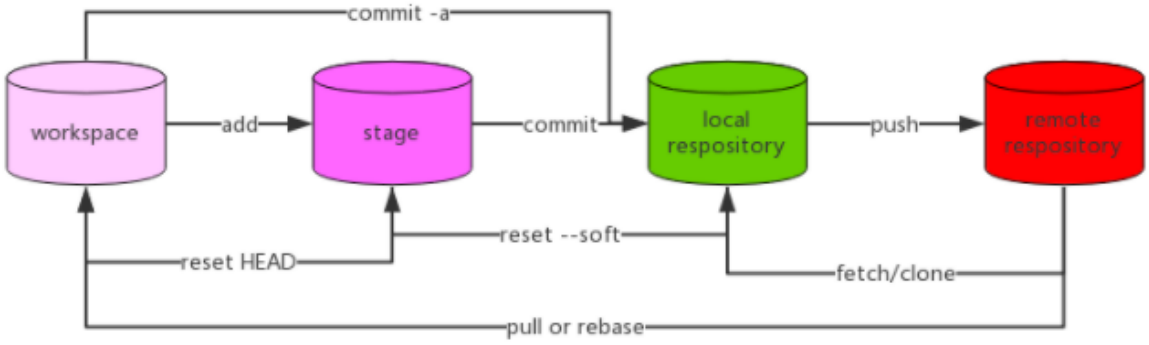
git 构成



workspace : 工作区，文件改动的区域

stage : git add 命令后保存修改的区域，存放在.git文件夹下的index文件中

local repository : 本地仓库，存放在.git文件夹

remote repository : 远程仓库

workspace 为文件区，stage与local repository为暂存区

本地 git

- git init

  创建.git子目录，保存版本信息

- git add

  add指令集成了保存对象和更新暂存区两个部分。

  一条指令能够完整地将修改后的文件进行对象的保存，并且载入stage暂存区。

- git commit

  git 仓库中的提交记录保存的是你的目录下所有文件的快照，就像是把整个目录复制，然后再粘贴一样，但比复制粘贴优雅许多。

  条件允许的情况下，它会将当前版本与仓库中的上一个版本进行对比，并把所有的差异打包到一起作为一个提交记录。

  Git 还保存了提交的历史记录，提交记录就是项目的快照，非常轻量，可以在提交记录之间来回切换。

- git branch

  Git 分支仅仅是指向某个提交记录，创建分支没有内存或者存储上的开销。现在只要记住使用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。”

