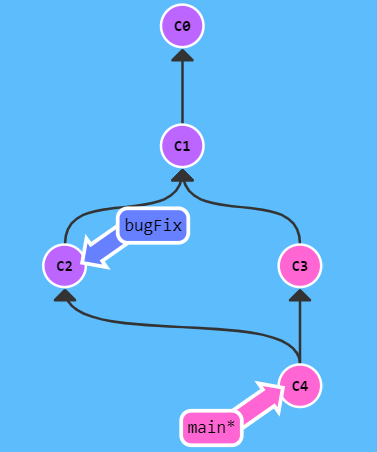
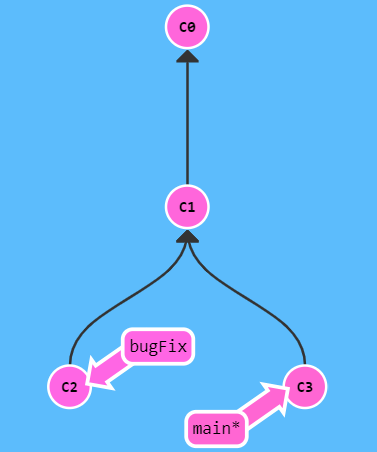
提交记录 git commit

创建新分支 git branch XXX

切换新分支 git checkout XXX switch代替checkout

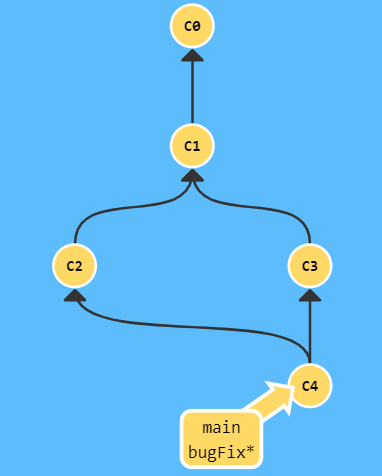
创建一个新的分支同时切换到新创建的分支 git checkout -b <your-branch-name>

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 第一种方法 —— 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 parent 节点。翻译成自然语言相当于：“我要把这两个 parent 节点本身及它们所有的祖先都包含进来。” 我们要把 bugFix 合并到 main 里 git merge bugfix



再把 main 分支合并到 bugfix

git checkout bugfix; git merge main



因为 main 继承自 bugFix，Git 什么都不用做，只是简单地把 bugFix 移动到 main 所指向的那个提交记录。

现在所有提交记录的颜色都一样了，这表明每一个分支都包含了代码库的所有修改！大功告成！

创建新分支 bugFix

git branch bugFix

用 git checkout bugFix 命令切换到该分支

git checkout bugFix

提交一次

git commit

用 git checkout main 切换回 main

git checkout main

再提交一次

git commit

用 git merge 把 bugFix 合并到 main

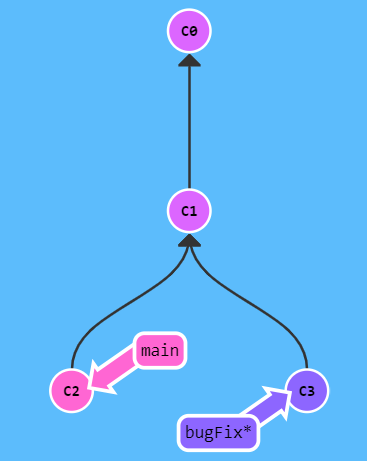
git merge bugFix

你随时都可以用“objective”命令来打开这个对话框！\*

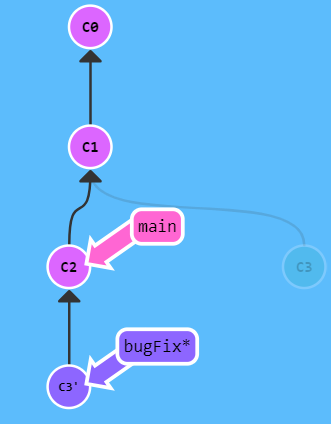
第二种合并分支的方法是 git rebase。Rebase 实际上就是取出一系列的提交记录，“复制”它们，然后在另外一个地方逐个的放下去。

