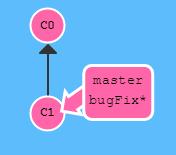
**Git命令大全，学习网址：https://learngitbranching.js.org/?demo**

**Git branch：**

Git 的分支也非常轻量。它们只是简单地指向某个提交纪录。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 branch bugFix

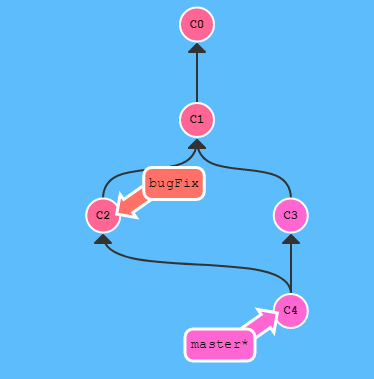
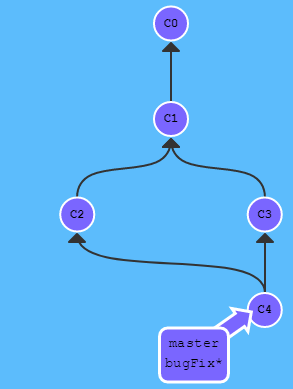


**Git merge：**

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。”

图一为 Git merge bugFix (所在分支为master)

图二为 git merge master (所在分支为 bugFix)

**Git rebase:**

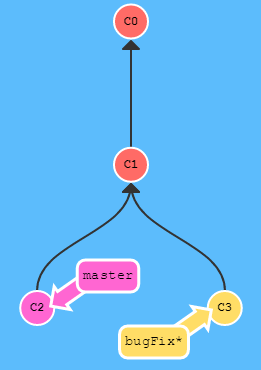
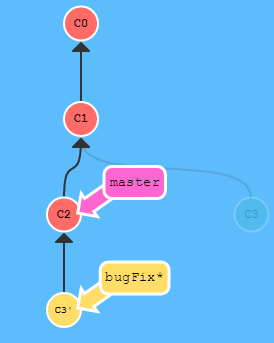
Rebase 实际上就是取出一系列的提交记录，“复制”它们，然后在另外一个地方逐个的放下去。Rebase 的优势就是可以创造更线性的提交历史，这听上去有些难以理解。如果只允许使用 Rebase 的话，代码库的提交历史将会变得异常清晰。

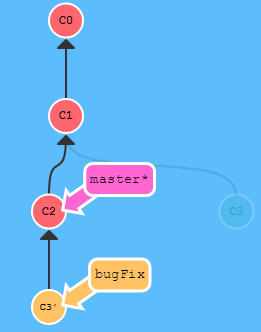
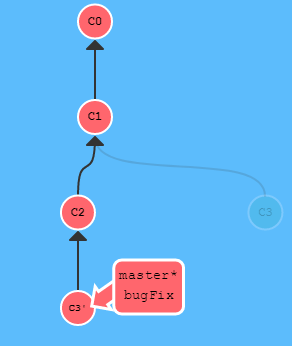
我们想要把 bugFix 分支里的工作直接移到 master 分支上。移动以后会使得两个分支的功能看起来像是按顺序开发，但实际上它们是并行开发的。

Git rebase master

Git checkout master

Git rebase bugFix

**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。在命令执行之前的状态如下所示：

HEAD -> master -> C1

HEAD 指向 master， master 指向 C1

我们可以用哈希值来控制HEAD的指向，比如git checkout fed2da64c0efc5293610bdd892f82a58e8cbc5d8

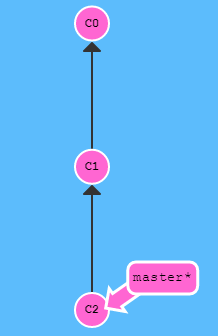
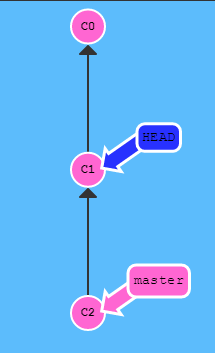
比较令人欣慰的是，Git 对哈希的处理很智能。你只需要提供能够唯一标识提交记录的前几个字符即可。因此我可以仅输入fed2 而不是上面的一长串字符。