  git commit 在当前的branch上提交记录

git branch A 在当前的提交记录上创建新的分支

git checkout B 从分支A切换到分支B

在不同的分支上进行checkout会产生提交记录的分支结构

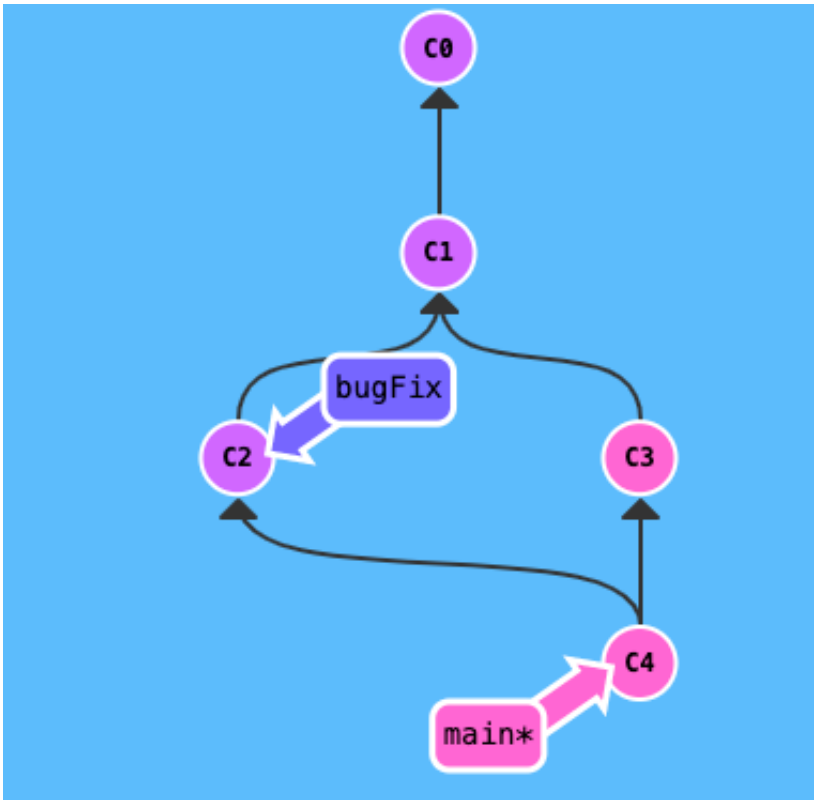
在一个提交记录上进行git branch不会产生提交记录的分支结构

- git merge (git merge foo,由当前分支下掉一个commit，当前分支指向下面一个commit，两叉分别指向foo与先前的commit，所以说把之前的commit和foo合并了)

  当有两个分支的时候，我们先开发一个新功能，再合并回main，产生一个全新的提交记录，完成新功能的merge。

  git checkout main

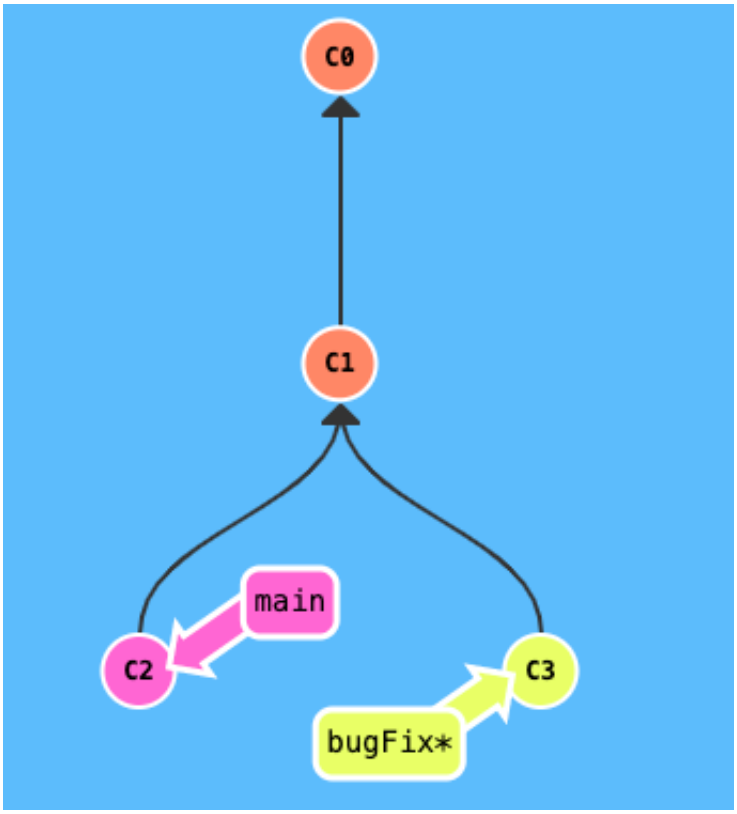
  git merge bugfix （merge后是main分支，包含了bugfix）



