git

Git 是整个 CS 中使用的事实上的标准版本控制系统。它也名副其实，使用起来很*痛苦*。慢慢来，把它当作练习使用 Git 命令行的机会。

这个实验*应该可以*在几乎任何计算机上正常工作（嗯...也许不是 Windows），但我们仍然建议在虚拟机中完成它（主要是因为我们可以在虚拟机中测试该实验室...）。

如果您想以此 `Vagrantfile` 为基础进行工作，可以使用以下内容：

```
Vagrant.configure("2") do |config|
  config.vm.box = "generic/debian12"
  config.vm.synced_folder ".", "/vagrant"

  config.vm.provision "shell", inline: <<-SHELL
    apt-get update -y
    apt-get install -y git git-man apt-file
    apt-file update
  SHELL
end
```

Git 文档

Git 附带了*大量*文档。跑步：

```
apropos git
```

要查看全部内容，或运行：

```
apropos git -a tutorial
```

查找与 git 和（ `-a` ）教程相关的文档。阅读您认为对该命令可能有用的任何文档 `man` 。

任务：有一个手册页记录了您可能想要使用的*日常 git 命令*。使用命令查找 `apropos` 并读取它 `man` 。

您可能还想阅读这些 `gittutorial` 页面...

配置您的身份

Git 的作用就是跟踪源代码的更改。这意味着它需要知道*谁*做了*什么*更改。

运行以下两行以正确设置 git。您只需在安装 git 时执行一次此操作，而不是每次创建新存储库时执行此操作。

```
git config --global user.name "YOURNAME"
git config --global user.email "YOUREMAIL"
```

姓名和电子邮件地址实际上并未发送到任何地方或进行检查...它们只是与您对代码所做的更改一起列出，因此如果出现问题，后来的程序员知道应该归咎于谁（请参阅 参考资料 `man git-blame` ）。你可以在这里放置任何你喜欢的内容（git 会很乐意接受 - 作为你的电子邮件地址，并且它不会向你发送电子邮件）。

这会更改*全局git 配置（应用于您使用的每个*git 存储库的设置），但您也可以逐个存储库进行这些更改。只需删除 `--global` 并在要应用更改的 git 存储库中运行命令即可。*如果您是Bruce Wayne* 并且需要将公共和私人开发项目分开（或者如果您进行分包开发工作），这非常有用。

对于那些使用 Vagrant 的人

如果您在实验室计算机上运行虚拟机，则每次 vagrant 重建虚拟机时都需要重新配置 git，例如当您登录到不同的实验室计算机时。您可以将这些命令放入您的 Vagrantfile 中，就像您在 vagrant（重新）构建您的盒子时想要运行的任何其他命令一样，但它们需要以 vagrant 用户而不是 root 用户身份运行。因此，将以下块添加到该行 `Vagrantfile` 之前 `end` ，明显地编辑您的姓名和电子邮件地址。通常 vagrant 将以系统管理员身份运行这些配置块，但您可以通过添加关键字 `root` 以普通用户身份运行它。 `vagrant privileged: false`

```
config.vm.provision :shell, privileged: false, inline: <<-SHELL
    git config --global user.name "YOURNAME"
    git config --global user.email "YOUREMAIL"
SHELL
```

如果您开始收到有关未安装 git 命令的错误：安装它！如果您使用基于 Debian 的虚拟机，您需要的命令是 `apt` （看看 `man apt` 您是否不熟悉它）。
有些人发现有两个配置块有点混乱。您可以将它们减少到只有一个块，但您需要使用该 `su` 命令来确保将 git 配置为 `vagrant` 用户。

示例项目和存储库

假设您想开始开发 C 程序。让我们创建一个文件夹：

```
mkdir project1
cd project1
git init
```

最后一个命令在名为 的子文件夹中创建了一个空的 git 存储库 `.git` 。我们可以检查 `git status` 是否有任何更改，以及 git 报告 `nothing to commit` 。

使用文本编辑器创建一个文件 `main.c` ，并添加一些示例内容，如下所示（您应该能够复制粘贴到终端中）：

```
// file: main.c
#include <stdio.h>

int main() {
    puts("Hi");
    return 0;
}
```

Advanced note

您应该使用哪种文本编辑器？老实说，这并不重要，但它需要能够编写纯文本文档（没有格式的文档——所以不是 Microsoft Word）。

标准*编辑*器称为 `ed` : 但不要使用它！它是为计算机有屏幕之前而设计的。

传统*的*程序员编辑器是 `vim` 或 `emacs` 。您*应该*学习其中之一（Jo 和 Matt 都会推荐 `vim` ），但如果您以前从未使用过程序员的编辑器，那么它们会相当令人困惑。不过绝对值得；只是预计在一两周内打字速度会相当慢。

更简单的编辑器包括 `nano` 在命令行上运行的编辑器、 `gedit` 图形化的编辑器或 `Visual Studio` 可以在远程系统上编辑代码的 Microsoft 编辑器。它们都是可配置的，您可以让它们全部执行诸如语法突出显示和代码完成之类的操作。

