当你不清楚具体操作时，查阅官方文档是一个不错的选择：

https://git-scm.com/book/zh/v2

本地初始化版本控制仓库

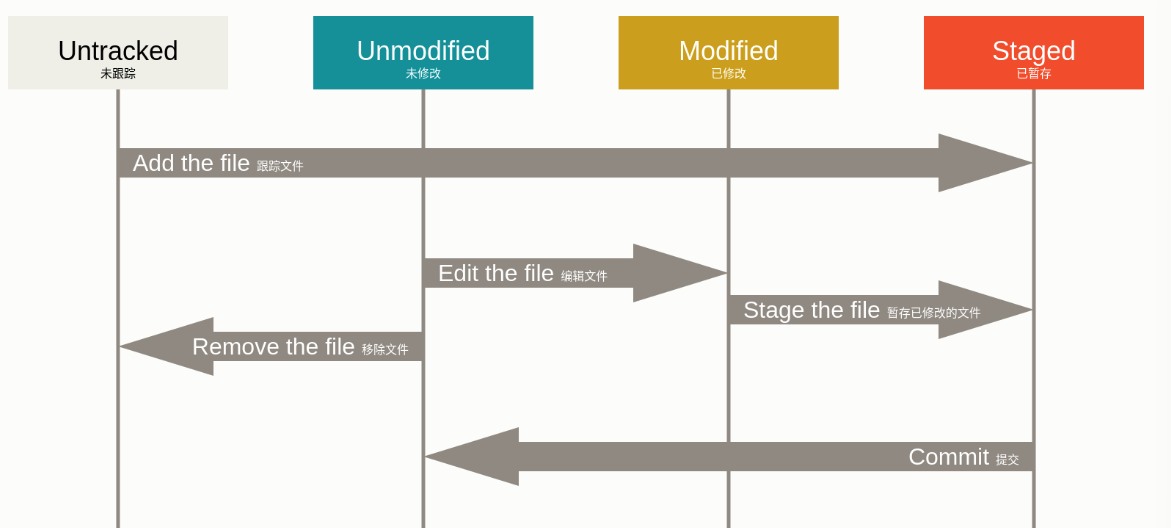
|  |
| --- |
| git init |

克隆仓库

|  |
| --- |
| git clone <url> |

克隆仓库时候自定义仓库名称(目录)

|  |
| --- |
| git clone <url> <name> |

Git仓库文件的四种状态

|  |  |
| --- | --- |
| Untracked(未跟踪) | Git不知道的文件 |
| Unmodified(未修改) |  |
| Modified(已修改) |  |
| Staged(已暂存) |  |

添加跟踪文件(转向暂存区)

|  |
| --- |
| git add <文件名> –——添加目录下的指定文件 |
| git add . –——添加目录下的所有文件 |

可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等

查看仓库状态

|  |
| --- |
| git status |
| git status -s (git status --short) ——更紧凑的输出状态 |

注意： Github上有各种语言与项目的.gitignore文件列表，可以访问

<https://github.com/github/gitignore> 找到它

查看距离上一次提交具体做了哪些更改

|  |
| --- |
| git diff |
| git diff --staged(--cached) ——对比已暂存的文件与最后一次提交的文件差异 |

提交更新

|  |
| --- |
| git commit ——启动文本编辑器输入提交说明 |

可以使用git config --global core.editor命令设置喜欢的编辑器

|  |
| --- |
| git commit -m “提交说明” |

跳过暂存器提交

|  |
| --- |
| git commit -a ——跳过add步骤，直接包含在其中(执行了git add .) |

移除文件

|  |
| --- |
| git rm <文件名> |

下一次提交时，该文件就不再纳入版本管理了。 如果要删除之前修改过或已经放到暂存区的文件，则必须使用强制删除选项 -f（译注：即 force 的首字母）。 这是一种安全特性，用于防止误删尚未添加到快照的数据，这样的数据不能被 Git 恢复。

放弃暂存区的跟踪

|  |
| --- |
| git rm --cached <文件名> |

移动跟踪文件(如果对文件进行更名，git会推断发生了什么)

在Git中对文件改名

|  |
| --- |
| git mv <原来的文件名> <新文件名> |

相当于

|  |
| --- |
| mv <old filename> <new filename> |
| git rm <old filename> |
| git add <new filename> |

* 查看历史提交

|  |
| --- |
| git log |

* 对比每次提交所引入的差异哦

|  |
| --- |
| git log – p(--patch) |

* 限制显示数目

|  |
| --- |
| git log -<限制数目> |

* 查看简略总结信息

|  |
| --- |
| git log --stat |

* 使用不同的格式展示提交历史(--pretty)

|  |
| --- |
| git log --pretty=online ——online(显示一行)的替换参数还有short，full，fuller |

* 定制记录的显示格式(format)

|  |
| --- |
| $ git log --pretty=format:"%h - %an, %ar : %s"  ca82a6d - Scott Chacon, 6 years ago : changed the version number  085bb3b - Scott Chacon, 6 years ago : removed unnecessary test  a11bef0 - Scott Chacon, 6 years ago : first commit |

以下是关于git log --pretty=format的常用选项

| **选项** | **说明** |
| --- | --- |
| %H | 提交的完整哈希值 |
| %h | 提交的简写哈希值 |
| %T | 树的完整哈希值 |
| %t | 树的简写哈希值 |
| %P | 父提交的完整哈希值 |
| %p | 父提交的简写哈希值 |
| %an | 作者名字 |
| %ae | 作者的电子邮件地址 |
| %ad | 作者修订日期（可以用 --date=选项 来定制格式） |
| %ar | 作者修订日期，按多久以前的方式显示 |
| %cn | 提交者的名字 |
| %ce | 提交者的电子邮件地址 |
| %cd | 提交日期 |
| %cr | 提交日期（距今多长时间） |
| %s | 提交说明 |

|  |
| --- |
| 注意：作者指的是实际作出修改的人，提交者指的是最后将此工作成果提交到仓库的人。 所以，当你为某个项目发布补丁，然后某个核心成员将你的补丁并入项目时，你就是作者，而那个核心成员就是提交者。 |

* 当 oneline 或 format 与另一个 log 选项 --graph 结合使用时尤其有用。 这个选项添加了一些 ASCII 字符串来形象地展示你的分支、合并历史：

|  |
| --- |
| $ git log --pretty=format:"%h %s" --graph  \* 2d3acf9 ignore errors from SIGCHLD on trap  \* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit  |\  | \* 420eac9 Added a method for getting the current branch.  \* | 30e367c timeout code and tests  \* | 5a09431 add timeout protection to grit  \* | e1193f8 support for heads with slashes in them  |/  \* d6016bc require time for xmlschema  \* 11d191e Merge branch 'defunkt' into local |

* git log的常用选项

| **选项** | **说明** |
| --- | --- |
| -p | 按补丁格式显示每个提交引入的差异。 |
| --stat | 显示每次提交的文件修改统计信息。 |
| --shortstat | 只显示 --stat 中最后的行数修改添加移除统计。 |
| --name-only | 仅在提交信息后显示已修改的文件清单。 |
| --name-status | 显示新增、修改、删除的文件清单。 |
| --abbrev-commit | 仅显示 SHA-1 校验和所有 40 个字符中的前几个字符。 |
| --relative-date | 使用较短的相对时间而不是完整格式显示日期（比如“2 weeks ago”）。 |
| --graph | 在日志旁以 ASCII 图形显示分支与合并历史。 |
| --pretty | 使用其他格式显示历史提交信息。可用的选项包括 oneline、short、full、fuller 和 format（用来定义自己的格式）。 |
| --oneline | --pretty=oneline --abbrev-commit 合用的简写。 |

* 使--since --until这种可以按照时间做限制

例如，列出最近两周的所有的提交

|  |
| --- |
| git log --since =2.weeks |

改名了也可是具体某一天“2008-01-15”,也可以是“2 year 1 day 3 minutes ago”对应日期

* 过滤搜索条件

用 --author 选项显示指定作者的提交，用 --grep 选项搜索提交说明中的关键字

强大的过滤器-S，它接受一个字符串参数，并且只会显示那些添加或删除了该字符串的提交

|  |
| --- |
| git log -S function\_name |

* 查阅路径(path)：如果只关心某些文件或者目录的历史提交，可以在 git log 选项的最后指定它们的路径。 因为是放在最后位置上的选项，所以用两个短划线（--）隔开之前的选项和后面限定的路径名。（笔者也没搞懂这个指令）