我们想要把 bugFix 分支里的工作直接移到 main 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。

咱们这次用 git rebase 实现此目标

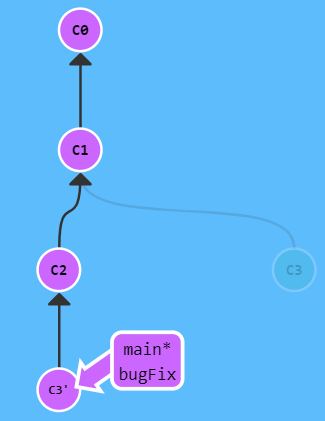


git rebase main



现在我们切换到了 main 上。把它 rebase 到 bugFix 分支上……

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。

新建并切换到 bugFix 分支

git checkout -b bugFix

提交一次

git commit

切换回 main 分支再提交一次

git checkout main; git commit

再次切换到 bugFix 分支，rebase 到 main 上

git checkout bugFix; git rebase main

HEAD 是一个对当前所在分支的符号引用 —— 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 -> main -> C1

git checkout c1

HEAD 指向 main， main 指向 C1

想完成此关，从 bugFix 分支中分离出 HEAD 并让其指向一个提交记录。

通过哈希值指定提交记录。每个提交记录的哈希值显示在代表提交记录的圆圈中。

git checkout c4

相对引用

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。

并且哈希值在真实的 Git 世界中也会更长（译者注：基于 SHA-1，共 40 位）。例如前一关的介绍中的提交记录的哈希值可能是 fed2da64c0efc5293610bdd892f82a58e8cbc5d8。舌头都快打结了吧...

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。

正如我前面所说，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。这个就很厉害了!

使用相对引用的话，你就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

首先看看操作符 (^)。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

现在咱们切换到 main 的 parent 节点

要完成此关，切换到 bugFix 的 parent 节点。这会进入分离 HEAD 状态。git checkout bugfix^

如果你愿意的话，使用哈希值也可以过关，但请尽量使用相对引用！

强制修改分支位置

你现在是相对引用的专家了，现在用它来做点实际事情。

我使用相对引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。

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（变更到底是通过哪种方式被撤销的）组成。我们这个应用主要关注的是后者。

主要有两种方法用来撤销变更 —— 一是 git reset，还有就是 git revert。接下来咱们逐个进行讲解。

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 Revert

虽然在你的本地分支中使用 git reset 很方便，但是这种“改写历史”的方法对大家一起使用的远程分支是无效的哦！

为了撤销更改并分享给别人，我们需要使用 git revert。奇怪！在我们要撤销的提交记录后面居然多了一个新提交！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。

revert 之后就可以把你的更改推送到远程仓库与别人分享啦。

完成此关，分别撤销 local 分支和 pushed 分支上的最近一次提交。共需要撤销两个提交（每个分支一个）。

git reset local

git revert pushed

记住 pushed 是远程分支，local 是本地分支 —— 这么说你应该知道用分别哪种方法了吧？

本系列的第一个命令是 git cherry-pick, 命令形式为:

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 是最直接的方式了。

git cherry-pick C2 C4我们只需要提交记录 C2 和 C4，所以 Git 就将被它们抓过来放到当前分支下了。

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。

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。 遗憾的是由于某种逻辑的原因，我们的课程不支持此功能，因此我不会详细介绍这个操作。简而言之，它允许你把多个提交记录合并成一个。

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。

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。你肯定不想这样，应该还有更好的方式……

实际我们只要让 Git 复制解决问题的那一个提交记录就可以了。跟之前我们在“整理提交记录”中学到的一样，我们可以使用

git rebase -i

git cherry-pick

来达到目的。

提交的技巧 #1

接下来这种情况也是很常见的：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 git 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 main 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。

