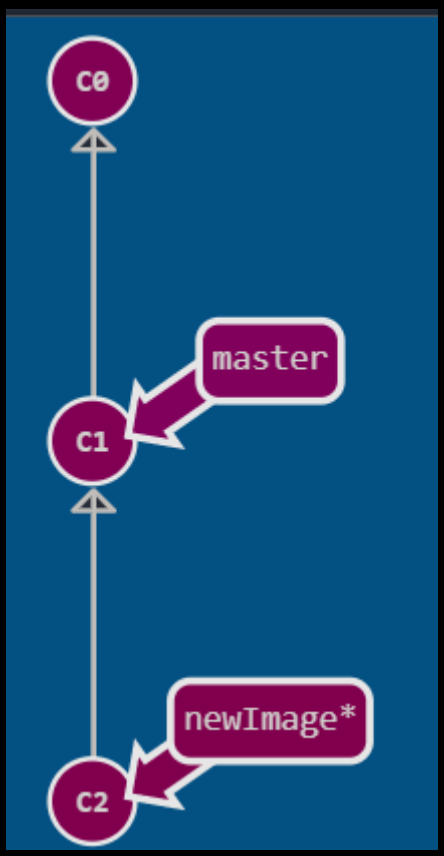
创建分支：git branch Name
创建的分支指向当前提交
带*号的表示当前提交到的分支，一般是 master

提交后只有带*号的会形成一个新的子提交



切换分支：

git checkout newImage



创建分支并切换过去：

git checkout -b <your-branch-name>

分支与合并

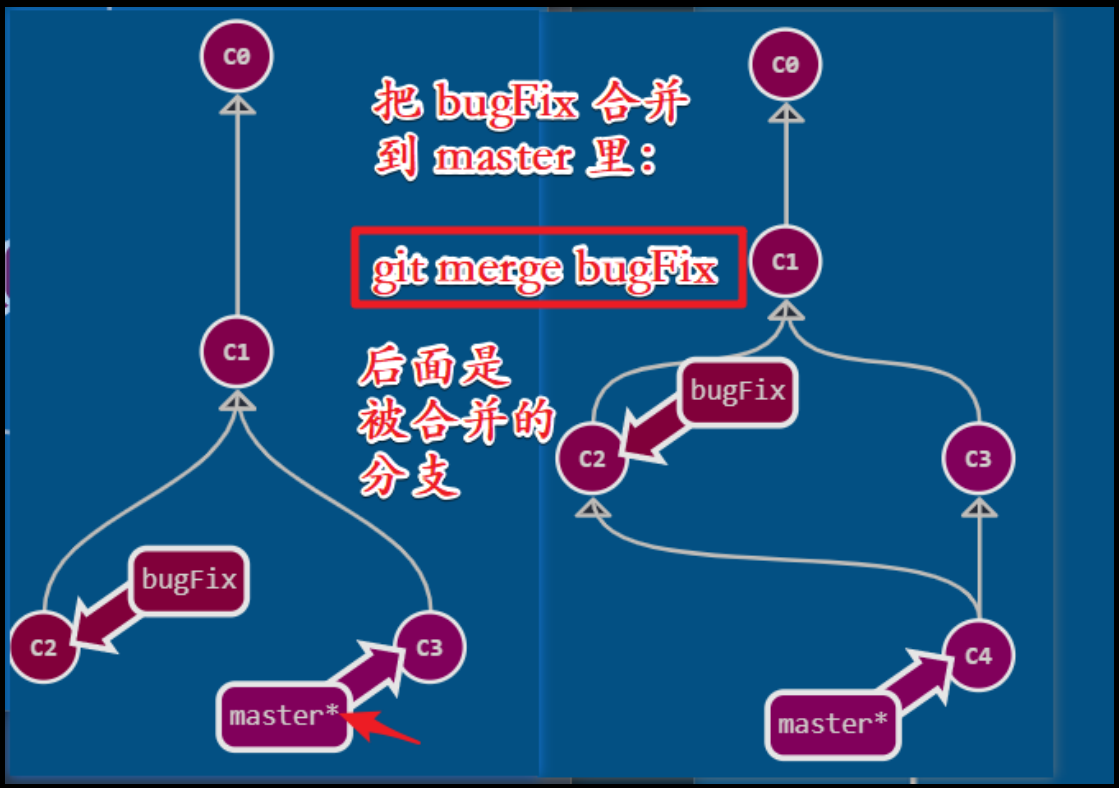
如何将两个分支合并到一起。就是说我们新建一个分支，在其上开发某个新功能，开发完成后再合并回主线。