限制git log 输出选项

| **选项** | **说明** |
| --- | --- |
| -<n> | 仅显示最近的 n 条提交。 |
| --since, --after | 仅显示指定时间之后的提交。 |
| --until, --before | 仅显示指定时间之前的提交。 |
| --author | 仅显示作者匹配指定字符串的提交。 |
| --committer | 仅显示提交者匹配指定字符串的提交。 |
| --grep | 仅显示提交说明中包含指定字符串的提交。 |
| -S | 仅显示添加或删除内容匹配指定字符串的提交。 |

注意：按照你代码仓库的工作流程，记录中可能有为数不少的合并提交，它们所包含的信息通常并不多。 为了避免显示的合并提交弄乱历史记录，可以为 log 加上 --no-merges 选项。

Git——撤销操作

* 有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 --amend 选项的提交命令来重新提交

|  |
| --- |
| git commit --amend; |

例如，你提交后发现忘记了暂存某些需要的修改，可以像下面这样操作：

|  |
| --- |
| git commit -m “initial commit”  git add forgotten\_file  git commit –amend |

这里只会有一个提交，第二次提交将替代第一次提交的结果。

* 取消暂存文件(当你已经add文件到上传去，且未commit)

|  |
| --- |
| git reset HEAD <file> ---取消文件的暂存 |

|  |
| --- |
| 注意：git reset 确实是个危险的命令，如果加上了 --hard 选项则更是如此。在Git官方文档“重置揭秘”会聊到更多reset细节 |

撤销对文件的修改(将文件还原到上一次提交的状态)

|  |
| --- |
| git checkout -- <file> |

|  |
| --- |
| 注意：请务必记得 git checkout — <file> 是一个危险的命令。 你对那个文件在本地的任何修改都会消失——Git 会用最近提交的版本覆盖掉它。 除非你确实清楚不想要对那个文件的本地修改了，否则请不要使用这个命令。 |

Git——远程仓库的使用

为了能在任意 Git 项目上协作，你需要知道如何管理自己的远程仓库。 远程仓库是指托管在因特网或其他网络中的你的项目的版本库。 你可以有好几个远程仓库，通常有些仓库对你只读，有些则可以读写。 与他人协作涉及管理远程仓库以及根据需要推送或拉取数据。 管理远程仓库包括了解如何添加远程仓库、移除无效的远程仓库、管理不同的远程分支并定义它们是否被跟踪等等。

|  |
| --- |
| 注意： 你完全可以在一个“远程”仓库上工作，而实际上它在你本地的主机上。 词语“远程”未必表示仓库在网络或互联网上的其它位置，而只是表示它在别处。 在这样的远程仓库上工作，仍然需要和其它远程仓库上一样的标准推送、拉取和抓取操作。 |

* 查看远程仓库名称

|  |
| --- |
| git remote -v |

如果你的远程仓库不止一个，该命令会将它们全部列出。

这表示我们能非常方便地拉取其它用户的贡献。我们还可以拥有向他们推送的权限

* 添加远程仓库

|  |
| --- |
| git remote add <shortname> <url> |

例如：

|  |
| --- |
| git remote add origin https;//… |

现在可以在命令行使用origin代表整个URL

例如拉取origin仓库中有，而我本地没有的信息，可以运行git fetch origin；

|  |
| --- |
| 注意：git fetch命令只会将数据下载到你的本地仓库——它并不会自动合并或修改你当前的工作 |

git clone命令会获取远程的所有分支，所有文件，但只会自动设置本地master与远程仓库的master关联，并只显示master

* 推送到远程仓库

|  |
| --- |
| git push <remote> <branch> |

* 查看某个远程仓库

|  |
| --- |
| git remote show <remote> |

例如

|  |
| --- |
| $ git remote show origin  \* remote origin  Fetch URL: https://github.com/schacon/ticgit  Push URL: https://github.com/schacon/ticgit  HEAD branch: master  Remote branches:  master tracked  dev-branch tracked  Local branch configured for 'git pull':  master merges with remote master  Local ref configured for 'git push':  master pushes to master (up to date) |

它同样会列出远程仓库的 URL 与跟踪分支的信息。 这些信息非常有用，它告诉你正处于 master 分支，并且如果运行 git pull， 就会抓取所有的远程引用，然后将远程 master 分支合并到本地 master 分支。 它也会列出拉取到的所有远程引用。

* 查看更多远程仓库信息

|  |
| --- |
| git remote show <remote name> |

|  |
| --- |
| $ git remote show origin  \* remote origin  URL: https://github.com/my-org/complex-project  Fetch URL: https://github.com/my-org/complex-project  Push URL: https://github.com/my-org/complex-project  HEAD branch: master  Remote branches:  master tracked  dev-branch tracked  markdown-strip tracked  issue-43 new (next fetch will store in remotes/origin)  issue-45 new (next fetch will store in remotes/origin)  refs/remotes/origin/issue-11 stale (use 'git remote prune' to remove)  Local branches configured for 'git pull':  dev-branch merges with remote dev-branch  master merges with remote master  Local refs configured for 'git push':  dev-branch pushes to dev-branch (up to date)  markdown-strip pushes to markdown-strip (up to date)  master pushes to master (up to date) |

这个命令列出了当你在特定的分支上执行 git push 会自动地推送到哪一个远程分支。 它也同样地列出了哪些远程分支不在你的本地，哪些远程分支已经从服务器上移除了， 还有当你执行 git pull 时哪些本地分支可以与它跟踪的远程分支自动合并。

* 远程仓库的重命名与移除

你可以运行 git remote rename 来修改一个远程仓库的简写名(远程仓库服务器不受影响)

|  |
| --- |
| git remote rename  <old name> <new name> |

如果因为一些原因想要移除一个远程仓库——你已经从服务器上搬走了或不再想使用某一个特定的镜像了， 又或者某一个贡献者不再贡献了——可以使用 git remote remove 或 git remote rm ：（本地文件均不会删除，只是你们之间的连接断开了）

|  |
| --- |
| git remote remove paul |

断开与远程paul仓库的连接并移除该远程仓库，所有和这个远程仓库相关的远程跟踪分支以及配置信息也会一起被删除。

分支(杀手级特性)