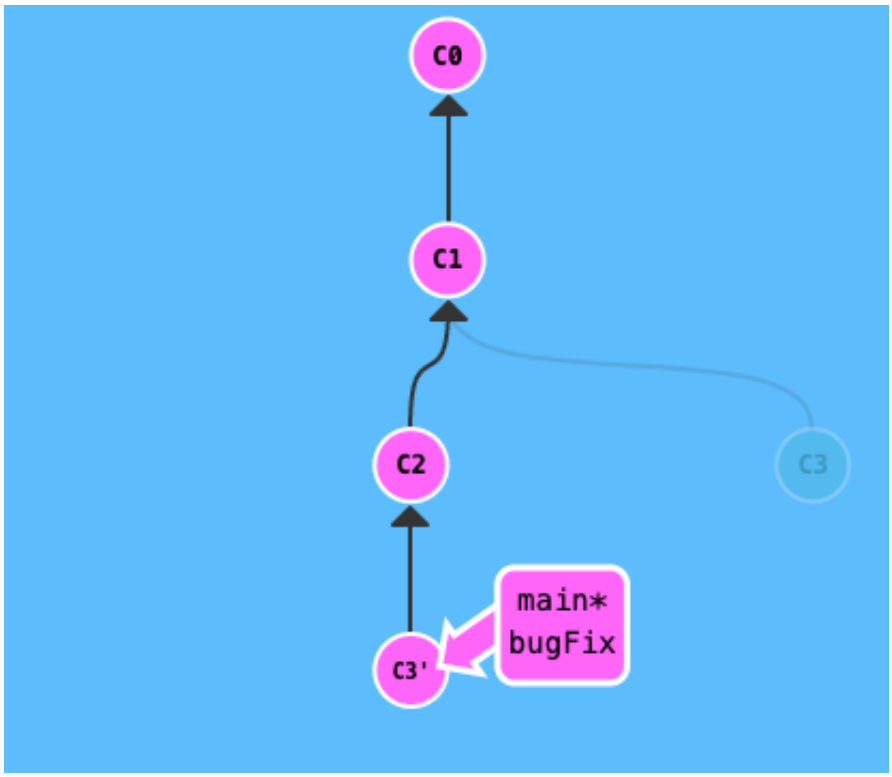
- git rebase (做commit的拷贝，形成线性的提交序列)

  当前在bugfix分支，使用rebase main 命令可以将main作为bugfix的父结点，形成线性的提交序列，分支节点依然存在，rebase仅仅完成了线性序列的复制

  从rebase的含义理解，git rebase main代表将当前的分支的父节点换为main



git rebase bugfix，因为bugfix是main的继承，所以git简单前移



故 git rebase A，若当前的分支为A的父节点，则直接将当前分支前移到A

若当前的分支为一个单独分支，则将当前分支(及其没有被rebase过的父节点)复制到A后面，成为A的子结点,故称为rebase

- 在提交树上移动HEAD （HEAD 可以指向提交记录，也可以指向分支名）

  git checkout C1 (以哈希值索引能够直接移动HEAD)

  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。

HEAD 通常情况下是指向分支名的（如 bugFix）。在你提交时，改变了 bugFix 的状态，这一变化通过 HEAD 变得可见。

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。

HEAD -> main -> C1 （其中HEAD指向main，main指向C1）

git checkout C1 转换为了 HEAD -> C1

- 相对引用 （直接移动HEAD [相对与绝对] ）

  通过指定提交记录哈希值的方式在 Git 中移动不太方便。在实际应用时，并没有像本程序中这么漂亮的可视化提交树供你参考，所以你就不得不用 `git log` 来查查看提交记录的哈希值。

  而且哈希值在 Git 世界中是基于SHA-1的40位。真正使用中只需要写出前几位即可。

  相对引用非常给力，这里我介绍两个简单的用法：

  - 使用 `^` 向上移动 1 个提交记录
  - 使用 `~<num>` 向上移动多个提交记录，如 `~3`

  作用是将 HEAD进行移动

  git checkout main^ 从main处上移一格 [相对]

  git checkout HEAD^ 上移一格 [相对]

  git checkout HEAD~3 [相对]

  git checkout fed3 为哈希值绝对引用 [绝对]

- 相对引用2 (根据相对位置移动main,HEAD等分支)

  git branch -f main HEAD~3 将main分支强制指向HEAD的第三级父提交

  移动的是main <-- head 而不仅仅是head

- 撤销变更

  - git reset HEAD~1（向上移动分支，原来指向的提交记录跟从来没提交过一样, reset 后面的参数代表退回到的目的地，**真正删掉了一个commit**）
  - git reset -soft "commit" 只回退local repo
  - git reset -hard "commit" 彻底回退到某个额版本
  - git revert HEAD（git reset 对远程分支无效，使用 revert 撤销更改并分享给别人, revert 后面的参数代表要revert的提交, revert 新增的commit正是当前HEAD的下一个子结点，**增加了一个删除动作的commit**）

    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。revert 之后就可以把你的更改推送到远程仓库与别人分享。