我们不关心您使用什么编辑器：但请确保通过配置使其适合您。您将花费大量的学位来编写代码：现在在文本编辑器上进行一些投资将在以后获得回报！

（我认为每个人都会评判那些在最后一年*仍在*使用的人……） `nano`

执行 a `git status` ，您将在*未跟踪的文件* `main.c` 下看到红色- 这是 git 还不知道的新文件。然后执行另一个操作，该文件现在在*要提交的文件*下呈绿色。 `git add main.c git status`

`git commit -m "first file"` 使用或类似的内容提交文件- 如果您希望提交消息中包含空格，则需要双引号。再试 `git status` 一次，您应该看到*没有任何可提交的内容，工作树干净*，这意味着 git 已更新您的文件。尝试一下 `git log` ，您会看到日志中现在有一个提交。

Advanced note

每个 git 提交都必须有一个提交消息。你可以添加带有该 `-m` 标志的一个，也可以将其关闭，git 会将你带入系统默认编辑器来编写一个。这通常是 vim （退出的命令是按 escape 键然后

ZZ ）。您可以通过使用命令设置*环境变量*来更改默认编辑器 `export EDITOR=nano` 。

如果您想在下次登录时重新启动 shell 时保留此设置，则导出行必须放入 `.profile` 主目录中调用的文件中，该文件是 bash shell 启动时处理的文件。

要在 vagrant 重建虚拟机时保留配置文件，您可以在 vagrantfile 中添加一条配置行以确保 `.profile` 更新：

```
echo 'export EDITOR=ed' >>~vagrant/.profile
```

忽略文件

使用 编译您的代码 `gcc main.c -o program` ，并检查 `./program` 它是否运行并打印*Hi*。（如果出现不存在的错误 `stdio.h` ，则表明您安装了 gcc，但未安装 C 开发库提示： `man apt-file` 。）

如果您现在查看 `git status` ，程序文件显示为未跟踪，但我们不想提交它：当您仅存储源代码时，存储库效果最好，任何需要的人都可以签出副本并从那里构建。除此之外，这意味着不同平台（例如 Linux 和 Mac、intel 和 ARM 等）上的人们都可以编译适合他们的版本。

因此，我们想告诉 git 忽略程序及其中的更改，我们通过创建一个名为 的文件 `.gitignore` 并在每一行添加一个表达式来说明要忽略哪些文件或文件夹 - 您可以使用它 `*.o` 来选择所有目标代码例如，文件。

- 创建一个文件并向其中 `.gitignore` 添加单行。 `program`
- 再做一次 `git status` ，请注意，虽然程序现在被忽略，但忽略文件被标记为新文件。该文件确实属于存储库，因此添加它并提交它。
- 检查 `git status` 报告是否再次*干净*，并且 `git log` 包含两次提交。

提交并结账

在开发过程中，您应该定期编码、提交、重复。要练习这一点，请在程序中将*Hi*更改为*Hello* `git status` ，重建并运行程序，然后再次添加并提交源文件 -最后检查是否再次*干净*。

该命令 `git add .` 一次性添加当前文件夹中的所有新的和更改的文件和文件夹，并且当您想要提交自上次提交以来的所有更改时，通常是添加内容的最快方法。

有时您想返回并查看另一个提交，或者撤消破坏某些内容的提交 - 这就是您想要签出的时候。

- 用于 `git log` 显示您的提交历史记录。（当您有多个屏幕时， `git log |less` 可以滚动。）
- 请注意添加忽略文件的提交的提交哈希的前 6 个左右字符，但在将*Hi*更改为*Hello*之前。您至少需要 6 个字符，但只要足够多，这样 git 就不会混淆您所指的提交。

- `git checkout HASH` 在 HASH 为 6 或您需要的相关提交字符数的情况下运行。Git 将打印有关 HEAD 指针的警告。
- 检查源文件，注意它现在又回到了*Hi*上。
- 用于 `git checkout main` 返回文件的最新版本，git 将再次设置 HEAD 指针，准备接受新的提交。

Advanced note

如果您确实想撤消提交，那么您有两种选择：

- `git revert HASH` 添加一个新的提交，将文件返回到使用给定哈希提交之前的状态。这在团队开发期间可以安全使用，因为它只是添加新的提交。如果您已经提交了 A、B，那么 `git revert B` 您将获得一个新的提交 C，以便其他使用存储库的人看到一系列提交 A、B、C；但C中文件的状态与A中相同。
- `git reset HASH` 通过将 HEAD 指针移回到具有给定哈希的提交来撤消提交，但保留工作副本（您 `--hard` 也可以使用该选项来更改文件）。如果您与其他开发人员共享了新的提交，这会破坏事情，但可以安全地用于撤消尚未推送的更改（我们下次将了解这一点）。效果就好像您重置的提交从未发生过一样。