相对引用：

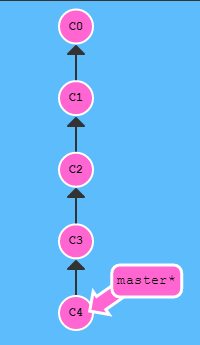
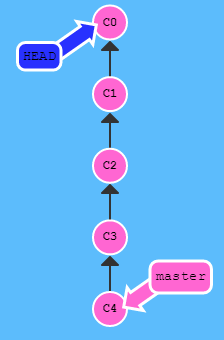
使用 ^ 向上移动 1 个提交记录

使用 ~<num> 向上移动多个提交记录，如 ~3

Git checkout master^

Git checkout master~4

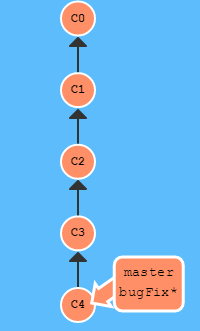
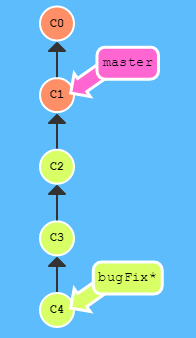
**强制修改分支位置**

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。

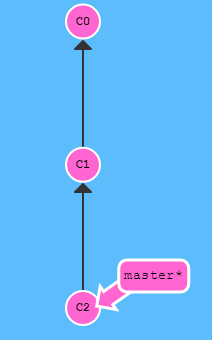
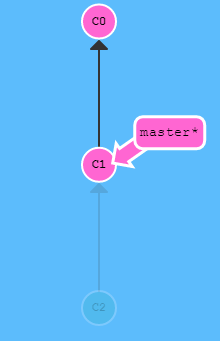
 

**撤销变更**

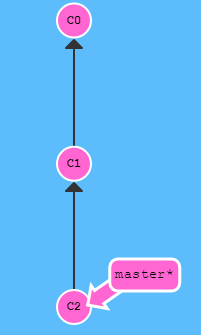
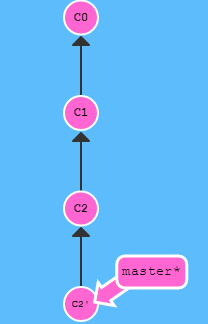
在 Git 里撤销变更的方法很多。和提交一样，撤销变更由底层部分（暂存区的独立文件或者片段）和上层部分（变更到底是通过哪咱方式被撤销的）组成

主要有两种方法用来撤销变更 —— 一是 git reset，还有就是 git revert。

Git reset HEAD~1（这种方法主要用于本地使用）

Git revert HEAD（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啦。）

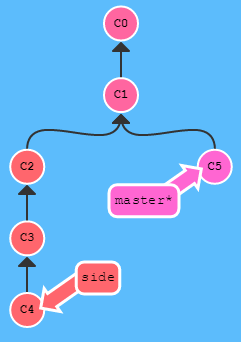
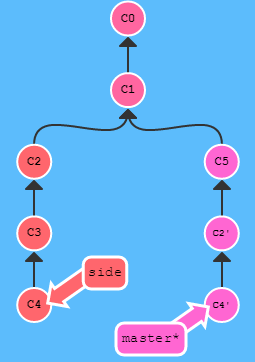
**Git cherry-pick**

命令形式为:

git cherry-pick <提交号>...

这里有一个仓库, 我们想将 side 分支上的工作复制到 master 分支

Git cherry-pick C2 C4

**交互式的rebase**

当你知道你所需要的提交记录（**并且**还知道这些提交记录的哈希值）时, 用 cherry-pick 再好不过了 —— 没有比这更简单的方式了。

但是如果你不清楚你想要的提交记录的哈希值呢? 幸好 Git 帮你想到了这一点, 我们可以利用交互式的 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。

在实际使用时，所谓的 UI 窗口一般会在文本编辑器 —— 如 Vim —— 中打开一个文件。

在 Vim 中，选择对应的提交记录，完成后会在那个节点下面依次排序展开，效果如 cherry-pick 一样。

Git rebase –i + <对应的分支名字>

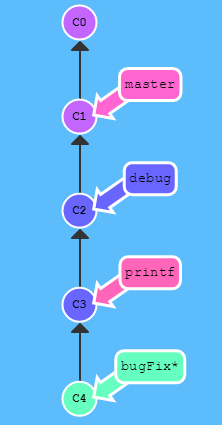
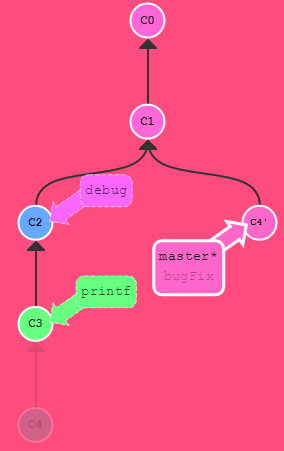
练习1：