- Cherry-pick (绝对哈希访问的时候极为便利)

  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 是最直接的方式了。

  cherry-pick直接将想要复制的commit直接挂接在当前head下面

git cherry-pick C2 C4，C2' 与C4'直接会在HEAD下进行挂接

- 交互式的rebase (提交重新排序后复制)

  当无法使用哈希值进行绝对访问的时候，可以使用交互式的rebase

  git rebase -i HEAD~4 交互式的作用能在对话框中针对特定的commit进行复制，可以交换顺序也可以删除特定节点

- 本地栈式提交

  来看一个在开发中经常会遇到的情况：我正在解决某个特别棘手的 Bug，为了便于调试而在代码中添加了一些调试命令并向控制台打印了一些信息。这些调试和打印语句都在它们各自的提交记录里。最后我终于找到了造成这个 Bug 的根本原因，解决掉以后觉得沾沾自喜！最后就差把 `bugFix` 分支里的工作合并回 `main` 分支了。你可以选择通过 fast-forward 快速合并到 `main` 分支上，但这样的话 `main` 分支就会包含我这些调试语句了。

在我们bugFix过程中，我们多次产生commit，从而将最终的fixed commit并入main分支，完成我们的更新

- 提交的技巧

  在想要对某个中间版本进行代码修改时，根据先前的rebase信息进行改进

  先使用 git rebase -i 进行重拍和复制，构成临时的 commit 序列，把想要修改的部分改到最前端进行修改

  再在 commit 序列上进行修改 git commit --amend

  最后再对 commit 序列进行重拍和复制，和main一起rebase

- 提交的技巧2

  在想要对某个中间版本进行代码修改时，

  先使用 git cherry-pick 将要修改的部分单独拿出

  再在单独拿出的序列上进行修改 git commit --amend

  最后再用 cherry-pick 进行序列的还原

- Git Tags

  设置永远指向某个提交记录的标示，不能经过人为修改，与分支区别开来。

  Git Tag就是这个功能。

  git tag v1 C1 用 v1 永远地标记 C1

- Git Describe