也就是说，我在对比结果的时候只会对比提交树的结构，对于 ' 的数量上的不同，并不纳入对比范围内。只要你的 main 分支结构与目标结构相同，我就算你通过。

先经过一次复制，rebase -i 交互式复制得到C2’ C3’ 再经过一次commit amend 单独复制出C2’’ 再经过一次交互式复制，就得到C2’’’ C3’’了

先移动分支，-f 选项让分支A指向另一个提交B git branch -f A B再切换新分支git checkout XXX

要在心里牢记 cherry-pick 可以将提交树上任何地方的提交记录取过来追加到 HEAD 上（只要不是 HEAD 上游的提交就没问题）

这一关的目标和上一关一样，通过 --amend 改变提交记录 C2，但你不能用 rebase -i。

指向C2就会有C2的amend

Git 的 tag 就是干这个用的啊，它们可以（在某种程度上 —— 因为标签可以被删除后重新在另外一个位置创建同名的标签）永久地将某个特定的提交命名为里程碑，然后就可以像分支一样引用了。

更难得的是，它们并不会随着新的提交而移动。你也不能切换到某个标签上面进行修改提交，它就像是提交树上的一个锚点，标识了某个特定的位置。

git tag V1 C1

我们将这个标签命名为 v1，并且明确地让它指向提交记录 C1，如果你不指定提交记录，Git 会用 HEAD 所指向的位置。

按照目标建立两个标签，然后切换到 v1 上面，要注意你会进到分离 HEAD 的状态 —— 这是因为不能直接在v1 上面做 commit。

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 的​​语法是：

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 会使用你目前所在的位置（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

多分支 rebase

哇，现在我们这里出现了很多分支呢！让我们把所有这些分支上所做的工作都通过 rebase 合并到 main 分支上吧。

但是你的领导给你提了点要求 —— 他们希望得到有序的提交历史，也就是我们最终的结果应该是 C6' 在 C7' 上面， C5' 在 C6' 上面，依此类推。

即使你搞砸了也没关系，用 reset 命令就可以重新开始了。

选择 parent 提交记录

操作符 ^ 与 ~ 符一样，后面也可以跟一个数字。

但是该操作符后面的数字与 ~ 后面的不同，并不是用来指定向上返回几代，而是指定合并提交记录的某个 parent 提交。还记得前面提到过的一个合并提交有两个 parent 提交吧，所以遇到这样的节点时该选择哪条路径就不是很清晰了。

Git 默认选择合并提交的“第一个” parent 提交，在操作符 ^ 后跟一个数字可以改变这一默认行为。

这里有一个合并提交记录。如果不加数字修改符直接切换到 main^，会回到第一个 parent 提交记录。

(在我们的图示中，第一个 parent 提交记录是指合并提交记录正上方的那个提交记录。)

git checkout main^

git checkout HEAD~

git checkout HEAD^2不是正上方的那个第二个分支

git checkout HEAD~2再往上两个分支

git checkout HEAD~^2~2支持链式操作

要完成此关，在指定的目标位置创建一个新的分支。

很明显可以简单地直接使用提交记录的哈希值（比如 C6），但我要求你使用刚刚讲到的相对引用修饰符！

现在我们的 main 分支是比 one、two 和 three 要多几个提交。出于某种原因，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上。

one 需要重新排序并删除 C5，two 仅需要重排排序，而 three 只需要提交一次。

慢慢来，你会找到答案的 —— 记得通关之后用 show solution 看看我们的答案哦。

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), 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!

我们创建远程仓库的命令

直到现在, 教程都聚焦于本地仓库的操作（branch、merge、rebase 等等）。但我们现在需要学习远程仓库的操作 —— 我们需要一个配置这种环境的命令, 它就是 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。显然这和真实命令的意思刚好相反，但是它帮咱们把本地仓库和远程仓库关联到了一起，在教程中就凑合着用吧。

git clone

远程分支

既然你已经看过 git clone 命令了，咱们深入地看一下发生了什么。

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.