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 master 分支了。

Git checkout master

Git cherry-pick bugFix

练习2：

你之前在 newImage 分支上进行了一次提交，然后又基于它创建了 caption 分支，然后又提交了一次。

此时你想对的某个以前的提交记录进行一些小小的调整。比如设计师想修改一下 newImage 中图片的分辨率，尽管那个提交记录并不是最新的了。

我们可以通过下面的方法来克服困难：

先用 git rebase -i 将提交重新排序，然后把我们想要修改的提交记录挪到最前

然后用 commit --amend 来进行一些小修改

接着再用 git rebase -i 来将他们调回原来的顺序

最后我们把 master 移到修改的最前端（用你自己喜欢的方法），就大功告成啦！

当然完成这个任务的方法不止上面提到的一种（我知道你在看 cherry-pick 啦），之后我们会多点关注这些技巧啦，但现在暂时只专注上面这种方法。 最后有必要说明一下目标状态中的那几个' —— 我们把这个提交移动了两次，每移动一次会产生一个 '；而 C2 上多出来的那个是我们在使用了 amend 参数提交时产生的，所以最终结果就是这样了。

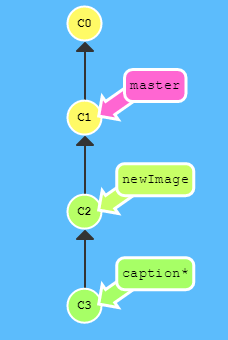
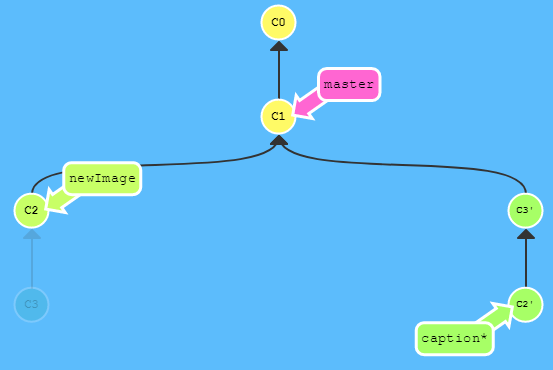
Git rebase -I HEAD~2 （重新排序）

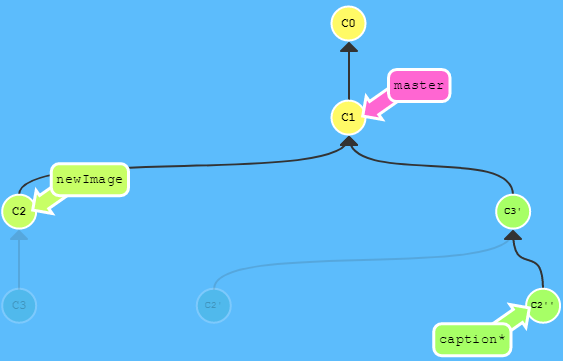
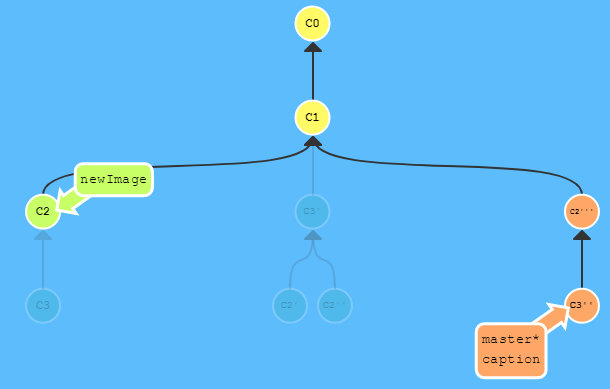
Git commit --amend

Git rebase -I HEAD~2

Git checkout master

Git rebase caption

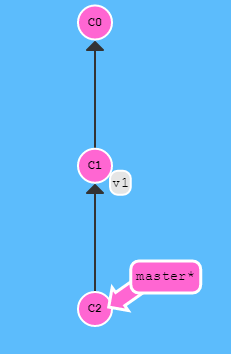
 

Git tag

分支很容易被人为移动，并且当有新的提交时，它也会移动。分支很容易被改变，大部分分支还只是临时的，并且还一直在变。

有没有什么可以*永远*指向某个提交记录的标识呢，比如软件发布新的大版本，或者是修正一些重要的 Bug 或是增加了某些新特性，有没有比分支更好的可以永远指向这些提交的方法呢？

Git tag v1 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.