* 创建分支

|  |
| --- |
| git branch <branch name> |

这会在当前提交对象上创建一个新的指针指向当前对象

Git通过HEAD指针知道当前是哪个分支

可以通过命令查看当前分支所指向的对象

|  |
| --- |
| git log --oneline --decorate |

* 切换到已存在的分支

|  |
| --- |
| git checkout <branch name> |

这样HEAD就指向新的分支上了。

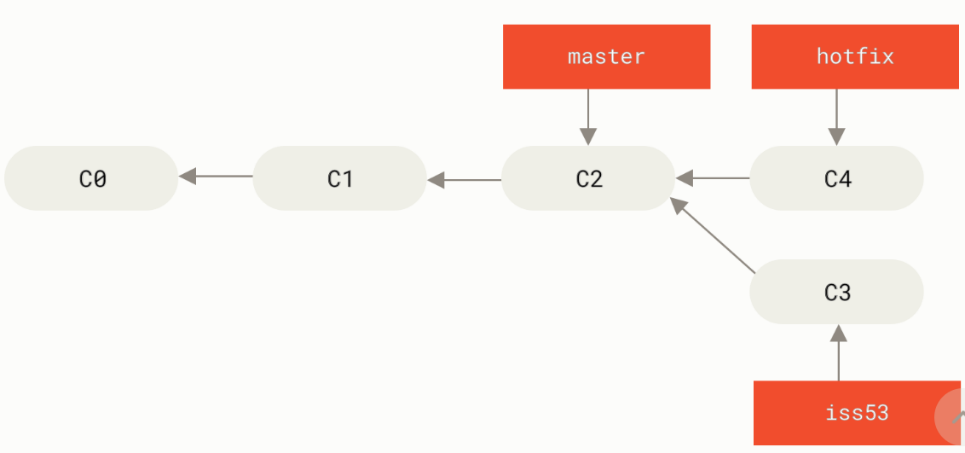
* 创建分支并切换到分支上

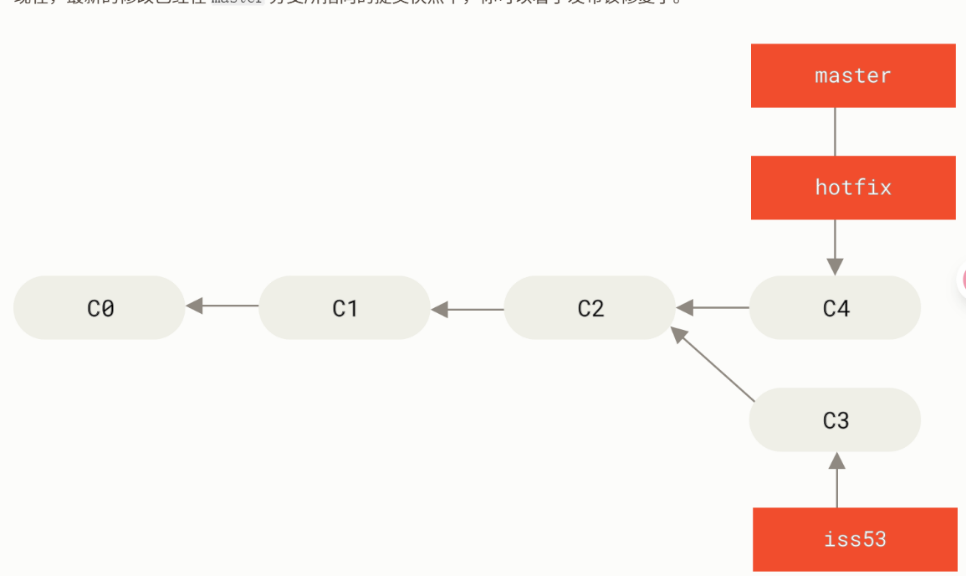
|  |
| --- |
| git checkout -b <new branch name> |

分支的新建与合并

* 合并

|  |
| --- |
| git merge <被合并的分支名> |

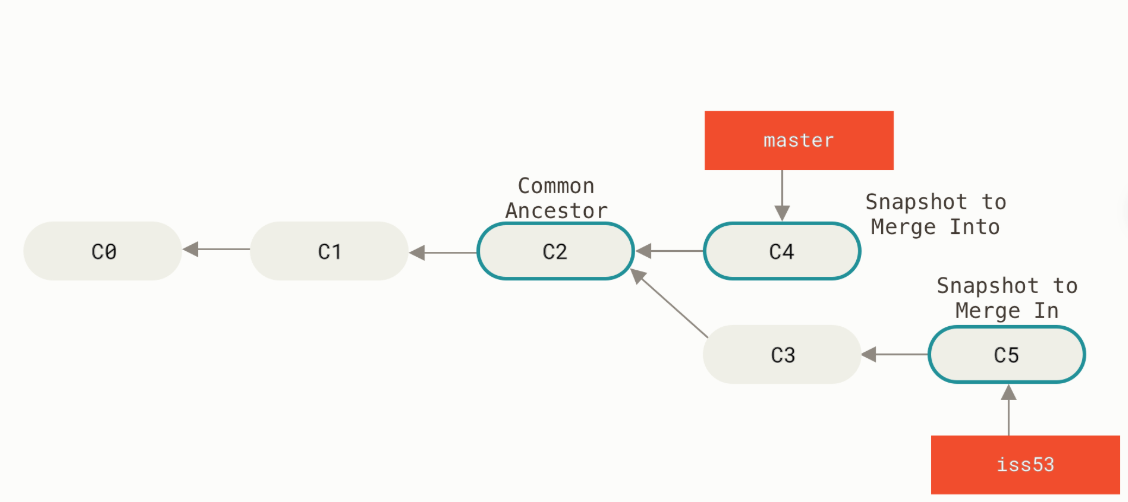
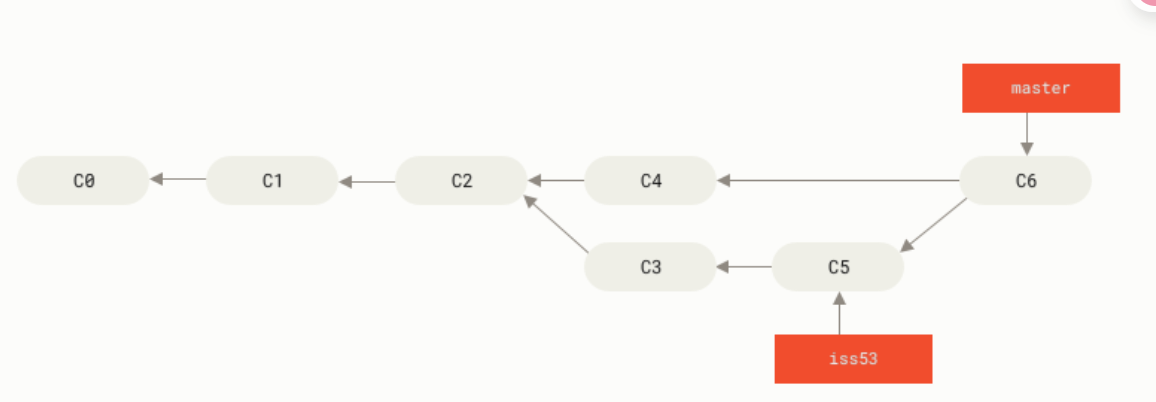
默认会使用“fast-forward”合并，当你试图合并两个分支时， 如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候， 只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧——这就叫做 “快进（fast-forward）”。

这里合并的master与hotfix分支

删除分支

|  |
| --- |
| git branch -d <branch name> |

当你想将iss53合并到master中时，这和你之前合并 hotfix 分支的时候看起来有一点不一样。 在这种情况下，你的开发历史从一个更早的地方开始分叉开来（diverged）。 因为，master 分支所在提交并不是 iss53 分支所在提交的直接祖先，Git 不得不做一些额外的工作。 出现这种情况的时候，Git 会使用两个分支的末端所指的快照（C4 和 C5）以及这两个分支的公共祖先（C2），做一个简单的三方合并。

和之前将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。 这个被称作一次合并提交，它的特别之处在于他有不止一个父提交

**遇到从冲突时的合并分支**

有时候合并操作不会如此顺利。 如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。

|  |
| --- |
| $ git merge iss53  Auto-merging index.html  CONFLICT (content): Merge conflict in index.html  Automatic merge failed; fix conflicts and then commit the result. |

此时 Git 做了合并，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，等待你去解决合并产生的冲突。 你可以在合并冲突后的任意时刻使用 git status 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

|  |
| --- |
| $ git status  On branch master  You have unmerged paths.  (fix conflicts and run "git commit")  Unmerged paths:  (use "git add <file>..." to mark resolution)  both modified: index.html  no changes added to commit (use "git add" and/or "git commit -a") |

如果你想使用图形化工具来解决冲突，你可以运行 git mergetool，该命令会为你启动一个合适的可视化合并工具，并带领你一步一步解决这些冲突。

解决完冲突后，通过git add 以及 git commit 完成合并提交。

**解决冲突的方法：**

1. 可视化工具
2. 手动

* 查阅分支列表

|  |
| --- |
| git branch |

* 查阅每个分支最后的提交

|  |
| --- |
| git branch -v |

* 查阅过滤以及合并或未合并到当前分支的分支(--merged 与 --no-merged)

|  |
| --- |
| git branch --merged |
| git branch --no-merged |

Tip:

上面描述的选项 --merged 和 --no-merged 会在没有给定提交或分支名作为参数时， 分别列出已合并或未合并到 **当前** 分支的分支。

你总是可以提供一个附加的参数来查看其它分支的合并状态而不必检出它们。 例如，尚未合并到 master 分支的有哪些？

|  |
| --- |
| $ git checkout testing  $ git branch --no-merged master  topicA  featureB |

* 如果一个分支没有完成合并工作，那么删掉它会失败，如果非要删掉

|  |
| --- |
| git branch -D <branch name> |

远程分支

**远程引用**是对远程仓库的引用（指针），包括分支、标签等等。 你可以通过 git ls-remote <remote> 来显式地获得远程引用的完整列表， 或者通过 git remote show <remote> 获得远程分支的更多信息。 然而，一个更常见的做法是利用**远程跟踪分支**。