远程分支有一个特别的属性，在你切换到远程分支时，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。

为什么有 o/？

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。

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/main;git commit

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

要通过本关，在 main 分支上做一次提交；然后切换到 o/main，再做一提交。这有助于你理解远程分支的不同，他们的更新只是反映了远程的状态。

Git Fetch

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。在上一节课程中我们已经提及过这一点了。

在解释 git fetch 前，我们先看看实例。这里我们有一个远程仓库, 它有两个我们本地仓库中没有的提交。git fetch

就是这样了! C2,C3 被下载到了本地仓库，同时远程分支 o/main 也被更新，反映到了这一变化

git fetch 做了些什么

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/main)

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。

it fetch 不会做的事

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

Git Pull

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/main

git rebase o/main

git merge o/main

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。

我们先来看看 fetch、merge 依次执行的效果

git fetch; git merge o/main

我们用 fetch 下载了 C3, 然后通过 git merge o/main 合并了这一提交记录。现在我们的 main 分支包含了远程仓库中的更新（在本例中远程仓库名为 origin）

如果使用 git pull 呢?

同样的结果！这清楚地说明了 git pull 就是 git fetch 和 git merge 的缩写

模拟团队合作

这里有一件棘手的事 —— 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..

fakeTeamwork 默认操作就是在远程仓库的 main 分支上做一次提交。

完成了 —— 远程仓库增加了一个新提交，我们还没有下载它，因为我们还没有执行 git fetch。

Git示范

你还可以指定提交的分支或是数量，只需要在命令后加上它们就可以了。

git fakeTeamwork foo 3

Git示范

你还可以指定提交的分支或是数量，只需要在命令后加上它们就可以了。

通过一个命令，我们就模拟队友推送了 3 个提交记录到远程仓库的 foo 分支。

克隆一个远程仓库（用 git clone），再在刚创建的远程仓库中模拟一些修改，然后在你自己的本地分支上做一些提交，再拉取远程仓库的变更。

Git Push

OK，我们已经学过了如何从远程仓库获取更新并合并到本地的分支当中。这非常棒……但是我如何与大家分享我的成果呢？

嗯，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。它有许多应用技巧，稍后我们会了解到，但是咱们还是先从基础的开始吧……

注意 —— 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。

这里我们准备了一些远程仓库中没有的提交记录, 咱们开始先上传吧! git push过去了, 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！

要完成本关，需要向远程仓库分享两个提交记录。

偏离的工作

现在我们已经知道了如何从其它地方 pull 提交记录，以及如何 push 我们自己的变更。看起来似乎没什么难度，但是为何还会让人们如此困惑呢？

困难来自于远程库提交历史的偏离。在讨论这个问题的细节前，我们先来看一个例子……

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。

git push看见了吧？什么都没有变，因为命令失败了！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。

那该如何解决这个问题呢？很简单，你需要做的就是使你的工作基于最新的远程分支。

有许多方法做到这一点呢，不过最直接的方法就是通过 rebase 调整你的工作。咱们继续，看看怎么 rebase！

git fetch;git rebase o/main;git push

从远程仓库下载并更新远程分支提交到本地

合并

从本地推送记录到远程

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。

咱们们用 merge 替换 rebase 来试一下……

git fetch;git merge o/main;git push

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

很好！但是要敲那么多命令，有没有更简单一点的？

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！

让我们看看简写命令是如何工作的。这次用 --rebase……

git pull --rebase;git push更短了

换用常规的 pull

git pull;git push还是跟以前一样！

由 fetch、rebase/merge 和 push 组成的工作流很普遍。后续课程我们会讲解更复杂的工作流，不过现在我们先解决这个关卡吧。

要完成本关，你需要完成以下几步：

克隆你的仓库

模拟一次远程提交（fakeTeamwork）

完成一次本地提交

用 rebase 发布你的工作

git clone

git fakeTeamwork main

git pull

git checkout C1

git commit

git rebase C2 C3

git branch -f main HEAD

git checkout main

git push

远程服务器拒绝!(Remote Rejected)

如果你是在一个大的合作团队中工作, 很可能是main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main, 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.)

为什么会被拒绝?

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.

解决办法

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.