Git Describe 能帮你在提交历史中移动了多次以后找到方向；当你用 git bisect（一个查找产生 Bug 的提交记录的指令）找到某个提交记录时，或者是当你坐在你那刚刚度假回来的同事的电脑前时， 可能会用到这个命令。

git describe <ref>

<ref> 可以是任何能被 Git 识别成提交记录的引用，如果你没有指定的话，Git 会以你目前所检出的位置（HEAD）。

它输出的结果是这样的：

<tag>\_<numCommits>\_g<hash>

tag 表示的是离 ref 最近的标签， numCommits 是表示这个 ref 与 tag 相差有多少个提交记录， hash 表示的是你所给定的 ref 所表示的提交记录哈希值的前几位。

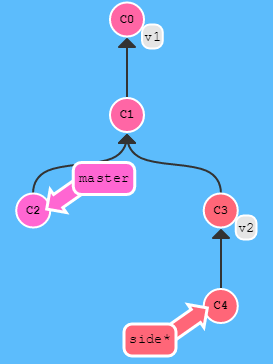
当 ref 提交记录上有某个标签时，则只输出标签名称。

git describe master 会输出：

v1\_2\_gC2

git describe side 会输出：

v2\_1\_gC4



**远程仓库**

远程仓库并不复杂, 在如今的云计算盛行的世界很容易把远程仓库想象成一个富有魔力的东西, 但实际上它们只是你的仓库在另个一台计算机上的拷贝。你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

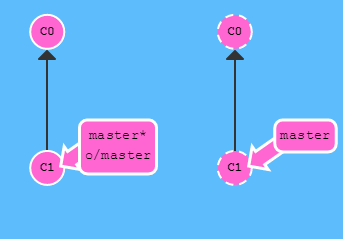
话虽如此, 远程仓库却有一系列强大的特性

首先也是最重要的的点, 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。

还有就是, 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)

现在用网站来对远程仓库进行可视化操作变得越发流行了(像 [Github](https://github.com/) 或 [Phabricator](http://phabricator.org/)), 但远程仓库**永远**是这些工具的顶梁柱, 因此理解其概念非常的重要!

Git clone



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>

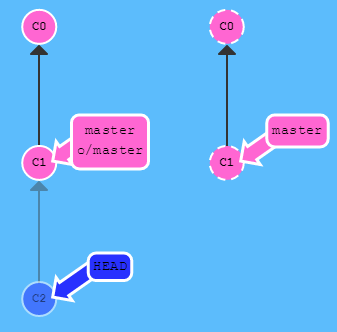
因此，如果你看到一个名为 o/master 的分支，那么这个分支就叫 master，远程仓库的名称就是 o。

大多数的开发人员会将它们主要的远程仓库命名为 origin，并不是 o。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

不过 origin 对于我们的 UI 来说太长了，因此不得不使用简写 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 checkout o/master

Git commit



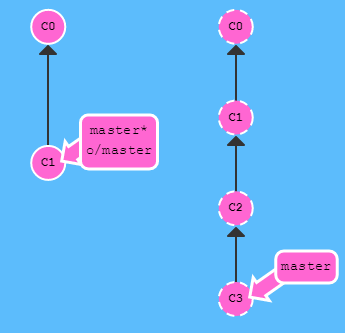
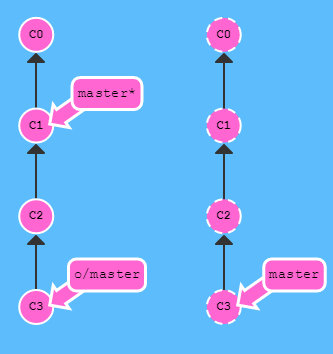
Git 变成了分离 HEAD 状态，当添加新的提交时 o/master 也不会更新。这是因为 o/master 只有在远程仓库中相应的分支更新了以后才会更新。