  由于标签在代码库中起着"锚点"的作用，Git 还为此专门设计了一个命令用来**描述**离你最近的锚点（也就是标签），它就是 `git describe`！

  能帮你在提交历史中移动了多次以后找到方向

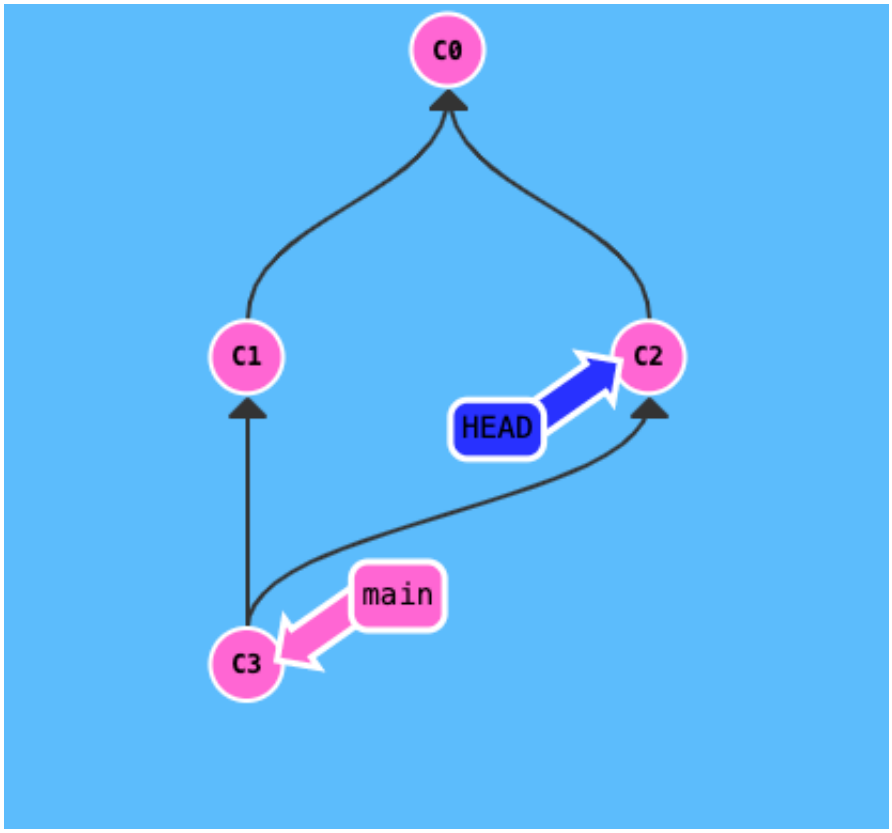
  针对特定的分支，git describe 可以找到对应最近的 tag。

- 多分支 rebase

  当我们所在的分支是A，想要添加到的线性序列是B时，git rebase B 会让B分支上所有的修改(没有被rebase过)复制到另一分支上。

它的原理是首先找到这两个分支（即当前分支 `experiment`、变基操作的目标基底分支 `master`）的最近共同祖先 `C2`，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件， 然后将当前分支指向目标基底 `C3`, 最后以此将之前另存为临时文件的修改依序应用。

- 两个父结点



git checkout main^2 会让HEAD移动到C2结点，即第二个父亲节点。

数字还可以进行组合，形成 git checkout HEAD~^2~2此类命令。

- 纠缠不清的分支

我们需要把 `main` 分支上最近的几次提交做不同的调整后，分别添加到各个的分支

基本思路 rebase / rebase -i / cherry-pick

1. Cherrry-pick 直接添加到该分支的下方，并且分支标记进行移动
2. rebase -i 需要有 rebase 历史才可进行重排
3. rebase 在寻找目标和当前HEAD的共同父节点之后进行复制，连同分支一起进行复制并移动

远程 git

- 远程仓库

远程仓库仅仅是自己仓库在另一台计算机上的拷贝。

可以通过因特网与这台计算机通信 —— 也就是增加或是获取提交记录

远程仓库是一个强大的备份。本地仓库也有恢复文件到指定版本的能力, 但所有的信息都是保存在本地的。有了远程仓库以后，即使丢失了本地所有数据, 你仍可以通过远程仓库拿回你丢失的数据。

远程让代码社交化了! 既然你的项目被托管到别的地方了, 你的朋友可以更容易地为你的项目做贡献(或者拉取最新的变更)

- git clone

  从技术上来讲，`git clone` 命令在真实的环境下的作用是在**本地**创建一个远程仓库的拷贝（比如从 github.com）。 但在我们的教程中使用这个命令会有一些不同 —— 它会在远程创建一个你本地仓库的副本。