合并特性分支

既然你应该很熟悉 fetch、pull、push 了，现在我们要通过一个新的工作流来测试你的这些技能。

在大型项目中开发人员通常会在（从 main 上分出来的）特性分支上工作，工作完成后只做一次集成。这跟前面课程的描述很相像（把 side 分支推送到远程仓库），不过本节我们会深入一些.

但是有些开发人员只在 main 上做 push、pull —— 这样的话 main 总是最新的，始终与远程分支 (o/main) 保持一致。

对于接下来这个工作流，我们集成了两个步骤：

将特性分支集成到 main 上

推送并更新远程分支

让我们看看如何快速的更新 main 分支并推送到远程。

git pull --rebase;git push

我们执行了两个命令:

将我们的工作 rebase 到远程分支的最新提交记录

向远程仓库推送我们的工作

这个关卡的 Boss 很厉害 —— 以下是通关提示：

这里共有三个特性分支 —— side1 side2 和 side3

我需要将这三分支按顺序推送到远程仓库

因为远程仓库已经被更新过了，所以我们还要把那些工作合并过来

为什么不用 merge 呢?

为了 push 新变更到远程仓库，你要做的就是包含远程仓库中最新变更。意思就是只要你的本地分支包含了远程分支（如 o/main）中的最新变更就可以了，至于具体是用 rebase 还是 merge，并没有限制。

那么既然没有规定限制，为何前面几节都在着重于 rebase 呢？为什么在操作远程分支时不喜欢用 merge 呢？

在开发社区里，有许多关于 merge 与 rebase 的讨论。以下是关于 rebase 的优缺点：

优点:

Rebase 使你的提交树变得很干净, 所有的提交都在一条线上

缺点:

Rebase 修改了提交树的历史

比如, 提交 C1 可以被 rebase 到 C3 之后。这看起来 C1 中的工作是在 C3 之后进行的，但实际上是在 C3 之前。

一些开发人员喜欢保留提交历史，因此更偏爱 merge。而其他人（比如我自己）可能更喜欢干净的提交树，于是偏爱 rebase。仁者见仁，智者见智。 :D

本关，我们还是解决上一关卡中的问题，但是要用 merge 替换 rebase。这显然有点画蛇添足，但这只是为了更好的说明上面的观点。

远程跟踪分支

在前几节课程中有件事儿挺神奇的，Git 好像知道 main 与 o/main 是相关的。当然这些分支的名字是相似的，可能会让你觉得是依此将远程分支 main 和本地的 main 分支进行了关联。这种关联在以下两种情况下可以清楚地得到展示：

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！

远程跟踪

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。

这也解释了为什么会在克隆的时候会看到下面的输出：

local branch "main" set to track remote branch "o/main"

我能自己指定这个属性吗？

当然可以啦！你可以让任意分支跟踪 o/main, 然后该分支会像 main 分支一样得到隐含的 push 目的地以及 merge 的目标。 这意味着你可以在分支 totallyNotMain 上执行 git push，将工作推送到远程仓库的 main 分支上。

有两种方法设置这个属性，第一种就是通过远程分支切换到一个新的分支，执行:

git checkout -b totallyNotMain o/main

就可以创建一个名为 totallyNotMain 的分支，它跟踪远程分支 o/main。

闲话少说，咱们先看看演示！我们切换到一个名叫 foo 的新分支，让其跟踪远程仓库中的 main

git checkout -b foo o/main;git pull

正如你所看到的, 我们使用了隐含的目标 o/main 来更新 foo 分支。需要注意的是 main 并未被更新

Git示范

git push 同样适用

git checkout -b foo o/main;git commit;git push

我们将一个并不叫 main 的分支上的工作推送到了远程仓库中的 main 分支上

第二种方法

另一种设置远程追踪分支的方法就是使用：git branch -u 命令，执行：

git branch -u o/main foo

这样 foo 就会跟踪 o/main 了。如果当前就在 foo 分支上, 还可以省略 foo：

git branch -u o/main

OK! 本节我们在不切换到 main 分支的情况下将工作推送到的远程仓库中的 main 分支上。

Git Push 的参数

很好! 既然你知道了远程跟踪分支，我们可以开始揭开 git push、fetch 和 pull 的神秘面纱了。我们会逐个介绍这几个命令，它们在理念上是非常相似的。