Git 远程仓库相当的操作实际可以归纳为两点：向远程仓库传输数据以及从远程仓库获取数据。既然我们能与远程仓库同步，那么就可以分享任何能被 Git 管理的更新（因此可以分享代码、文件、想法、情书等等）。

本节课我们将学习如何从远程仓库获取数据 —— 命令如其名，它就是 git fetch。

你会看到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

Git fetch

git fetch 完成了仅有的但是很重要的两步:

从远程仓库下载本地仓库中缺失的提交记录

更新远程分支指针(如 o/master)

git fetch 实际上将本地仓库中的远程分支更新成了远程仓库相应分支最新的状态。

如果你还记得上一节课程中我们说过的，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了！希望我说的够明白了，你已经了解 git fetch 与远程分支之间的关系了吧。

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 master 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。我们在后面的课程中将会讲解能完成该操作的命令 :D

所以, 你可以将 git fetch 的理解为单纯的下载操作。

既然我们已经知道了如何用 git fetch 获取远程的数据, 现在我们学习如何将这些变化更新到我们的工作当中。

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/master

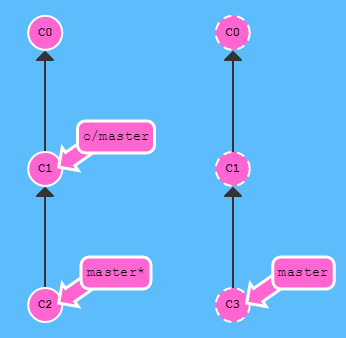
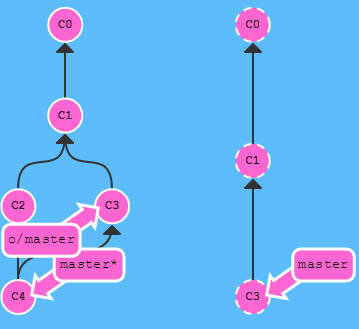
git rebase o/master

git merge o/master

等等

实际上，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

Git pull

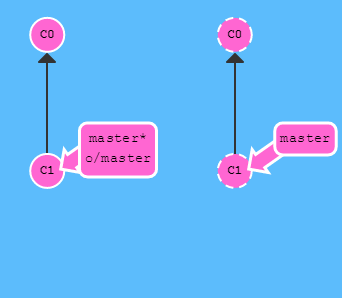
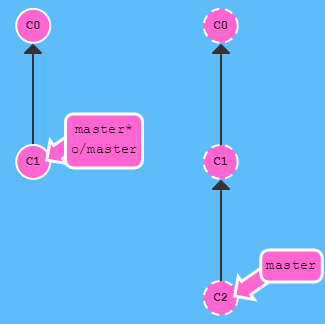
 

里有一件棘手的事 —— 为了接下来的课程, 我们需要先教你如何制造远程仓库的变更。

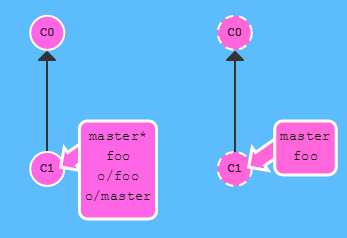
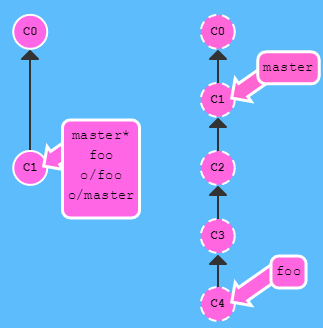
这意味着，我们需要“假装”你的同事、朋友、合作伙伴更新了远程仓库，有可能是某个特定的分支，或是几个提交记录。

为了做到这点，我们引入一个自造命令 git fakeTeamwork！它的名称已经说明了一切，先看演示..