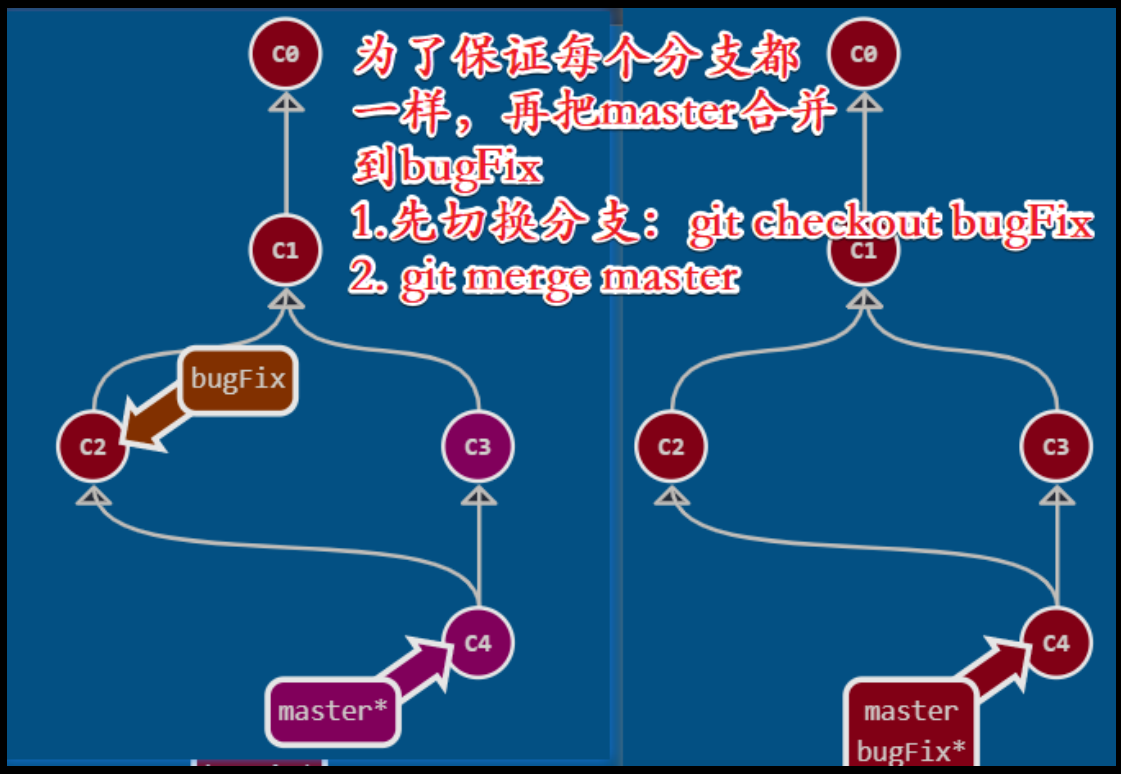
1. 方法一：

   git merge

   在 Git 中合并两个分支时会产生一个特殊的提交记录，它有两个父节点。翻译成自然语言相当于："我要把这两个父节点本身及它们所有的祖先都包含进来。"