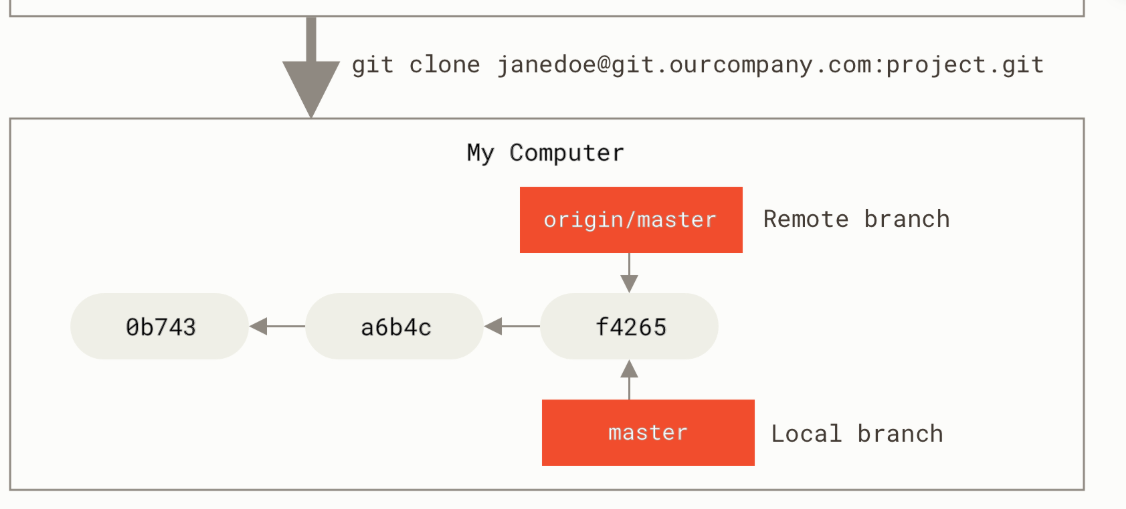
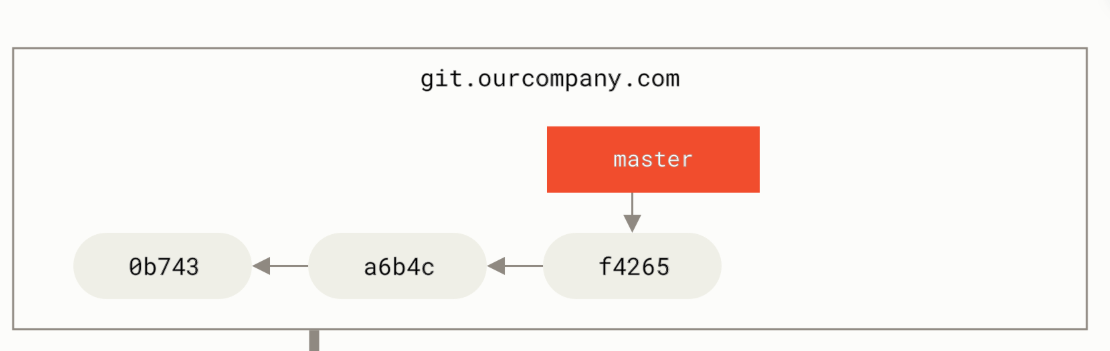
**远程跟踪分支**是远程分支状态的引用。它们是你无法移动的本地引用。一旦你进行了网络通信， Git 就会为你移动它们以精确反映远程仓库的状态。请将它们看做书签， 这样可以提醒你该分支在远程仓库中的位置就是你最后一次连接到它们的位置。

* **远程跟踪分支的命名形式**： <remote>/<branch>

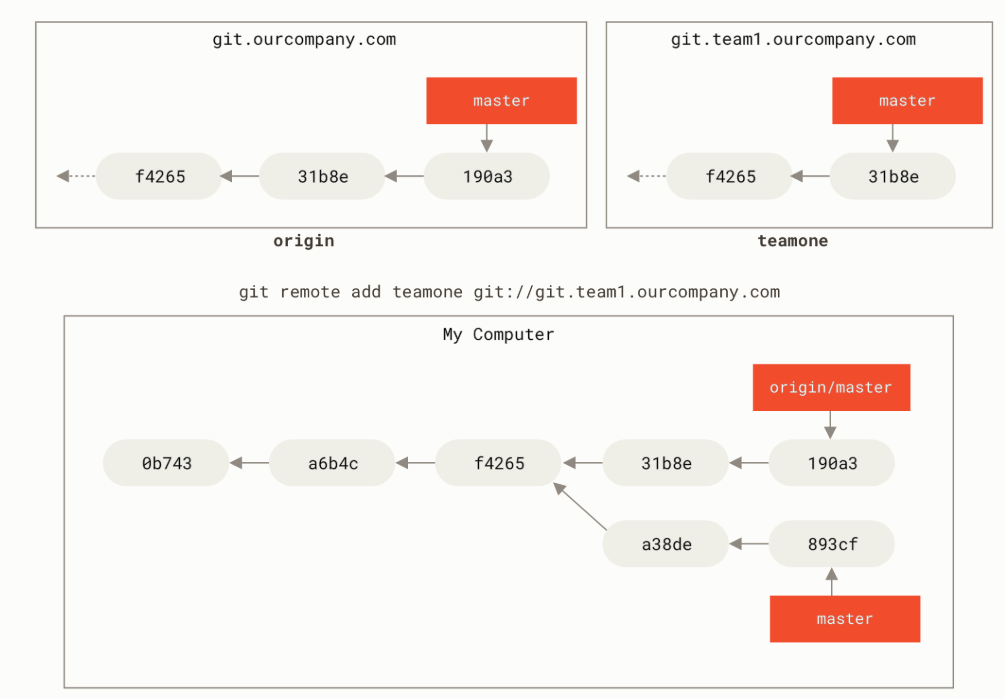
  例如，如果你想要看你最后一次与远程仓库 origin 通信时 master 分支的状态，你可以查看 origin/master 分支。 你与同事合作解决一个问题并且他们推送了一个 iss53 分支，你可能有自己的本地 iss53 分支， 然而在服务器上的分支会以 origin/iss53 来表示。

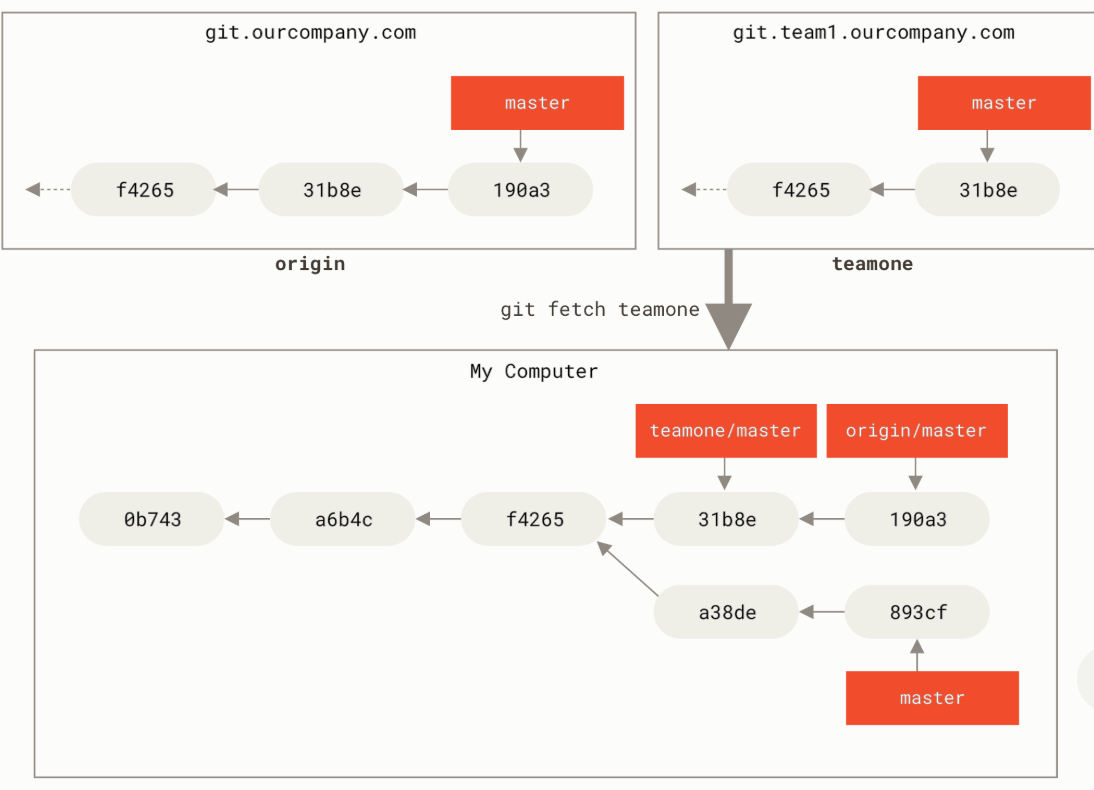
 假设你的网络里有一个在 git.ourcompany.com 的 Git 服务器。 如果你从这里克隆，Git 的 clone 命令会为你自动将其命名为 origin，拉取它的所有数据， 创建一个指向它的 master 分支的指针，并且在本地将其命名为 origin/master。 Git 也会给你一个与 origin 的 master 分支在指向同一个地方的本地 master 分支，这样你就有工作的基础。