Git fakeTeamwork

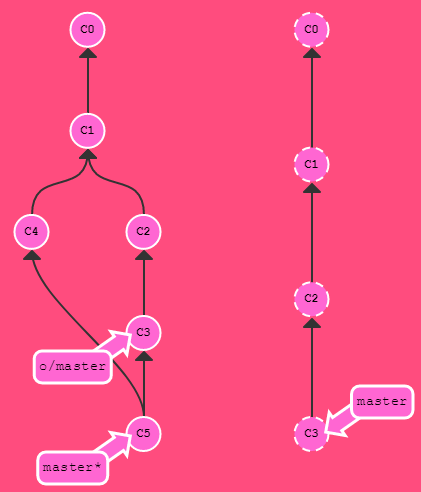
 

Git fakeTeamwork foo 3

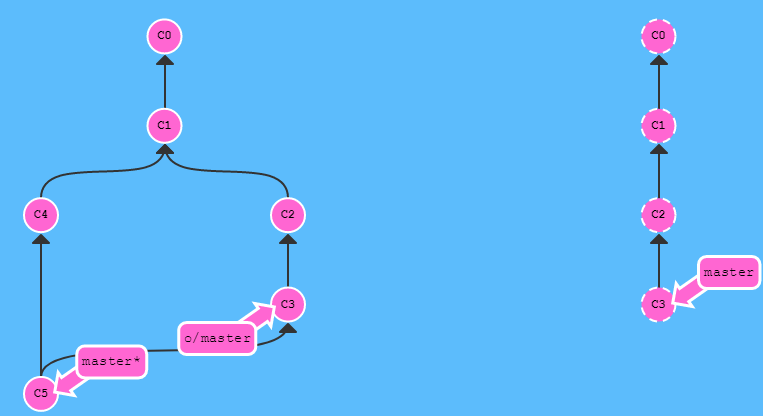
 

模拟团队练习

目标







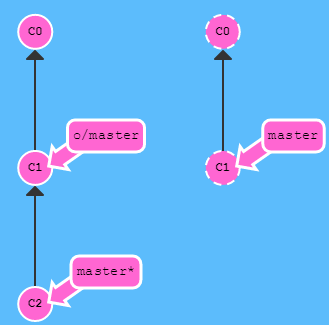
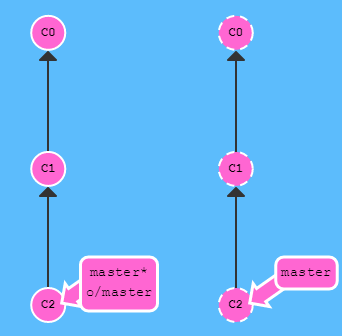
**Git push**

上传自己分享内容与下载他人的分享刚好相反，那与 git pull 相反的命令是什么呢？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！

你可以将 git push 想象成发布你成果的命令。

Git push

下图的过程

git commit

git commit

git push





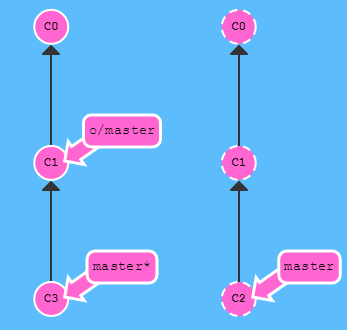
常见问题 历史偏离

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，异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。

比如这种情况下，git push 是无效的。git push 失败是因为你最新提交的 C3 基于远程分支中的 C1。而远程仓库中该分支已经更新到 C2 了，所以 Git 拒绝了你的推送请求。

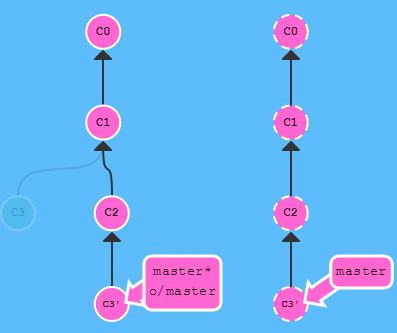


解决方法：

Git fetch

Git rebase o/master

Git push



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。

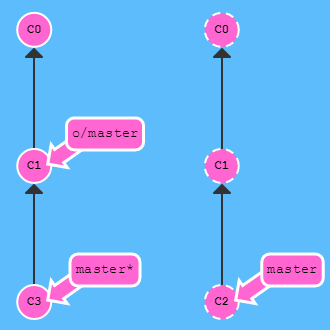
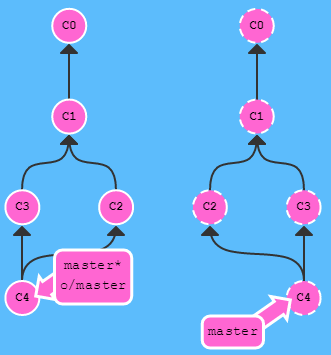
还有其它的方法可以在远程仓库变更了以后更新我的工作吗? 当然有，我们还可以使用 merge