注意：如果你想恢复提交，因为你不小心提交了一个包含秘密信息的文件，并且你已经推送了该提交，那么你还必须在线查找如何“强制推送”你的更改以删除所有痕迹github（或其他在线提供商）上的文件。如果秘密文件包含任何密码，即使您立即恢复提交，那么您也应该考虑密码已泄露并立即更改它们。

第 2 部分：Git 锻造

在本练习中，我们将通过远程提供商设置并使用 git forge 帐户。您通常看到的托管 git 存储库的*典型*内容是：

- github.com
- gitlab.com
- bitbucket.org

但还存在更多。您甚至可以仅使用 SSH 服务器来创建自己的服务器。乔最喜欢的是一个叫 sourcehut的地方，但你必须付费。

Advanced note

如果你确实想从头开始构建自己的 git 服务器，这并不难，但你必须使用*裸露的*git 存储库（我们不涉及）并设置一些时髦的文件权限。勇敢者可以在这里找到说明。

本练习基于 GitHub，因为它是最受欢迎的提供商，但如果您愿意，也可以使用其他两个提供商之一 — 尽管 Web 用户界面和一些高级功能不同，但可以在命令行是相同的，并且所有三个都提供无限的私人和公共存储库（在合理范围内）。

设置好一切

访问github.com并使用用户名、电子邮件地址和密码进行注册。您可能需要单击发送给您的电子邮件中的确认链接。

我们将通过 SSH 使用 git，因此您需要让 git 知道您的公钥（记住，您永远不会向任何人提供您的私钥！）。单击屏幕右上角代表您头像的图标（当然您可以根据需要设置自定义头像），然后在菜单中选择*"设置"*，然后在设置页面上选择*"SSH 和 GPG 密钥"*。

选择*New SSH key*，然后将您的 SSH 公钥粘贴到框中 `Key` （您上周创建了一个，请参阅 `man ssh-keygen` ）。如果您愿意，可以为您的密钥指定一个标题，然后使用绿色按钮添加它。Github 支持所有常见的 SSH 密钥格式，但如果您做了一些愚蠢的事情（例如上传私钥或使用过时且弱密码的密钥），它会向您发出警告。一些提供商 (Bitbucket) 坚持要求您使用特定类型的密钥（通常）：如果需要， `ed25519` 在生成密钥时添加适当的标志来创建它 ( )。 `-t ed25519`

Advanced note

如果您有许多工作设备（台式机、笔记本电脑）和连接的许多服务器（github、gitlab、实验室机器等），您如何管理密钥?

虽然为不同的服务使用相同的公钥并不完全是一个安全问题：即使一项服务被黑客攻击并且您在黑客控制下连接到它，也不会泄露您的私钥；感觉有点恶心。生成密钥很容易，只需为您使用的每台服务器和每台机器拥有单独的密钥就可以了。

然而，重复使用公钥可能会带来隐私问题，因为您使用相同公钥（或电子邮件地址或电话号码等）的每项服务都可能与其他服务合作，从而知道您是同一个人。对不同的服务使用不同的密钥对是没有问题的，在这种情况下，您可能需要在 `~/.ssh/config` 文件中使用类似的块

```
Host github.com
    User git
    IdentityFile ~/.ssh/github.com.key.or.whatever.you.called it.
```

搜索手册页以 `ssh_config` 获取完整的配置选项。

我们假设您将在虚拟机上运行本节其余部分中的 git 命令，无论是在您的计算机上还是在实验室计算机上，但是如果您直接在自己的计算机上安装了 git（这是一个好主意），那么你也可以在那里进行这个练习。

命名注意事项

主分支的名称发生变化：以前称为 `master` . 您可能会看到名为 `master` 或 的默认分支 `main` ，或者完全是其他名称。只要保持一致，默认分支的名称根本不重要（如果您有偏好，您可以配置它），您只需要知道在这些练习中我们将使用它 `main` 来引用默认分支分支，如果不同，您应该将其替换为您自己的默认分支名称。

创建存储库

在主页上，您应该在左侧看到一个空的*存储库栏，并带有一个新按钮*。使用它来创建存储库，在下一页上为其命名并勾选*"添加自述文件"*框。

在存储库页面上，有一个绿色的*代码*按钮。单击该按钮将打开一个包含三个选项卡的框：*HTTPS*、*SSH*和*GitHub CLI*。

每个存储库都有一个由两部分组成的名称：第一部分是所有者的 github 用户名，第二部分是存储库名称。例如，该单元的存储库称为 `cs-uob/COMSM0085` 。有两种方式与远程存储库交互：

- 通过 HTTPS。如果您只是克隆公共存储库，这是可以的，因为它不需要任何身份验证。要与私有存储库交互或推送文件，HTTPS 需要用户名/密码身份验证，我们可以做得更好。
- 通过 SSH，使用密钥。这是使用 Git 的推荐方式。