这很好理解，当你从远程拉取了项目下来，git默认为远程分支创建了一个master指针，随后在本地为你创建了一个master指针与指向远程master的origin/master指针。



* git fetch 更新你的远程跟踪分支

为了演示有多个远程仓库与远程分支的情况，我们假定你有另一个内部 Git 服务器，仅服务于你的某个敏捷开发团队。 这个服务器位于 git.team1.ourcompany.com。 你可以运行 git remote add 命令添加一个新的远程仓库引用到当前的项目，将这个远程仓库命名为 teamone，将其作为完整 URL 的缩写。

现在，可以运行 git fetch teamone 来抓取远程仓库 teamone 有而本地没有的数据。 因为那台服务器上现有的数据是 origin 服务器上的一个子集， 所以 Git 并不会抓取数据而是会设置远程跟踪分支 teamone/master 指向 teamone 的 master 分支。

* 推送本地分支更新远程仓库上的分支(如果远程仓库有该分支，则需要确保有远程引用；如果没有该分支，则会自动创建分支并关联)

|  |
| --- |
| git push origin <branch name> |

* 推送本地分支到远程分支上：推送本地的 serverfix 分支，将其作为远程仓库的 serverfix 分支(如果远程仓库有该分支，则需要确保有远程引用；如果没有该分支，则会自动创建分支并关联)

|  |
| --- |
| git push origin serverfix: serverfix |

如果你不喜欢这个远程仓库的分支有这个名字，也可以改

|  |
| --- |
| git push origin serverfix: YangBranch |

下一次其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支 origin/serverfix，指向服务器的 serverfix 分支的引用：

|  |
| --- |
| $ git fetch origin  remote: Counting objects: 7, done.  remote: Compressing objects: 100% (2/2), done.  remote: Total 3 (delta 0), reused 3 (delta 0)  Unpacking objects: 100% (3/3), done.  From https://github.com/schacon/simplegit  \* [new branch] serverfix -> origin/serverfix |

|  |
| --- |
| 注意：当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本（拷贝）。 换一句话说，这种情况下，不会有一个新的 serverfix 分支——只有一个不可以修改的 origin/serverfix 指针。  可以运行 git merge origin/serverfix 将这些工作合并到当前所在的分支。 如果想要在自己的 serverfix 分支上工作，可以将其建立在远程跟踪分支之上：  $ git checkout -b serverfix origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix'  这会给你一个用于工作的本地分支，并且起点位于 origin/serverfix。 |

**跟踪分支**

从一个远程跟踪分支检出一个本地分支会自动创建所谓的“跟踪分支”（它跟踪的分支叫做“上游分支”）。 跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 git pull，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

当克隆一个仓库时，它通常会自动地创建一个跟踪 origin/master 的 master 分支。 然而，如果你愿意的话可以设置其他的跟踪分支，或是一个在其他远程仓库上的跟踪分支，又或者不跟踪 master 分支。 最简单的实例就是像之前看到的那样，运行 git checkout -b <branch> <remote>/<branch>。 这是一个十分常用的操作所以 Git 提供了 --track 快捷方式：

|  |
| --- |
| $ git checkout --track origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

由于这个操作太常用了，该捷径本身还有一个捷径。 如果你尝试检出的分支 (a) 不存在且 (b) 刚好只有一个名字与之匹配的远程分支，那么 Git 就会为你创建一个跟踪分支：

|  |
| --- |
| $ git checkout serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

如果想要将本地分支与远程分支设置为不同的名字，你可以轻松地使用上一个命令增加一个不同名字的本地分支：

|  |
| --- |
| $ git checkout -b sf origin/serverfix  Branch sf set up to track remote branch serverfix from origin.  Switched to a new branch 'sf' |

现在，本地分支 sf 会自动从 origin/serverfix 拉取。

设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支， 你可以在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch 来显式地设置。

|  |
| --- |
| $ git branch -u origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin. |

如果想要查看设置的所有跟踪分支，可以使用 **git branch**的 **-vv** 选项。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。

|  |
| --- |
| $ git branch -vv  iss53 7e424c3 [origin/iss53: ahead 2] forgot the brackets  master 1ae2a45 [origin/master] deploying index fix  \* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it  testing 5ea463a trying something new |

这里可以看到 iss53 分支正在跟踪 origin/iss53 并且 “ahead” 是 2，意味着本地有两个提交还没有推送到服务器上。 也能看到 master 分支正在跟踪 origin/master 分支并且是最新的。 接下来可以看到 serverfix 分支正在跟踪 teamone 服务器上的 server-fix-good 分支并且领先 3 落后 1， 意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。 最后看到 testing 分支并没有跟踪任何远程分支。

需要重点注意的一点是这些数字的值来自于你从每个服务器上最后一次抓取的数据。 这个命令并没有连接服务器，它只会告诉你关于本地缓存的服务器数据。 如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。 可以像这样做：

|  |
| --- |
| $ git fetch --all; git branch -vv |

**拉取**

当 git fetch 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。 它只会获取数据然后让你自己合并。 然而，有一个命令叫作 git pull 在大多数情况下它的含义是一个 git fetch 紧接着一个 git merge 命令。 如果有一个像之前章节中演示的设置好的跟踪分支，不管它是显式地设置还是通过 clone 或 checkout 命令为你创建的，git pull 都会查找当前分支所跟踪的服务器与分支， 从服务器上抓取数据然后尝试合并入那个远程分支。

由于 git pull 的魔法经常令人困惑所以通常单独显式地使用 fetch 与 merge 命令会更好一些。

**删除远程分支**

假设你已经通过远程分支做完所有的工作了——也就是说你和你的协作者已经完成了一个特性， 并且将其合并到了远程仓库的 master 分支（或任何其他稳定代码分支）。 可以运行带有 --delete 选项的 git push 命令来删除一个远程分支。 如果想要从服务器上删除 serverfix 分支，运行下面的命令：

|  |
| --- |
| $ git push origin --delete serverfix  To https://github.com/schacon/simplegit  - [deleted] serverfix |

基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

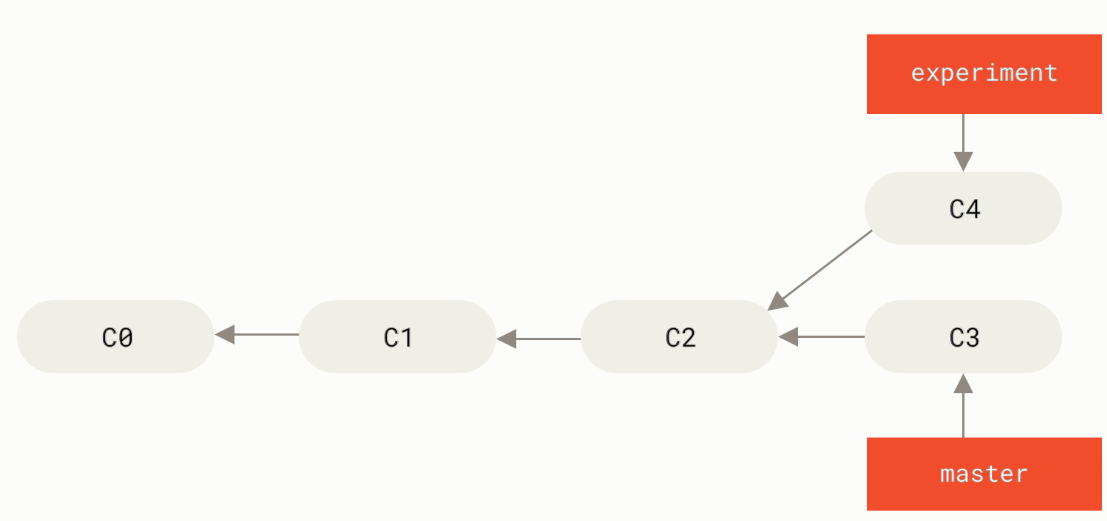
我知道这很难理解，慢慢消化把，时间会让你明白的

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*杀手特性：难懂，难用，难办\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