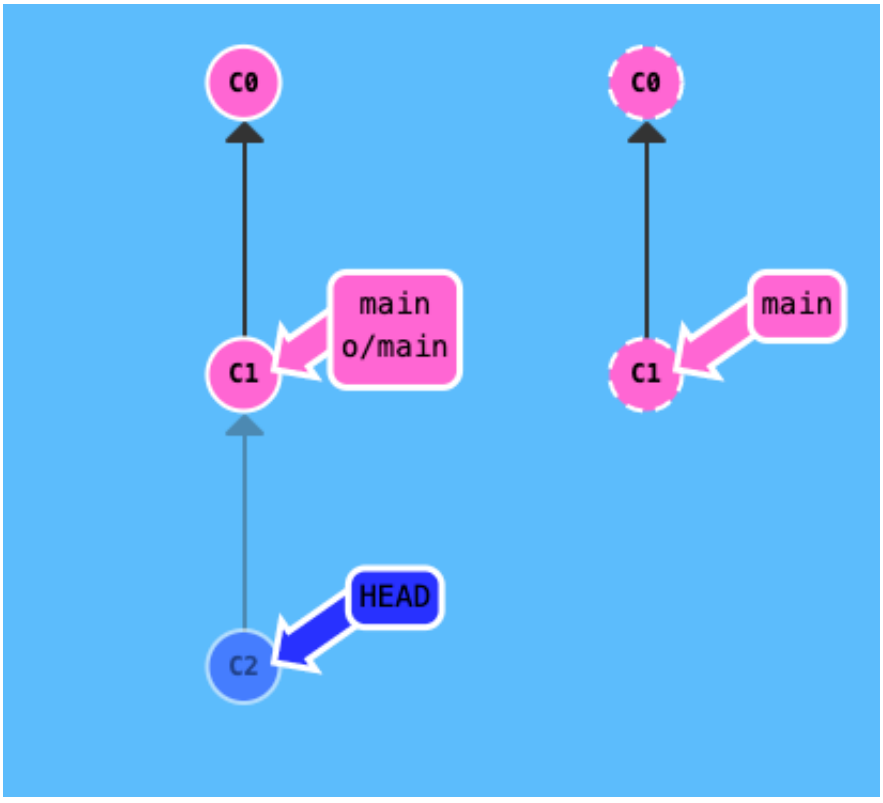
- 远程分支

  远程分支反映了远程仓库(在你上次和它通信时)的**状态**。

  远程分支有一个特别的属性，在你检出时自动进入分离 HEAD 状态。Git 这么做是出于不能直接在这些分支上进行操作的原因, 你必须在别的地方完成你的工作, （更新了远程分支之后）再用远程分享你的工作成果。**远程更新只是反映了远程的状态, HEAD 与 分支没有HEAD->分支的关系**

  很好理解，因为这里的 o/xxx并不是真实的，而是影子，所以不能用HEAD指向他们

  在commit之后自动进入HEAD分离状态

  你可能想问这些远程分支的前面的 `o/` 是什么意思呢？好吧, 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`。

  但是要记住, 当你使用真正的 Git 时, 你的远程仓库默认为 `origin`!

- git fetch （用于更新本地的远程分支[o/main]）

  当我们从远程仓库获取数据时, 远程分支也会更新以反映最新的远程仓库。

o/main 也会与远程的main一同更新

`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。

获得所有新的远程跟踪分支和标记，作为o/main放在本地，但不会更改合并到自己的分支中

- git pull

  其实有很多方法的 —— 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。也就是说就是你可以执行以下命令:

  - `git cherry-pick o/main`
  - `git rebase o/main`
  - `git merge o/main`
  - 等等

  Fetch + merge 较为常用，因此提供了专门的命令---pull

  Git pull 做到了获得远程仓库的新branch (fetch获得o/main)，再进行与本地的main进行merge，完成自己文件和远程仓库的更新

- 团队合作

  在别人修改了远程仓库之后我使用

- git push

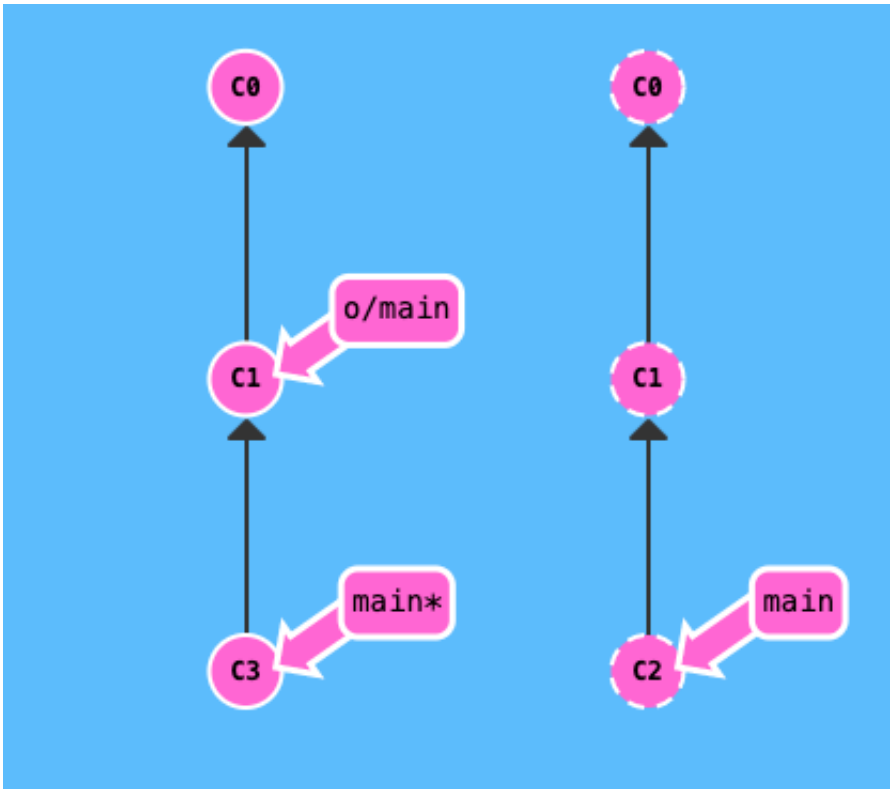
  `git push` 负责将**你的**变更上传到指定的远程仓库，并在远程仓库上合并你的新提交记录。一旦 `git push` 完成, 你的朋友们就可以从这个远程仓库下载你分享的成果了!