首先来看 git push。在远程跟踪课程中，你已经学到了 Git 是通过当前所在分支的属性来确定远程仓库以及要 push 的目的地的。这是未指定参数时的行为，我们可以为 push 指定参数，语法是：

git push <remote> <place>

<place> 参数是什么意思呢？我们稍后会深入其中的细节, 先看看例子, 这个命令是:

git push origin main

把这个命令翻译过来就是：

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。

需要注意的是，因为我们通过指定参数告诉了 Git 所有它需要的信息, 所以它就忽略了我们所切换分支的属性！

git checkout C0;git push origin main

git checkout C0;git push命令失败了（正如你看到的，什么也没有发生）! 因为我们所切换的 HEAD 没有跟踪任何分支。

本关我们要更新远程仓库中的 foo 和 main, 但是 git checkout 被禁用了！

注意：远程分支使用 o/ 开头是因为 origin/ 对于 UI 来说太长了。不用太在意这个，直接用 origin 作为远程仓库的名称就可以了。

<place>参数详解

还记得之前课程说的吧，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。

哎，很遗憾 Git 做不到…… 开个玩笑，别当真！当然是可以的啦 :) Git 拥有超强的灵活性（有点过于灵活了）

接下来咱们看看是怎么做的……

要同时为源和目的地指定 <place> 的话，只需要用冒号 : 将二者连起来就可以了：

git push origin <source>:<destination>

这个参数实际的值是个 refspec，“refspec” 是一个自造的词，意思是 Git 能识别的位置（比如分支 foo 或者 HEAD~1）

一旦你指定了独立的来源和目的地，就可以组织出言简意赅的远程操作命令了，让我们看看演示！

记住，source 可以是任何 Git 能识别的位置：

git push origin foo^:main这是个令人困惑的命令，但是它确实是可以运行的 —— Git 将 foo^ 解析为一个位置，上传所有未被包含到远程仓库里 main 分支中的提交记录。

如果你要推送到的目的分支不存在会怎么样呢？没问题！Git 会在远程仓库中根据你提供的名称帮你创建这个分支！

git push origin main:newBranch

在这个关卡中，试着完成目标窗口展示的提交树，记住参数格式哟：

<source>:<destination>

选中foo git push origin foo:main 使o/main重新指向foo

选中C5 git push origin main^:foo 使foo重新指向main的上一个

Git fetch 的参数

我们刚学习了 git push 的参数，很酷的 <place> 参数，还有用冒号分隔的 refspecs（<source>:<destination>）。 这些参数可以用于 git fetch 吗？

你猜中了！git fetch 的参数和 git push 极其相似。他们的概念是相同的，只是方向相反罢了（因为现在你是下载，而非上传）

让我们逐个讨论下这些概念……

<place> 参数

如果你像如下命令这样为 git fetch 设置 的话：

git fetch origin foo

Git 会到远程仓库的 foo 分支上，然后获取所有本地不存在的提交，放到本地的 o/foo 上。

来看个例子（还是前面的例子，只是命令不同了）

git fetch origin foo

我们只下载了远程仓库中 foo 分支中的最新提交记录，并更新了 o/foo

你可能会好奇 —— 为何 Git 会将新提交放到 o/foo 而不是放到我本地的 foo 分支呢？之前不是说这样的 参数就是同时应用于本地和远程的位置吗？

好吧, 本例中 Git 做了一些特殊处理，因为你可能在 foo 分支上的工作还未完成，你也不想弄乱它。还记得在 git fetch 课程里我们讲到的吗 —— 它不会更新你的本地的非远程分支, 只是下载提交记录（这样, 你就可以对远程分支进行检查或者合并了）。

“如果我们指定 <source>:<destination> 会发生什么呢？”