变基

**变基**

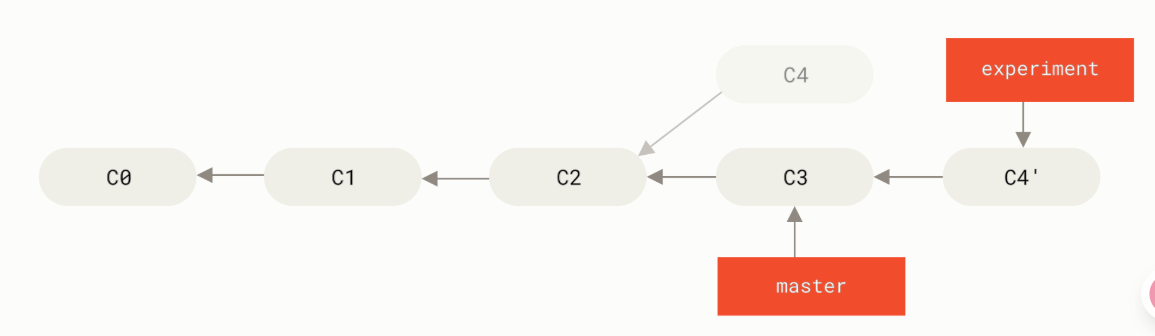
在 Git 中整合来自不同分支的修改主要有两种方法：**merge** 以及 **rebase**。 在本节中我们将学习什么是“变基”，怎样使用“变基”，并将展示该操作的惊艳之处，以及指出在何种情况下你应避免使用它。

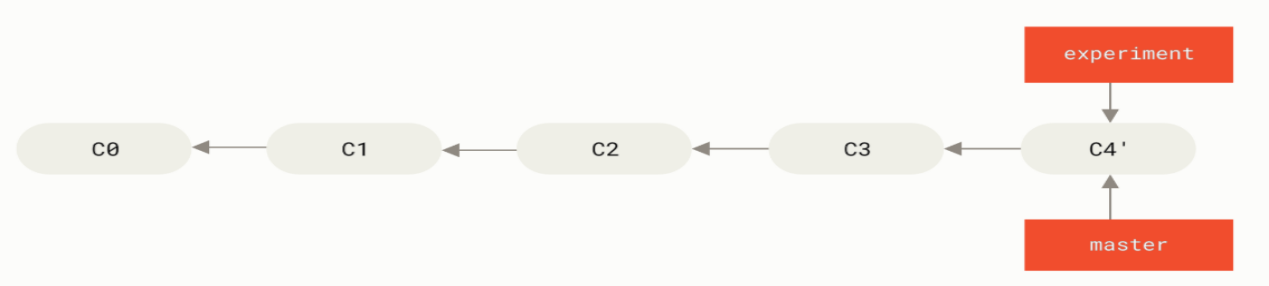
回顾之前的合并例子，这里使用的是merge命令， 它会把两个分支的最新快照（C3 和 C4）以及二者最近的共同祖先（C2）进行三方合并，合并的结果是生成一个新的快照（并提交）。

其实，还有一种方法：你可以提取在 C4 中引入的补丁和修改，然后在 C3 的基础上应用一次。 在 Git 中，这种操作就叫做 **变基（rebase）**。 你可以使用 rebase 命令将提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样**(变基会直接移动指针的位置)**

在这个例子中，你可以检出 experiment 分支，然后将它变基到 master 分支上：

|  |
| --- |
| $ git checkout experiment  $ git rebase master  First, rewinding head to replay your work on top of it...  Applying: added staged command |

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。

现在回到master分支，进行一次快速合并

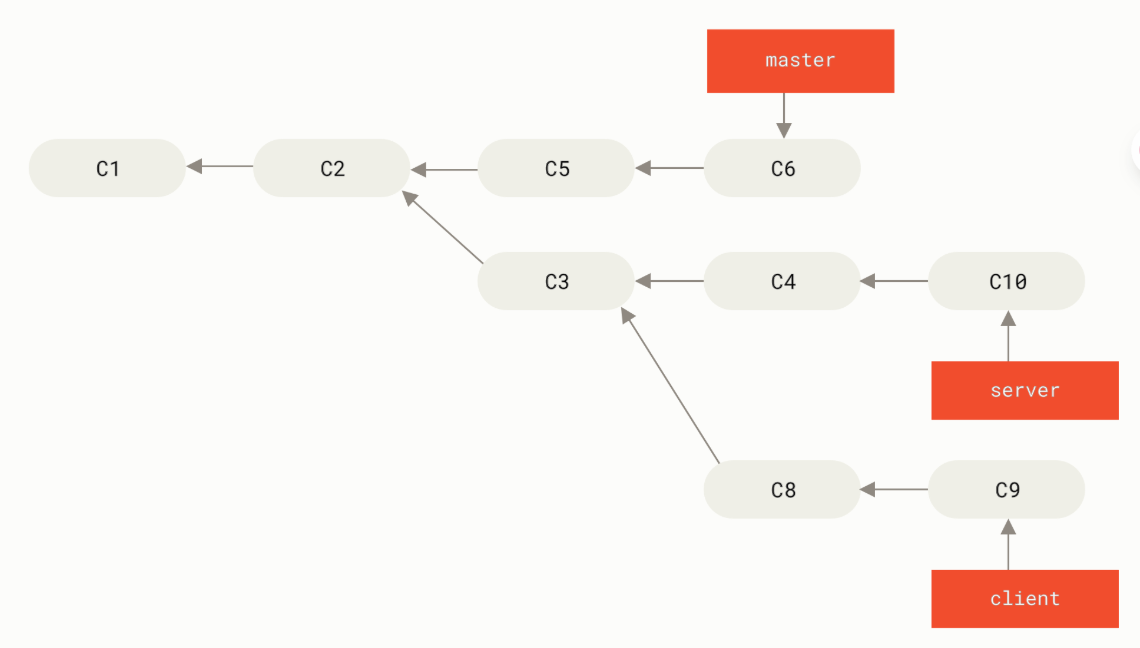
|  |
| --- |
| $ git checkout master  $ git merge experiment |

此时，C4' 指向的快照就和 [the merge example](https://git-scm.com/book/zh/v2/ch00/ebasing-merging-example) 中 C5 指向的快照一模一样了。 这两种整合方法的最终结果没有任何区别，但是变基使得提交历史更加整洁。 你在查看一个经过变基的分支的历史记录时会发现，尽管实际的开发工作是并行的， 但它们看上去就像是串行的一样，提交历史是一条直线没有分叉。

一般我们这样做的目的是为了确保在向远程分支推送时能保持提交历史的整洁——例如向某个其他人维护的项目贡献代码时。 在这种情况下，你首先在自己的分支里进行开发，当开发完成时你需要先将你的代码变基到 origin/master 上，然后再向主项目提交修改。 这样的话，该项目的维护者就不再需要进行整合工作，只需要快进合并便可。

请注意，无论是通过变基，还是通过三方合并，**整合的最终结果所指向的快照始终是一样的，只不过提交历史不同罢了。** 变基是将一系列提交按照原有次序依次应用到另一分支上，而合并是把最终结果合在一起。