单击 SSH 选项卡并将 URL 复制到此处 - 它应该类似于 `git@github.com:USERNAME/REPONAME.git` 。

在命令行上，运行命令， `git clone git@github.com:USERNAME/REPONAME.git` 将 USERNAME 和 REPONAME 替换为存储库的 SSH 选项卡中的部分。Git 克隆您的存储库并将内容放入以存储库名称命名的子文件夹中 - 您可以通过提供不同的文件夹名称作为额外的命令行参数来更改此设置 `git clone` ，或者您可以稍后移动或重命名该文件夹。

注意：当您尝试通过 ssh 访问 github 时，某些操作系统/ISP/DNS 组合可能会导致"资源暂时不可用"。问题在于， `ssh.github.com` 当您尝试直接连接到 github 时，实际地址并非所有设置都能正确传递重定向。如果您遇到此错误，您可以使用 `ssh.github.com` 代替 `github.com` ，或在您的 `~/.ssh/config` 文件中添加一个条目，如下所示（如果您必须先创建此文件，请确保除您自己之外的任何人都不可写入该文件，否则 ssh 将拒绝接受它）：

```
Host github.com
  Hostname ssh.github.com
  Port 22
```

转到该文件夹，然后尝试 `git remote show origin` 。这里，是*远程* `origin` 的默认名称，结果应该看起来有点像这样：

```
* remote origin
  Fetch URL: git@github.com:USERNAME/REPONAME
  Push  URL: git@github.com:USERNAME/REPONAME
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

有关的内容 `main` 与分支有关，我们将在另一项活动中更详细地讨论。

Advanced note

您可以拥有多个具有不同名称的遥控器 - 例如，如果您分叉（创建自己的其他人存储库的副本），那么您将获得原始遥控器作为名为*上游*的第二个遥控器，因此您可以与他们共享更改 - 这就是方式例如，您为CSS 网站创建新内容。

您还可以使用文件夹作为远程文件夹：如果您想练习解决合并冲突，您可以执行以下操作：

```
mkdir remote
cd remote
git init --bare
cd ..
mkdir user1
git clone remote user1
mkdir user2
git clone remote user2
```

这将为您提供一个遥控器和两个位于不同文件夹中的“用户”供您使用。远程设置 `--bare` 使其不包含工作副本，而是充当纯存储库。

您现在可以 `cd user1` 模拟用户 1 正在工作，并且可以按如下所述进行获取/推送/拉取。（忽略有关“上游”的警告，一旦您将文件提交到存储库，它们就会消失。）然后您可以 `cd ../user2` 切换到第二个工作副本，您可以假装它是另一台计算机上的另一个用户。

如果您想调整提交的用户名，则运行 `git config user.name "YOURNAME"` 且不使用上次的 `git config user.email "YOUREMAIL"` 选项 `--global` 只会更改一个存储库的设置。

执行 a `git status` 并注意与上次活动相比出现了新行：

