Git

*表示当前指向的提交记录

git commit：提交记录

git branch <branch_name>：创建一个新的分支（但是指针仍然指向main，需要切换）

git checkout <branch_name>：切换到指定分支（即切换指针）

git checkout –b <branch_name> 哈希地址：在指定位置创建新分支，并切换指针到新分支

git switch

HEAD是一个指针，

直接checkout某个提交记录可以改变HEAD的指向（默认指向特定的分支名）

实际使用中，可以checkout某个提交记录的哈希值，来指向它

在 Git 中，origin 通常指的是 远程仓库的默认名称。当你使用 git clone 克隆一个远程仓库时，Git 会自动将该仓库命名为 origin，并将远程仓库的 URL 与之关联。

简单来说，origin 就是一个指向远程仓库的指针，方便你在本地进行操作时引用远程仓库。

- merge

  git merge bugFix：（未融合前指向main）将两个分支融合，指向新的main（但是只融合至了main，bugFix的指针仍然指向未融合的那个提交记录）

  git checkout bugFix

  git merge main：这样才让bugFix也融合了main，bugFix和main的指针都指向了新的提交记录

- rebase

  git rebase main：（当前指针在bugFix上）直接将bugFix添加到main，并创造一个新提交记录，看上去就像是线性提交一样，实际上是并行提交

  不过当前指针在新的bugFix上，需要把指针移动到main上，并git rebase bugFix，由于bugFix继承自main，所以 Git 只是简单的把 main 分支的引用向前移动了一下而已

  rebase也可以加一个参数–i，可以重新排序提交记录，也可以删除某些提交记录

  git rebase –i HEAD~4，表示重排之前的4个提交记录，会重新生成几个新的提交记录

  也可以添加参数：git rebase C1 C2：表示把指针移到C2，并rebase到C1

- 相对引用

  可以使用git log来查看提交记录的哈希值

  通过哈希值指定提交记录很不方便，所以 Git 引入了相对引用。

  使用相对引用的话，可以从一个易于记忆的地方（比如 bugFix 分支或 HEAD）开始计算。

  相对引用非常给力，这里介绍两个简单的用法：

  - 使用 ^ 向上移动 1 个提交记录
    - 操作符 (^)。把这个符号加在引用名称的后面，表示让 Git 寻找指定提交记录的 parent 提交。所以 main^ 相当于“main 的 parent 节点”。

      main^^ 是 main 的第二个 parent 节点

      也可以使用HEAD^，来表示当前提交记录往上走
  - 使用 ~<num> 向上移动多个提交记录，如 ~3，表示向上移动3次

  相对引用使用最多的就是移动分支。可以直接使用 –f 选项让分支指向另一个提交。例如:

  git branch –f main HEAD~3

  上面的命令会将 main 分支强制指向 HEAD 的第 3 级 parent 提交。

- reset

  git reset HEAD~1：撤销操作，向上撤销至指定位置（指针往上移动1位）

  git reset对远程分支无效，只能对本地进行操作，但是是彻底删除

- revert

  git revert

  要撤销的提交记录后面会多一个新提交，这是因为新提交记录 C2' 引入了更改 —— 这些更改刚好是用来撤销 C2 这个提交的。也就是说 C2' 的状态与 C1 是相同的。（同时也说明，原来的提交记录还在）

  revert 之后就可以把更改推送到远程仓库与别人分享了。

- cherry-pick

  git cherry-pick C1：把C1复制到HEAD所指的提交记录（可以填入多个参数以复制多个commit）

- clone

  git clone：把远程仓库复制一份到本地

  你可能注意到的第一个事就是在我们的本地仓库多了一个名为 o/main 的分支, 这种类型的分支就叫远程分支。由于远程分支的特性导致其拥有一些特殊属性。

  远程分支反映了远程仓库(在你上次和它通信时)的状态。这会有助于你理解本地的工作与公共工作的差别 —— 这是你与别人分享工作成果前至关重要的一步.