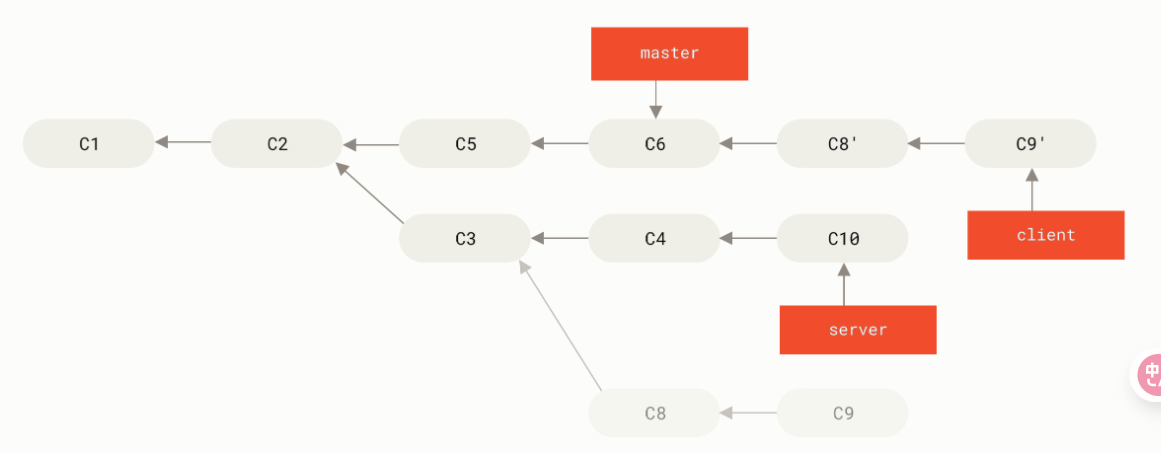
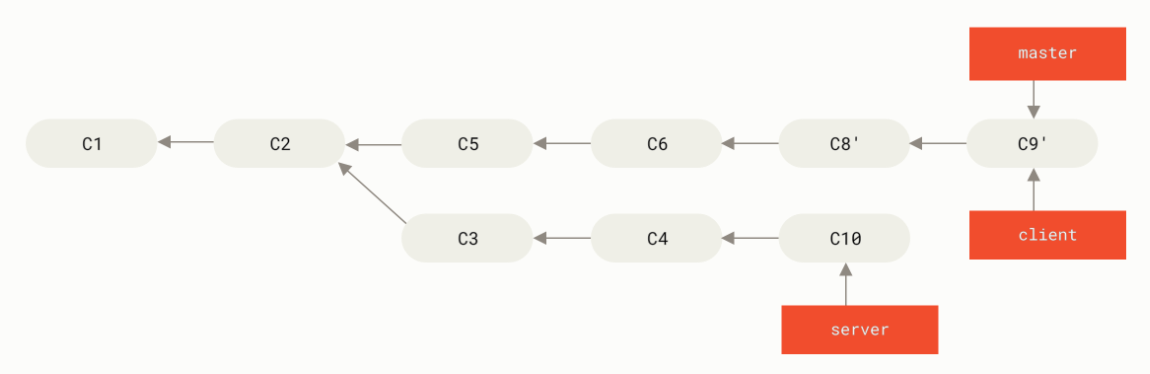
**更有趣的变基例子**

 你创建了一个主题分支 server，为服务端添加了一些功能，提交了 C3 和 C4。 然后从 C3 上创建了主题分支 client，为客户端添加了一些功能，提交了 C8 和 C9。 最后，你回到 server 分支，又提交了 C10。

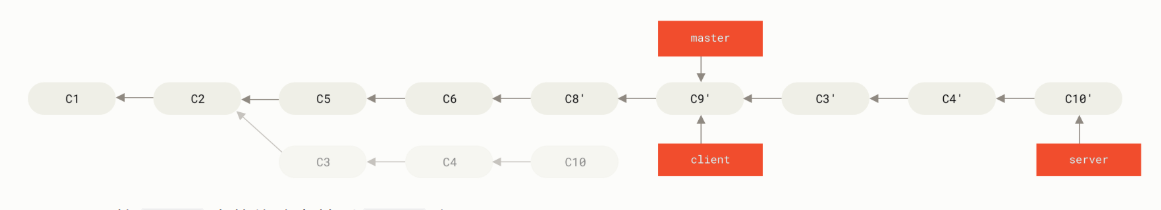
假设你希望将 client 中的修改合并到主分支并发布，但暂时并不想合并 server 中的修改， 因为它们还需要经过更全面的测试。这时，你就可以使用 git rebase 命令的 --onto 选项， 选中在 client 分支里但不在 server 分支里的修改（即 C8 和 C9），将它们在 master 分支上重放：

|  |
| --- |
| git **rebase --onto** master server client |

以上命令的意思是：“取出 client 分支，找出它从 server 分支分歧之后的补丁， 然后把这些补丁在 master 分支上重放一遍，让 client 看起来像直接基于 master 修改一样”。这理解起来有一点复杂，不过效果非常酷。

现在可以快进合并 master 分支了。（如图 [快进合并 master 分支，使之包含来自 client 分支的修改](https://git-scm.com/book/zh/v2/ch00/bdiag_g)）：

|  |
| --- |
| $ git checkout master  $ git merge client |

接下来你决定将 server 分支中的修改也整合进来。 使用 git rebase <basebranch> <topicbranch> 命令可以直接将主题分支 （即本例中的 server）变基到目标分支（即 master）上。 这样做能省去你先切换到 server 分支，再对其执行变基命令的多个步骤。

|  |
| --- |
| git rebase master server |

如图 将 server 中的修改变基到 master 上 所示，server 中的代码被“续”到了 master 后面。

然后就可以快进合并主分支 master 了：

|  |
| --- |
| $ git checkout master  $ git merge server |

至此，client 和 server 分支中的修改都已经整合到主分支里了， 你可以删除这两个分支，最终提交历史会变成图 [最终的提交历史](https://git-scm.com/book/zh/v2/ch00/bdiag_i) 中的样子：

|  |
| --- |
| $ git branch -d client  $ git branch -d server |

**变基的风险**

呃，奇妙的变基也并非完美无缺，要用它得遵守一条准则：

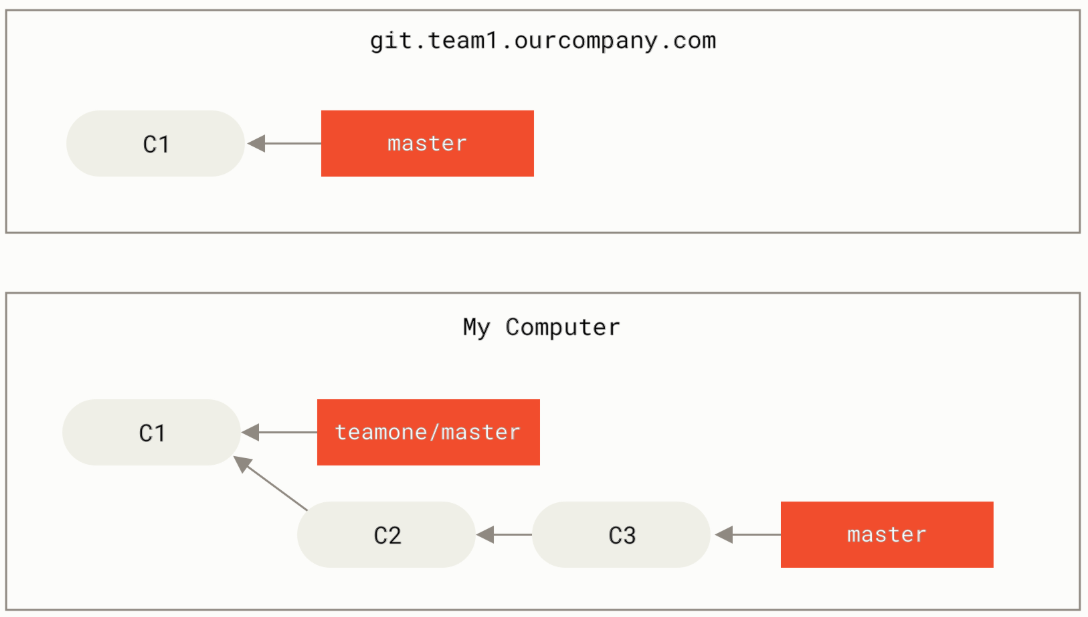
**如果提交存在于你的仓库之外，而别人可能基于这些提交进行开发，那么不要执行变基。**