```
Your branch is up to date with 'origin/main'.
```

该系列有四个版本：

- 最新：自上次同步以来，本地或远程存储库上没有任何提交。
- 远程之前：您已在本地进行提交，但尚未推送到远程。
- 远程背后：其他人或另一台计算机上的您已对远程进行了提交，但您在此计算机上尚未提交。
- 与远程分歧：自上次同步以来，您的计算机和远程都有不同的提交。

练习推送工作流程

对于此练习，您应该两人一组或更大的小组进行。需要有人一起工作吗？询问坐在你旁边的人。

一个人创建一个私有存储库（勾选该框以添加自述文件）并将组中的其他人添加到其中。你们都需要拥有同一提供商的帐户才能正常工作。

- 在 Github 上，将人员添加到存储库的方法是在存储库页面上：选择顶部菜单中的*"设置"*，*然后选择"管理访问"*。在这里，您可以按*"邀请协作者"*并输入他们的 Github 用户名。这会导致 Github 向他们发送一封电子邮件，其中包含一个链接，他们需要单击该链接才能接受邀请并将其添加到存储库中。*注意：点击邀请邮件中的链接时，您必须登录github，否则会收到错误消息。*

每个人 `git clone` 都将存储库存储到自己的 Debian VM（或直接存储在自己的机器上）。

每个人都执行以下操作，一次一个人执行所有步骤（相互协调）：

1. 想象一下，现在是上午，您正在开始一天的编码。
2. 首先，确保您的终端位于包含工作副本的文件夹中，然后输入 `git fetch` 。
    - 如果您没有收到更新，则自上次获取以来遥控器上没有任何更改，您就可以开始编码了。（这应该发生在第一个执行此步骤的人身上。）
    - 如果您得到输出，则说明遥控器上发生了变化。执行 a `git status` 查看详细信息（除了第一个人之外的每个人都应该看到这一点）。请注意该行 `behind origin/main ... can be fast-forwarded.` ，这意味着您只需这样做 `git pull` 即可获得文件的最新版本。也执行 `git log` 一次以查看最后一个人的提交消息。
3. 进行一些编码：对存储库进行更改 - 添加或更改文件，然后提交更改。如果您已按照上一个活动中的说明安装了终端，则可以使用 `nano FILENAME` 它在终端中创建和编辑文件。
4. 运行以下推送工作流程将更改推送到远程：
    1. 执行 a `git fetch` 查看是否有任何远程更改（对于本练习来说不应该有）。
    2. 做一个 `git status` 并确保你是 `ahead of origin` ，不是 `diverged` 。
    3. 执行 a `git push` 将您的更改发送到遥控器。

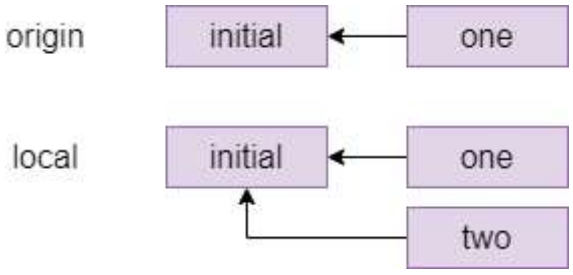
现在，您可以作为一个团队进行编码，只要一次只有一个人在工作 - 显然并不理想。

解决假冲突，第一部分

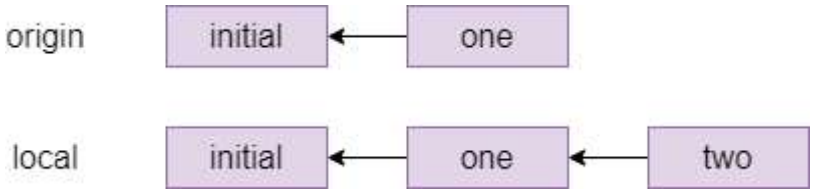
产生"假"冲突如下：

1. 两名团队成员确保他们拥有 `up to date` 自己的工作副本（执行 a `git pull` ，然后执行 `git status` ）。这代表你们都在早上开始编码。
2. 一名成员添加或更改一个文件，然后提交此更改并通过运行整个推送工作流程（获取、状态 - 检查是否领先、推送）来推送它。
3. 在第一个成员执行步骤 2 的同时，第二个成员添加或更改不同的文件，然后提交此更改。这代表两个团队成员并行工作，其中一个成员是第一个完成其工作并将提交推回远程的成员。
4. `git fetch` 第二个成员使用、然后启动推送工作流程 `git status` 。注意你有 `diverged` .（如果您尝试这样做 `git push` ，无论是否获取都会产生错误。）

成员二的提交图如下所示：



解决此冲突的一种方法是*rebase*，它假装第二个成员 one 在开始自己的工作之前实际上已经获取了提交。成员两种类型的命令是， `git rebase origin/main` 这意味着*假设 origin/main 中的所有内容都发生在我开始本地更改之前*，并给出了下图：



事实上，如果成员二 `git status` 在变基后执行了操作，他们将看到 `ahead of origin/main by 1 commit` 并且现在可以 `git push` 将本地更改发送到远程存储库。

不同的公司和团队对于变基何时有意义有不同的看法：有些地方完全禁止这样的变基，至少对于不同人之间真正共享的工作来说是这样。或多或少有一个普遍共识，即当不同的人编辑相同的文件时，您不应该重新设置基准，但对于冲突（例如您刚刚创建的不同人编辑不同文件的冲突），这是一种值得了解的技术，因为它使得更清晰的提交图。

假冲突第二部分

解决冲突的另一种方法 - 也是一些人会使用的唯一方法 - 是合并。让我们再做一次假冲突，但这次通过合并来解决它：

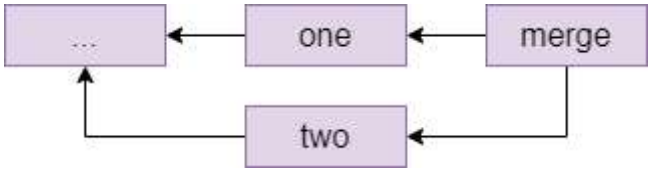
1. 两名团队成员都使用遥控器更新他们的存储库。如果您刚刚按照上述说明进行操作，那么团队成员一必须这样做 `git pull` ，而团队成员二已经是最新的，因为他们刚刚推送了；两个团队成员都应该检查他们是否 `git fetch` 是。 `git status up to date`
2. 像以前一样，团队成员之一编辑一个文件，提交它并执行整个推送工作流程（获取、状态 - 检查是否领先、推送）。同时，第二个团队成员在没有再次获取的情况下编辑了不同的文件并提交。
3. 第二个团队成员开始推送工作流程：获取、状态 - 请注意，您已经 `diverged` .

第二个成员的提交图看起来与变基之前的前一个类似，可能用更多提交代替了第一个提交。

第二个成员即将进行合并，该合并可能会成功（这里应该如此，因为不同的人编辑了不同的文件），也可能因合并冲突而失败（例如，如果不同的人编辑了同一个文件）。如果合并成功，那么 git 将进行合并提交并将它们放入系统的默认编辑器中，通常是 `vi` . 因为我们现在不想了解这一点，所以第二个成员应该 `echo $EDITOR` 在他们的 shell 中输入并查看他们得到了什么 - 如果他们

得到了 nano 那么他们就很好，如果他们得到一个空行那么他们应该这样做 `export EDITOR=nano` 。

第二个团队成员类型 `git pull` 。由于这是一个假冲突（不同的文件），这会让您进入编辑器，您可以看到第一行是一条以 开头的建议提交消息 `Merge branch main` ，按照惯例，无需更改即可接受该消息 - 再次退出编辑器。Git 回复 `Merge made by the recursive strategy.` ，您的提交图现在看起来像这样（代表 `...` 上一节的最后一次提交）：



解决真正的冲突

接下来，我们将练习处理真正的冲突，即两个人编辑同一个文件。

1. 两个团队成员再次同步他们的存储库：每个人都执行一个 `git pull` .
2. 团队成员创建一个名为的文件 `README.md` ，如果该文件已存在，则对其进行编辑，并添加一行，例如 `Created by NAME` 使用他们自己的名称。然后他们提交此更改并运行推送工作流程： `git fetch` , `git status` , 检查它们是否已 ahead 发送 `git push` 到远程。
3. 团队成员二在不获取最新提交的情况下创建相同的 `README.md` 文件并添加一行 `Created by NAME2` 并将其提交到本地存储库。这模拟了两个人在上次提取后并行处理同一文件，其中一个人（在本例中为成员一）是第一个将其更改返回到远程的人。
4. 团队成员二启动推送工作流程： `git fetch` ， `git status` 并注意您 diverged 再次启动了推送工作流程。
5. 以二号成员身份运行 `git pull` 。您应该看到以下消息：