   git-merge 命令是用于**从指定的 commit(s) 合并到当前分支的操作**。



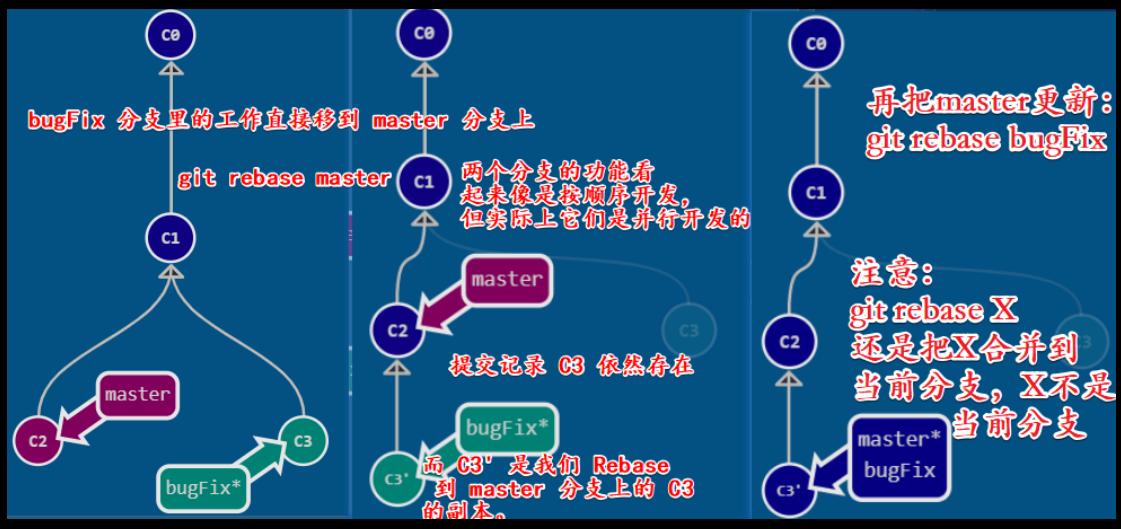


2. 方法二

   Git rebase

   Rebase 实际上就是取出一系列的提交记录，“复制”它们，然后在另外一个地方逐个的放下去。

   Rebase 的优势就是可以创造更线性的提交历史.如果只允许使用 Rebase 的话，代码库的提交历史将会变得异常清晰。



分支案例

- git branch br 创建分支 br
- git checkout br 切换分支
- git branch 查看分支

- git branch -a 查看远程分支
- git push origin master 本地仓库提交到远程仓库的 master 分支
- git push origin br 本地仓库提交到远程仓库的 br 分支
- git pull 从远程仓库的 master 分支 pull 到本地
- git pull origin master 同上
- git pull origin br 从远程仓库的 br 分支 pull 到本地
- git merge br 把 br 的最近的 commit 合并到当前分支
- git merge master 把 master 的最近的 commit 合并到当前分支

在提交树上移动

HEAD 是一个对当前检出记录的符号引用 —— 也就是指向你正在其基础上进行工作的提交记录。

HEAD 总是指向当前分支上最近一次提交记录。大多数修改提交树的 Git 命令都是从改变 HEAD 的指向开始的。
分离的 HEAD 就是让其指向了某个具体的提交记录而不是分支名。
通过 git checkout hash 值改变 head 的指向，

（git checkout 命令用于切换分支或恢复工作树文件，常用且危险的命令，因为这条命令会重写工作区）

但是很不方便，需要使用 git log 来查看提交记录
如 hash 值：fed2da64c0efc5293610bdd892f82a58e8cbc5d8
不过只需要指定可以区分其他提交的前几位就可以了，如 fed2

于是：出现**相对引用**

- 使用 ^ 向上移动 1 个提交记录：操作符 ^加在引用名称的后面：master^^ 是 master 的第二个父节点
- 使用 ~<num> 向上移动多个提交记录，如 ~3

可以直接使用 -f 选项让分支指向另一个提交。例如:
git branch -f master HEAD~3
上面的命令会将 master 分支强制指向 HEAD 的第 3 级父提交。

撤销变更

1. git reset

   git reset 通过把分支记录回退几个提交记录来实现撤销改动。你可以将这想象成“改写历史”。git reset 向上移动分支，原来指向的提交记录就跟从来没有提交过一样。