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。

Git fetch

Git merge o/master

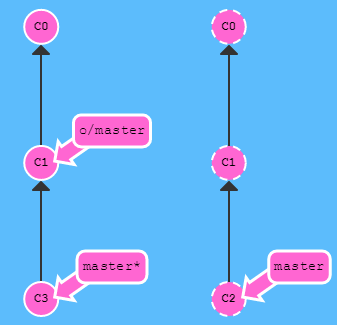
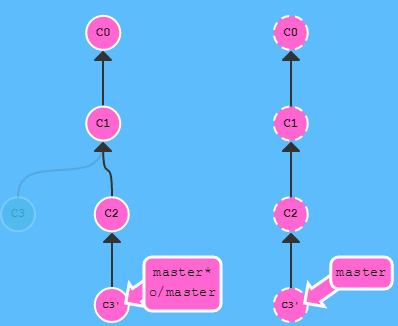
Git push

当然 —— 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！

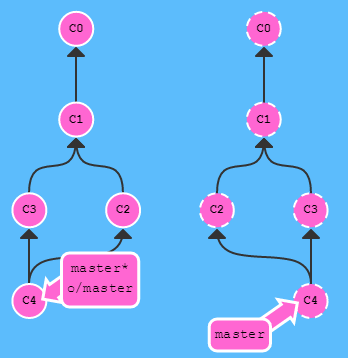
Git pull --rebase

Git push

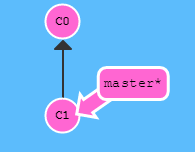
 

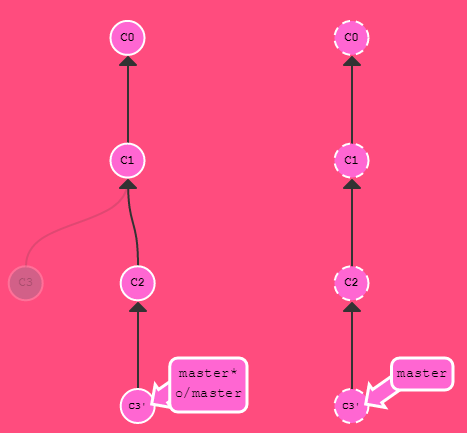
Git pull

Git push



练习：

变成下面的



git fakeTeamwork

git commit

git pull --rebase

git push

#查看本地都有哪些分支

git branch

#查看所有的分支，包括远程的

git branch -a

#删除本地的某个分支

git branch -D hongchangfirst

#删除远程的分支

git branch -r -d origin/hongchangfirst

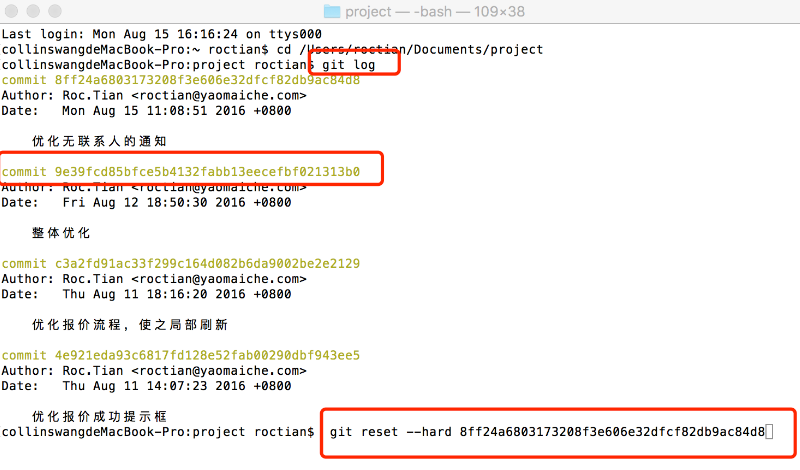
#注意这个只是删除本地的索引，而不是真正删除远程分支的内容，要想真正删除远程分支上的内容，可以这样：

git push origin :hongchangfirst

回退到某个分支：

git log

git reset –hard 8ff24a6803173208f3e606e32dfcf82db9ac84d8



git reset --hard HEAD~1 # 回退到上一次的提交, 如果是上n次就将1改成对应的数字

git reset --soft commitID 只是删除了commitId之后的commit记录,但是代码改动仍然存在

git reset --hard commitID 彻底的回到CommitID13:39时候的版本,之后的改动不存在了