  在切换到远程分支时，自动进入分离 HEAD 状态（这意味着HEAD不再指向任何分支，而是直接指向一个特定的提交）。Git 这么做是出于不能直接在远程分支上进行操作的原因, 你必须在从远程分支的本地拷贝上完成你的工作，先在本地提交，再推送到远程仓库。

  远程分支有一个命名规范 —— 它们的格式是:

  - <remote name>/<branch name>

  因此，如果你看到一个名为 o/main 的分支，那么这个分支就叫 main，远程仓库的名称就是 o。

  大多数的开发人员会将它们主要的远程仓库命名为 origin，这是因为当你用 git clone 某个仓库时，Git 已经帮你把远程仓库的名称设置为 origin 了

- fetch

  git fetch：把远程仓库中，本地没有的提交下载到本地，同时更新远程分支指针origin/main

  远程分支反映了远程仓库在你最后一次与它通信时的状态，git fetch 就是你与远程仓库通信的方式。

  git fetch 通常通过互联网（使用 http:// 或 git:// 协议）与远程仓库通信。

  git fetch 并不会改变你本地仓库的状态。它不会更新你的 main 分支，也不会修改你磁盘上的文件。

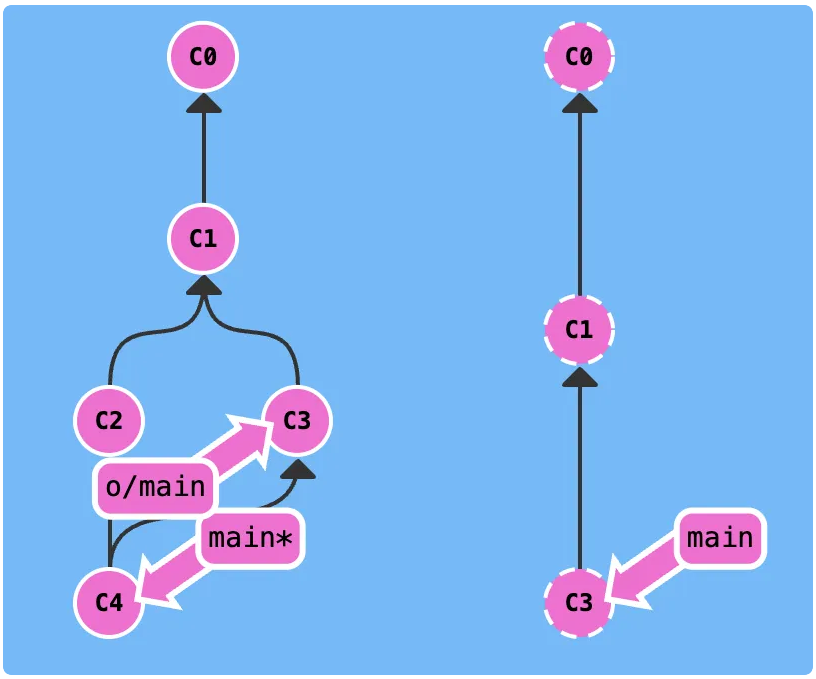
  理解这一点很重要，因为许多开发人员误以为执行了 git fetch 以后，他们本地仓库就与远程仓库同步了。它可能已经将进行这一操作所需的所有数据都下载了下来，但是并没有修改你本地的文件。

  所以, 你可以将 git fetch 的理解为单纯的下载操作。

- pull

  当远程分支中有新的提交时，你可以像合并本地分支那样来合并远程分支

  由于先抓取更新再合并到本地分支这个流程很常用，因此 Git 提供了一个专门的命令来完成这两个操作。它就是我们要讲的 git pull。



- push

  git push 负责将你的变更上传到指定的远程仓库，并在远程仓库上合并你的新提交记录。

  远程仓库接收了 C2，远程仓库中的 main 分支也被更新到指向 C2 了，我们的远程分支 (o/main) 也同样被更新了。所有的分支都同步了！

  git push 命令用于将本地分支的更新推送到远程仓库。其基本语法如下：

  ```
  Plain Text
  1  git push <远程主机名> <本地分支名>:<远程分支名>
  ```

  - <远程主机名>: 通常是 origin，但也可以是其他远程仓库的名称。
  - <本地分支名>: 你要推送的本地分支的名称。
  - <远程分支名>: 远程仓库中你要更新的分支的名称。

  所以，git push 后面跟的参数包括本地分支和远程分支。

  举例说明：

  ```
  Plain Text
  1  git push origin main:master
  ```

  这条命令会将本地的 main 分支推送到远程仓库 origin 的 master 分支。

  注意:

  - 如果省略 <远程分支名>，则默认推送到远程仓库中与本地分支同名的分支。
  - 如果远程分支不存在，则会在远程仓库中创建该分支。

- 远程分支偏离

  假设你周一克隆了一个仓库，然后开始研发某个新功能。到周五时，你新功能开发测试完毕，可以发布了。但是 —— 天啊！你的同事这周写了一堆代码，还改了许多你的功能中使用的 API，这些变动会导致你新开发的功能变得不可用。但是他们已经将那些提交推送到远程仓库了，因此你的工作就变成了基于项目旧版的代码，与远程仓库最新的代码不匹配了。

  这种情况下, git push 就不知道该如何操作了。如果你执行 git push，Git 应该让远程仓库回到星期一那天的状态吗？还是直接在新代码的基础上添加你的代码，亦或由于你的提交已经过时而直接忽略你的提交?

  因为这情况（历史偏离）有许多的不确定性，Git 是不会允许你 push 变更的。实际上它会强制你先合并远程最新的代码，然后才能分享你的工作。

  需要先git fetch，git rebase o/main，才能再git push（也可以使用merge，会把本地的所有变更也提交到远程仓库）

  或者简写git pull --rebase，再加上git push

- main锁定

  远程服务器拒绝直接推送(push)提交到main, 因为策略配置要求 pull requests 来提交更新.

  你应该按照流程,新建一个分支, 推送(push)这个分支并申请pull request,但是你忘记并直接提交给了main.现在你卡住并且无法推送你的更新.

  这时候就应当新建一个分支feature, 推送到远程服务器. 然后reset你的main分支和远程服务器保持一致, 否则下次你pull并且他人的提交和你冲突的时候就会有问题.

- remote add

  git remote add 是一个 Git 命令，用于将一个远程代码仓库关联到你的本地 Git 仓库。

  具体来说，它做了以下几件事：

  - 添加远程仓库信息： 它会将远程仓库的 URL 和名称存储在你的本地 Git 配置中。
  - 建立连接： 它允许你与远程仓库进行交互，例如推送 (push) 和拉取 (pull) 代码。
  - 方便操作： 你可以使用你指定的名称来代替远程仓库的 URL，简化后续的 Git 操作。

  例如：

  ```
  1  git remote add origin https://github.com/user/repo.git
  ```

  content_copyUse code with caution.

  这条命令将一个名为 "origin" 的远程仓库添加到你的本地仓库，该远程仓库的 URL 是 "https://github.com/user/repo.git"。之后，你就可以使用 "origin" 来指代这个远程仓库，例如：

  - git push origin master： 将本地 master 分支的代码推送到 origin 仓库的 master 分支。
  - git pull origin main： 从 origin 仓库的 main 分支拉取代码到本地。

- 本地仓库挪动位置

  当你创建了一个 Git 仓库后，如果需要移动整个文件夹到新的位置，你可以按照以下步骤操作：

  方法一：手动移动并更新配置

  - 关闭所有正在使用该仓库的程序，例如终端或代码编辑器。
  - 将整个仓库文件夹移动到新的位置。 你可以使用 Finder 或命令行工具 (例如 mv) 进行移动。
  - 进入移动后的仓库文件夹。
  - 更新 Git 配置中的工作目录路径。 使用以下命令：

```
1   git config --local core.worktree <新路径>
```

  content_copyUse code with caution.

  将 <新路径> 替换为仓库文件夹的新路径。

  - (可选) 更新远程仓库的 URL (如果适用) 如果你的仓库关联了远程仓库，你可能需要更新远程仓库的 URL，以反映新的路径。

  方法二：使用 git mv 命令 (仅限于 Git 仓库内部文件)

  如果你只想移动 Git 仓库内部的某些文件或文件夹到新的位置，可以使用 git mv 命令。例如：

```
1   git mv <旧路径> <新路径>
```

  重要提示：

  - 移动仓库文件夹后，请确保更新 Git 配置中的工作目录路径，否则 Git 将无法找到你的文件。
  - 如果你使用的是相对路径，例如在 .gitignore 文件中，你需要相应地更新这些路径。
  - 移动仓库文件夹可能会影响到一些 IDE 或 Git GUI 工具的设置，你可能需要手动更新这些设置。