  `git push` 不带任何参数时的行为与 Git 的一个名为 `push.default` 的配置有关。它的默认值取决于你正使用的 Git 的版本，但是在教程中我们使用的是 `upstream`。 这没什么太大的影响，但是在你的项目中进行推送之前，最好检查一下这个配置。

  在git push之后，我本地名叫 origin 的远程分支也会随之更新

- 偏离的工作 (如果main的位置和远程仓库相比没有区别，则会显示已经最新，不需要push，如果远程仓库被修改过之后就无法进行push,必须要merge/rebase远程最新的代码 [**远程分支是本地分支的祖先，我的提交包含了远程分支的所有变化, o/main 与远程完全一致，main与远程不一致**] 才能够push)

  当远程仓库被修改过了之后，无法进行git push，在远程仓库被修改的情况下，git是不会允许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。



为了能够成功push，必须要是用rebase完成本地和远程的一致，随后再进行push才能够成功。

还有其它的方法可以在远程仓库变更了以后更新我的工作吗? 当然有，我们还可以使用 `merge`。尽管 `git merge` 不会移动你的工作（它会创建新的合并提交），但是它会告诉 Git 你已经合并了远程仓库的所有变更。这是因为远程分支现在是你本地分支的祖先，也就是说你的提交已经包含了远程分支的所有变化。

我们用 `git fetch` 更新了本地仓库中的远程分支，然后**合并**了新变更到我们的本地分支（为了包含远程仓库的变更），最后我们用 `git push` 把工作推送到远程仓库

前面已经介绍过 `git pull` 就是 fetch 和 merge 的简写，类似的 `git pull --rebase` 就是 fetch 和 rebase 的简写！

为了解决我们在落后的本地上开发的问题，使用 git pull + git merge (git pull --rebase) + git push 的写法

```
1. git clone
2. git fakeTeamword（远程的main改变，o/main与远程main的等价关系被打破）
3. git commit （此时 o/main 与 远程main不一致， main 与远程main不一致）
4. git pull --rebase == git pull + git merge （完成不一致性的同步，完成o/main 与远程main的等价关系，**远程是本地的祖先**，满足push条件）
5. git push  （origin main）（远程仓库中远程main被更新为本地main，本地o/main也更新，从而与main保持一致，**远程与本地一致**）
6. git push origin feature （在**远程是本地的祖先的条件下**，在远程仓库添加一个feature，对应的本地添加一个o/feature，保证远程与本地一致）
7. git reset o/main （在本地完成main与o/main的强制统一）
```

- 远程服务器拒绝

  很可能是master被锁定了, 需要一些Pull Request流程来合并修改。如果你直接提交(commit)到本地master, 然后试图推送(push)修改, 你将会收到这样类似的信息:

  ```
  ! [远程服务器拒绝] main -> main (TF402455: 不允许推送(push)这个分支; 你必须使用pull request来更新这个分支.)
  ```

  远程服务器拒绝直接推送(push)提交到master, 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master.现在你卡住并且无法推送你的更新.

  解决办法:

  新建一个分支feature, 推送到远程服务器. 然后reset你的master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.