```
CONFLICT (add/add): Merge conflict in README.md
Auto-merging README.md
Automatic merge failed; fix conflicts and then commit the result.
```

例如，打开该文件， `nano README.md` 请注意 git 已对其进行了注释：

```
<<<<<<< HEAD
Created by NAME2.
=======
Created by NAME1.
>>>>>>> b955a75c7ca584ccf0c0bddccbcde46f445a7b30
```

`<<<<<<< HEAD` 和之间的行 `=======` 是本地更改（团队成员二），从 `======` 到 的 `>>>>>> ...` 行是从远程获取的提交中的更改，显示了提交 ID。

成员二现在必须通过编辑文件以生成他们想要提交的版本来解决冲突。例如，您可以删除所有有问题的行并将其替换为 `Created by NAME1 and NAME2.`

成员二现在可以执行以下操作：

- `git add README.md` （或任何其他受到影响的文件）。
- `git commit` 。你可以直接给出一条消息，但是没有提交的提交会让 `-m` 你进入编辑器，你会看到 git 在这里建议 `Merge branch main ...` 作为默认消息。按照惯例，按原样保留此消息，只需退出编辑器而不进行任何更改即可。
- 运行另一个推送工作流程： `git fetch` ， `git status` 请注意您现在 `ahead by 2 commits` ： 第一个是您所做的工作，第二个是合并提交。您已经领先，因此请使用 完成工作流程 `git push` 。

您的提交图看起来与您上次执行的合并相同。

如果您查看 Github 上的存储库页面（ `https://github.com/USERNAME/REPONAME` ，其中 `USERNAME` 是创建存储库的用户的名称），那么您可以单击顶部栏中的*Insights*，*然后单击左侧菜单上的Network*以查看存储库的提交历史记录图形。将鼠标悬停在提交节点上会显示提交者、消息和提交哈希 - 单击节点会将您带到一个页面，您可以在其中查看此提交中进行了哪些更改。

在存储库的 Github 主页面上，您还可以单击右上角带有数字的时钟图标（在足够宽的屏幕上，它还显示单词*commits*）以转到按时间顺序显示存储库中所有提交的页面命令。

与他人合作

在本活动中，您将按照在真实团队中使用 Git 的方式进行练习。您需要组建一个小组来进行这项活动，最好有两名以上的学生。

设置

一名成员在其中一个在线提供商上创建一个 Git 存储库，添加其他团队成员并共享克隆 URL。每个人都克隆存储库。

存储库必须至少有一次提交才能使以下内容正常工作。如果您选择提供商的选项来创建自述文件，则满足此条件；如果没有，那么现在创建一个文件，提交并推送。

开发分支

默认情况下，您的存储库有一个名为 `main` . 但您不想直接在这个分支上完成您的工作。 `develop` 相反，一名团队成员使用以下命令创建一个分支

```
git checkout -b develop
```

创建开发分支的团队成员现在应该对其进行提交。

该分支目前仅存在于他们的本地存储库中，如果他们尝试推送，他们会收到有关此的警告。他们需要做的是