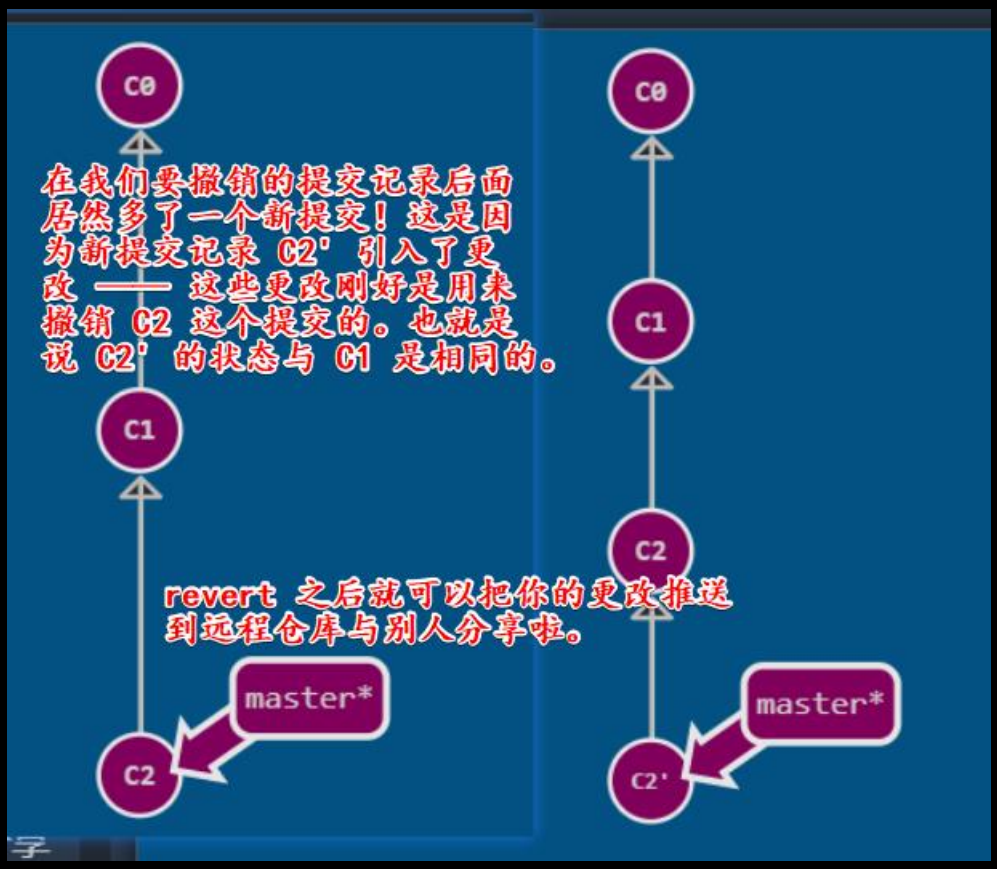
   Git rese HEAD~1

   回退到上一个版本

   注意：get reset 对远程分支是无效的

2. git revert

   撤销更改并**分享**给别人，可以用于远程仓库



   Git revert X：撤销的是 X，回到 X 的上一个

   而 get reset X: 是回到 X

整体提交

git cherry-pick

git cherry-pick <提交号>.

如果你想将一些提交复制到当前所在的位置（HEAD）下面的话， Cherry-pick 是最直接的方式

利用交互式的 rebase —— 如果你想从一系列的提交记录中找到想要的记录，这就是最好的方法了

交互式 rebase 指的是使用带参数 --interactive 的 rebase 命令，简写为 -i

如果你在命令后增加了这个选项，Git 会打开一个 UI 界面并列出将要被复制到目标分支的备选提交记录，它还会显示每个提交记录的哈希值和提交说明，提交说明有助于你理解这个提交进行了哪些更改。

当 rebase UI 界面打开时，你能做 3 件事：

- 调整提交记录的顺序（通过鼠标拖放来完成）
- 删除你不想要的提交（通过切换 pick 的状态来完成，关闭就意味着你不想要这个提交记录）
- 合并提交。

远程仓库

常见问题

CRLF

The file will have its original line endings in your working directory

Git 默认配置替换回车换行成统一的 CRLF，我们只需要修改配置禁用该功能即可。

Gitshell 中输入如下命令解决：

git config --global core.autocrlf false

The current branch master has no upstream branch

进行 git push 操作时报错：fatal: The current branch master has no upstream branch.
原因：**没有将本地的分支与远程仓库的分支进行关联**

解决
方式一
使用 git push --set-upstream origin master 命令

方式二
使用 git push -u origin master 命令

git 合并时冲突<<<<<<< HEAD

<<<<<<< HEAD
new code
=======
old code
>>>>>>> xxxxxxxxxxxxxxxxxxxxxxx

分析：head 到 ========里面的是自己的 commit 的内容
==========到 >>>>>>里面的是原来的内容
根据需要删除代码就行了
完事把<<<<<<< ======= >>>>>>>都删掉冲突就解决

```
# Branch Test
<<<<<<< HEAD
1. From master
=======
1. From Branch 1

>>>>>>> a9a161f5a86a0dd4d8622d2beacfd70c01c74b3f
```