如果你觉得直接更新本地分支很爽，那你就用冒号分隔的 refspec 吧。不过，你不能在当前切换的分支上干这个事，但是其它分支是可以的。

这里有一点是需要注意的 —— source 现在指的是远程仓库中的位置，而 <destination> 才是要放置提交的本地仓库的位置。它与 git push 刚好相反，这是可以讲的通的，因为我们在往相反的方向传送数据。

理论上虽然行的通，但开发人员很少这么做。我在这里介绍它主要是为了从概念上说明 fetch 和 push 的相似性，只是方向相反罢了。

git fetch origin foo~1:bar

哇! 看见了吧, Git 将 foo~1 解析成一个 origin 仓库的位置，然后将那些提交记录下载到了本地的 bar 分支（一个本地分支）上。注意由于我们指定了目标分支，foo 和 o/foo 都没有被更新。

Git示范

如果执行命令前目标分支不存在会怎样呢？我们看一下上个对话框中没有 bar 分支的情况。

git fetch origin foo~1:bar

看见了吧，跟 git push 一样，Git 会在 fetch 前自己创建立本地分支, 就像是 Git 在 push 时，如果远程仓库中不存在目标分支，会自己在建立一样。

没有参数呢?

如果 git fetch 没有参数，它会下载所有的提交记录到各个远程分支……

git fetch

相当简单，但是仅需更新一次，值得你去做！

好, 说得太多了！要完成本关，抓取目标窗口中指定的提交记录，使用这些魔幻的命令吧！

使用 fetch 时, 你必须指定 source 和 destination。 注意一下目标窗口, 因为提交对象的 ID 可能会变哦!

古怪的 <source>

Git 有两种关于 <source> 的用法是比较诡异的，即你可以在 git push 或 git fetch 时不指定任何 source，方法就是仅保留冒号和 destination 部分，source 部分留空。

git push origin :side

git fetch origin :bugFix

我们分别来看一下这两条命令的作用……

如果 push 空 到远程仓库会如何呢？它会删除远程仓库中的分支git push origin:foo就是这样子, 我们通过给 push 传空值 source，成功删除了远程仓库中的 foo 分支, 这真有意思...

如果 fetch 空 到本地，会在本地创建一个新分支。git fetch origin:bar

这个关卡很容易 —— 只要删除一个远程的分支, 再用 git fetch 在本地创建一个新分支就可以了！

Git pull 参数

既然你已经掌握关于 git fetch 和 git push 参数的方方面面了，关于 git pull 几乎没有什么可以讲的了 :)

因为 git pull 到头来就是 fetch 后跟 merge 的缩写。你可以理解为用同样的参数执行 git fetch，然后再 merge 你所抓取到的提交记录。

还可以和其它更复杂的参数一起使用, 来看一些例子:

以下命令在 Git 中是等效的:

git pull origin foo 相当于：

git fetch origin foo; git merge o/foo

还有...

git pull origin bar~1:bugFix 相当于：

git fetch origin bar~1:bugFix; git merge bugFix

抓远程的bar~1过来，在本地新建bugFix，merge bar~1和bugFix

看到了? git pull 实际上就是 fetch + merge 的缩写, git pull 唯一关注的是提交最终合并到哪里（也就是为 git fetch 所提供的 destination 参数）

一起来看个例子吧：

如果我们指定要抓取的 place，所有的事情都会跟之前一样发生，只是增加了 merge 操作

git pull origin main看到了吧! 通过指定 main 我们更新了 o/main。然后将 o/main merge 到我们的所在的分支，无论我们当前所在的位置是哪。

pull 也可以用 source:destination 吗? 当然喽, 看看吧:

git pull origin main:foo

哇, 这个命令做的事情真多。它先在本地创建了一个叫 foo 的分支，从远程仓库中的 main 分支中下载提交记录，并合并到 foo，然后再 merge 到我们的当前所在的分支 bar 上。操作够多的吧？！

好啦, 该结束了！请按照目标窗口中的状态进行操作。你需要下载一些提交，然后创建一些新分支，再合并这些分支到其它分支, 但这用不了几个命令 :P