```
git push --set-upstream origin develop
```

这会在本地开发分支上添加一个“上游”条目，表示它链接到名为 的存储库副本 `origin` ， 这是您克隆存储库的默认名称。

您可以使用 进行检查 `git remote show origin` ， 它应该显示以下内容：

```
Remote branches:
  develop tracked
  main  tracked
Local branches configured for 'git pull':
  develop merges with remote develop
  main  merges with remote main
Local refs configured for 'git push':
  develop pushes to develop (up to date)
  main  pushes to main  (up to date)
```

其他人现在可以 `git pull` 使用 来查看分支 `git branch -a` ， `-a` （全部）选项意味着包括仅存在于远程的分支。他们可以使用 切换到开发分支 `git checkout develop` ， 该分支应该显示：

```
Branch 'develop' set up to track remote branch 'develop' from 'origin'.
Switched to a new branch 'develop'
```

特色分支

每个团队成员现在独立尝试以下操作：

- 使用 创建一个新分支 `git checkout -b NAME` ， 为其功能分支选择一个唯一的名称。
- 在此分支上进行一些提交。
- 使用 推送您的功能分支 `git push --set-upstream origin NAME` 。
- 再进行一些提交。
- 做一个简单的事情 `git push` ， 因为您已经将分支链接到 `origin` .

由于每个人都在不同的分支上工作，因此您永远不会以这种方式发生冲突。

任何项目成员都可以访问 github 页面，可以看到那里的所有功能分支，但普通 `git branch` 不会显示你自己从未检查过的其他人的分支。相反，您想要 `git branch -a` 再次执行此操作，以显示所有分支，其名称类似于 `remotes/origin/NAME` 迄今为止仅存在于原始存储库中的分支。您可以像任何其他分支一样检查它们，以在工作副本中查看它们的内容。

合并

当一个功能完成后，您希望将其合并到开发中。每个人都应该尝试这个，其过程是

1. 提交所有更改并推送。
2. 从原点获取最新的更改（一个简单的 `git fetch` 操作）。
3. `git checkout develop`，这会将您切换到开发分支（最新功能的更改将在工作副本中消失，但它们仍在存储库中）。您总是合并到当前活动的分支，因此您需要继续 `develop` 合并到它。
4. `git status` 查看自您启动功能以来是否有其他人更新了开发。如果是这样，那么 `git pull`（你会*落后*而不是*分歧*，因为你还没有改变发展自己）。
5. `git merge NAME` 与您的功能分支的名称。
6. 如有必要，解决冲突（见下文）。
7. `git push` 与团队其他成员分享您的新功能。

如果自您启动分支以来没有其他人更改过 `develop`，或者您只更改了其他人没有更改过的文件，那么合并可能会在第一次尝试时成功。检查项目是否处于良好状态（例如，再次编译）以防万一，并修复开发分支上损坏的任何内容仍然是一个好主意。

如果合并因冲突而失败，那么您需要手动编辑所有冲突的文件（git 会告诉您这些是哪些文件，`git status` 如果您需要提醒，请执行此操作）并 `git commit` 再次编辑。

合并和解决冲突的工作流程本质上与上一个会话的工作流程相同，但由于每个人都在单独的分支上进行开发，因此您必须处理可能的合并冲突的唯一时间是将更改合并到开发 - 你自己的分支是“私有的”，如果你想快速提交并推动你的更改，作为你结束一整天的工作回家之前做的最后一件事，你不必担心会发生冲突。

拉取请求

Pull 请求不是 git 软件本身的功能，而是在线提供商的功能。他们让团队讨论和审查提交，然后将其合并到共享分支（例如开发或主分支）中。根据提供商的不同，分支也可以受到保护或分配所有者，以便只有分支所有者或具有正确权限的开发人员才能在某些分支上提交。

与 github 上的拉取请求合并的过程，您应该尝试一下：

- 提交并推送您的功能分支。
- 在 github.com 的存储库中，选择顶部栏中的*Pull Requests*，*然后选择New Pull Request*。
- *将基本*分支设置为您想要合并到的分支（例如开发），并将*比较*分支设置为包含您的更改的分支。选择*创建拉取请求*。
- 添加标题和描述以开始讨论，然后再次按*“创建拉取请求”*以创建请求。

团队中的任何人现在都可以转到存储库页面顶部栏中的*“拉取请求”并查看打开的请求*。您可以对它们发表评论，或者如果您在团队中的角色负责批准此分支的请求，则您可以批准创建合并的拉取请求。

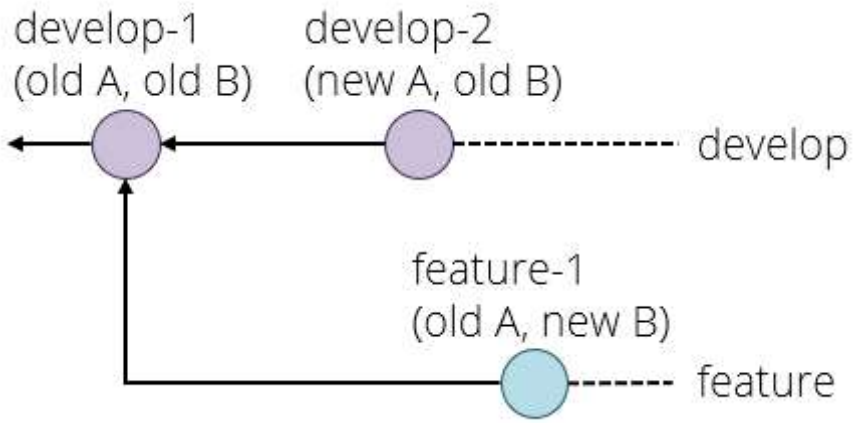
由于拉取请求链接到分支，因此您可以使用它进行代码审查，如下所示：

1. 开发人员创建功能分支并提交拉取请求。
2. 审阅者查看请求。如果他们发现错误或其他问题，他们会在讨论中添加评论。
3. 开发人员可以通过在其功能分支上进行新的提交并推送它来解决审阅者的评论，该提交会自动添加到讨论中。
4. 当审阅者满意时，他们会批准将功能分支的最新版本合并到基础分支中的请求（例如 `develop` ）。

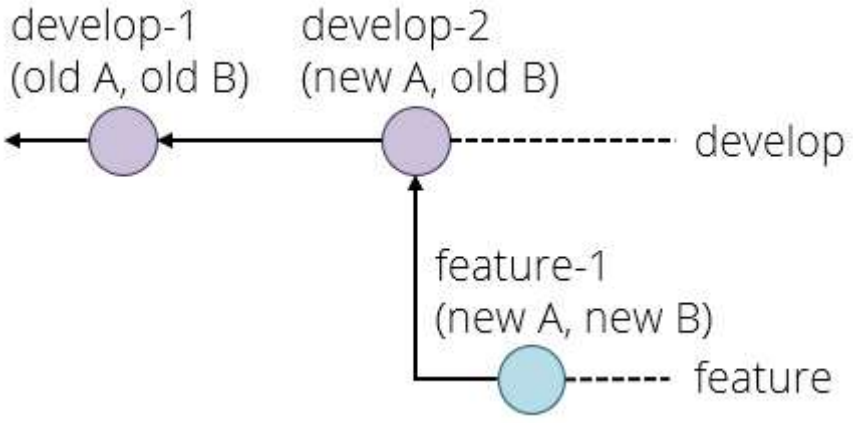
只剩下一个并发症了。假设发生以下情况：

- `develop-1` 您的项目从提交设置开发分支的初始版本开始。假设有两个文件，A 和 B。
- `feature-1` 您创建一个功能分支并进行仅更改文件 B 的提交。
- 与此同时，其他人做了一个更改文件 A 的功能，并将其合并 `develop-2` 到开发分支。

你现在处于 `develop-2` 有（新A，旧B）和你 `feature-1` 有（旧A，新B）的情况。这些都不是您想要的，您可能想要（新 A，新 B）。我们以前也遇到过这种情况，但是没有分支。以图形方式：



这里的解决方案是将您的分支*重新*设置为开发时的最新提交 `git rebase develop` ，并修复由此导致的任何冲突，这会产生以下情况：



如果您现在尝试推送功能分支，您可能会收到错误，因为原始存储库上的功能分支版本仍然是旧版本。这里的解决方案是强制推送，这会覆盖旧版本，