- 合并特性分支

  最快将更新的main分支推送到远程 -----

  1.git pull --rebase

  2.git的相关操作 git rebase / git merge

  3.git push

- 远程跟踪分支
  - pull 操作时, 提交记录会被先下载到 o/main 上，之后再合并到本地的 main 分支。隐含的合并目标由这个关联确定的。
  - push 操作时, 我们把工作从 `main` 推到远程仓库中的 `main` 分支(同时会更新远程分支 `o/main`) 。这个推送的目的地也是由这种关联确定的!

  直接了当地讲， `main` 和 `o/main` 的关联关系就是由分支的“remote tracking”属性决定的。`main` 被设定为跟踪 `o/main` —— 这意味着为 `main` 分支指定了推送的目的地以及拉取后合并的目标。

  你可能想知道 `main` 分支上这个属性是怎么被设定的，你并没有用任何命令指定过这个属性呀！好吧, 当你克隆仓库的时候, Git 就自动帮你把这个属性设置好了。

  当你克隆时, Git 会为远程仓库中的每个分支在本地仓库中创建一个远程分支（比如 `o/main`）。然后再创建一个跟踪远程仓库中活动分支的本地分支，默认情况下这个本地分支会被命名为 `main`。

  克隆完成后，你会得到一个本地分支（如果没有这个本地分支的话，你的目录就是“空白”的），但是可以查看远程仓库中所有的分支（如果你好奇心很强的话）。这样做对于本地仓库和远程仓库来说，都是最佳选择。

  git clone结束，有一个本地分支，默认是main，有所有远程分支o/main,o/feature等，任意分支都可以跟踪o/main

  跟踪分支:

  1.在pull时与跟踪的o/main进程保持同步

  2.在push时，o/main移动到该跟踪进程处

  设置跟踪分支的两种方法:

  1.git checkout -b foo o/main

2.git branch foo; git branch -u o/main foo

- git push 参数

  git push

  git push origin main 的含义是切到本地仓库中的“main”分支，获取所有的提交，再到远程仓库“origin”中找到“main”分支，将远程仓库中没有的提交记录都添加上去，搞定之后告诉我。

  我们通过“place”参数来告诉 Git 提交记录来自于 main, 要推送到远程仓库中的 main。它实际就是要同步的两个仓库的位置。

  通过“origin”参数来表达远程仓库的名称。

简单理解 git push origin 就是把origin远程仓库上的main做到与本地同步 (本地即移动o/xxx与xxx重叠)

git push在不加参数的情况下，默认是HEAD跟踪的分支，如果HEAD没有指向分支则报错

- git push 参数2

  当为 git push 指定 place 参数为 `main` 时，我们同时指定了提交记录的来源和去向。

  你可能想问 —— 如果来源和去向分支的名称不同呢？比如你想把本地的 `foo` 分支推送到远程仓库中的 `bar` 分支。

  git push origin :

  git push origin foo:bar

  把本地的foo做基准，远端bar与本地foo同步，即o/bar与foo重叠

- git fetch参数

  git fetch 与 git push概念相同，方向相反

  git fetch origin foo 效果：xxx移动本地 o/xxx 位置与远程 xxx 相同，本地的 xxx 与远程 xxx 不一致，因为没有merge仅仅是fetch

  git fetch origin foo:bar 效果：远程foo与本地bar重合，o/xxx不变

  git fetch 自动简历 远程 xxx 对应的 本地o/xxx

- 古怪的

  source可以留空

  git push origin :foo 删除了本地的o/foo和远程的foo

  git fetch origin :bar 创建了本地的bar

- git pull参数

  fetch + merge

  git pull origin main : 本地o/main 与 远程main 位置相同，o/main再被merge

  git pull origin main:foo ： 本地 foo 与 远程main位置相同，foo再被merge (本地的分支会自动创建)

重点：

- fetch和pull的区别

  fetch只是将远程仓库作为origin分支加入本地，pull将origin加入本地之后进行了合并

  fetch不会动本地的main，只是添加了o/main，pull会通过merge改动本地的main
- rebase和merge的区别
- git push时刻

  o/main 与 本地仓库 main 不一致，并且 远程分支是本地分支的祖先时可以成功git push

  若远程分支不是本地分支的祖先，则push报错

  若master被PR封锁，则push报错

  若o/main 与main一致且没有其他修改时，则报已经是最新的信息