```
git push --force origin BRANCHNAME
```

这是*在输入*某种命令之前要考虑的问题，因为如果您在共享分支上执行此操作，可能会破坏其他开发人员的工作。基本安全规则是：

- 仅在*私有*分支上重新建立基础。
- 仅在绝对必要的情况下强制推送*私有*分支（例如清理变基）。

私有分支是您知道没有其他人正在开发的分支，例如您自己的功能分支。

如果您遇到需要对共享分支（例如开发分支或主分支）进行变基或强制推送的情况，通常需要确保团队中的每个人都知道发生了什么，并在发布之前和之后同步他们的存储库。危险的操作，并且在有人处理它时不会进行任何提交或推送 - 基本上，从并发角度来看，他们在执行此操作时需要对整个存储库进行独占锁定。

这就是主分支和开发分支保持分离的原因之一 - 有些工作流程甚至包括名为 的第三个分支 `release` 。如果合并到主分支或发布分支仅来自开发分支，那么您需要对这些分支进行变基的情况永远不会发生。

总而言之，拉取请求工作流程是：

1. 提交并推动您的更改。
2. 如有必要，请在开发分支上重新建立功能分支。
3. 创建拉取请求。
4. 如有必要，参与讨论或审查，并做出额外承诺来解决其他开发人员提出的问题。
5. 有人（通常不是创建拉取请求的开发人员）批准它，在开发（或主）中创建合并提交。