**如果你遵循这条金科玉律，就不会出差错。 否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你！！**！

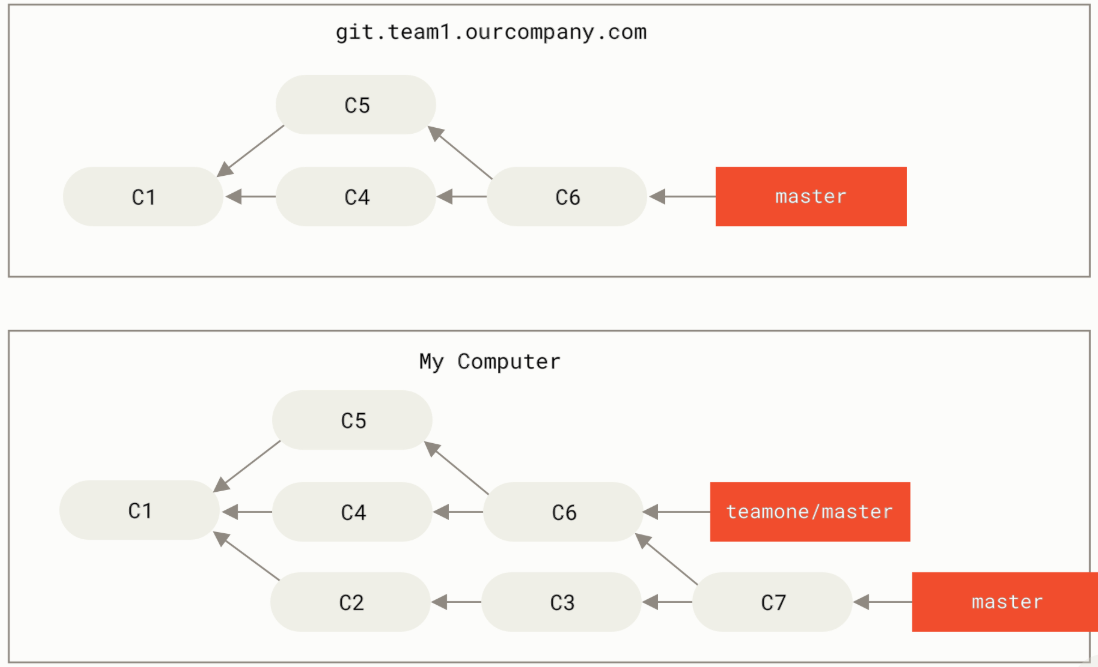
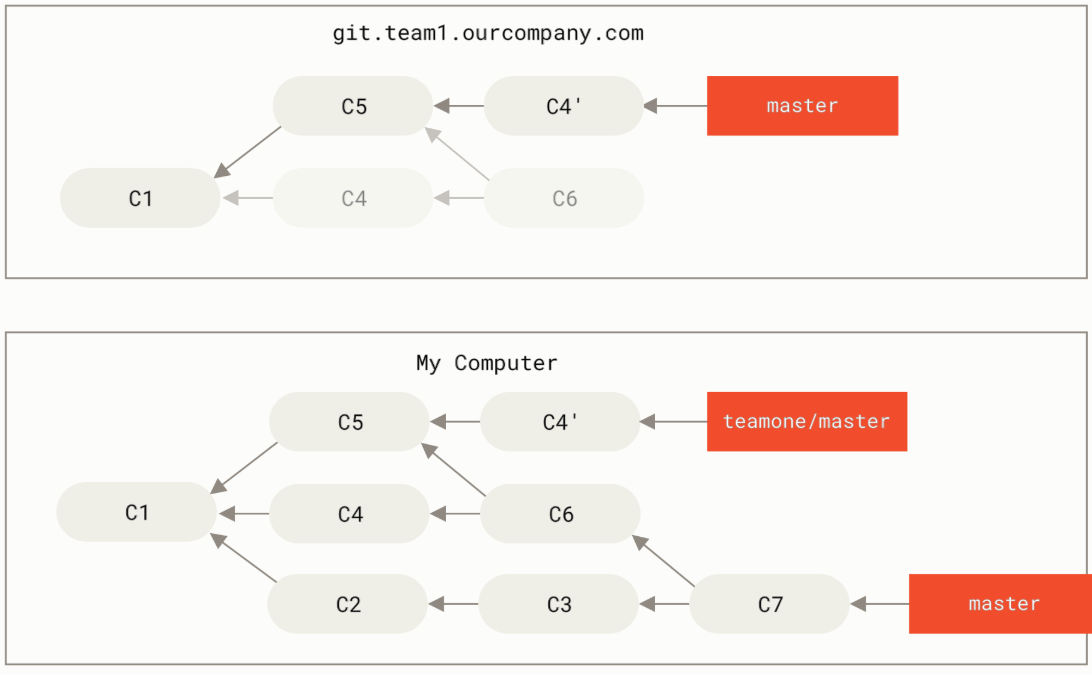
**变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。** 如果你已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果你用 git rebase 命令重新整理了提交并再次推送，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。

* 强制上传

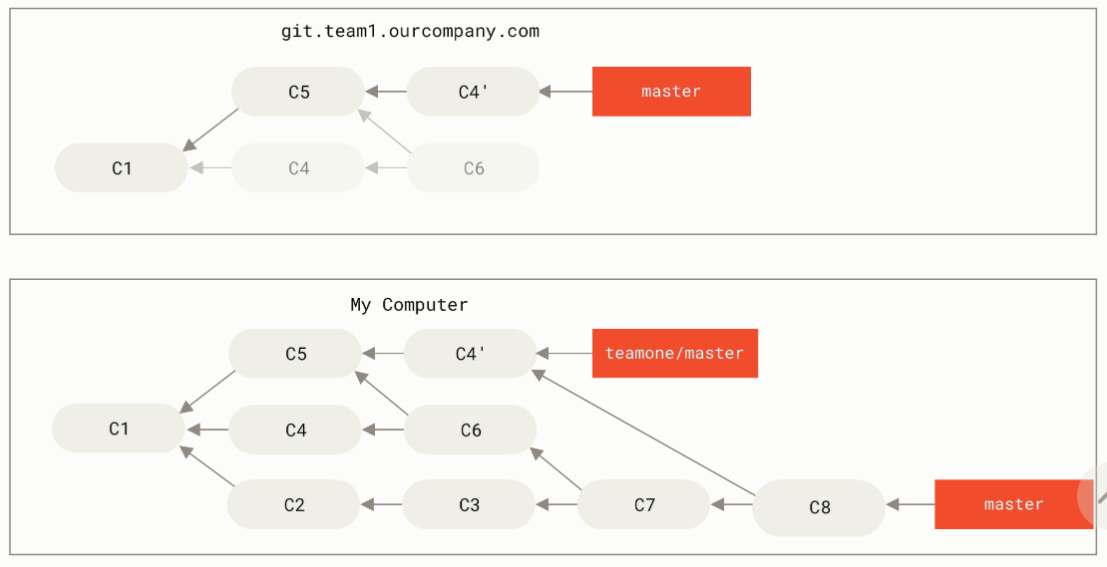
|  |
| --- |
| git push --force |

让我们来看一个在公开的仓库上执行变基操作所带来的问题。 假设你从一个中央服务器克隆然后在它的基础上进行了一些开发。 你的提交历史如图所示：

然后，某人又向中央服务器提交了一些修改，其中还包括一次合并。 你抓取了这些在远程分支上的修改，并将其合并到你本地的开发分支，然后你的提交历史就会变成这样：

接下来，这个人又决定把合并操作回滚，改用变基；继而又用 git push --force 命令覆盖了服务器上的提交历史。 之后你从服务器抓取更新，会发现多出来一些新的提交。

结果就是你们两人的处境都十分尴尬。 如果你执行 git pull 命令，你将合并来自两条提交历史的内容，生成一个新的合并提交，最终仓库会如图所示：

此时如果你执行 git log 命令，你会发现有两个提交的作者、日期、日志居然是一样的，这会令人感到混乱。 此外，如果你将这一堆又推送到服务器上，你实际上是将那些已经被变基抛弃的提交又找了回来，这会令人感到更加混乱。 很明显对方并不想在提交历史中看到 C4 和 C6，因为之前就是他把这两个提交通过变基丢弃的。

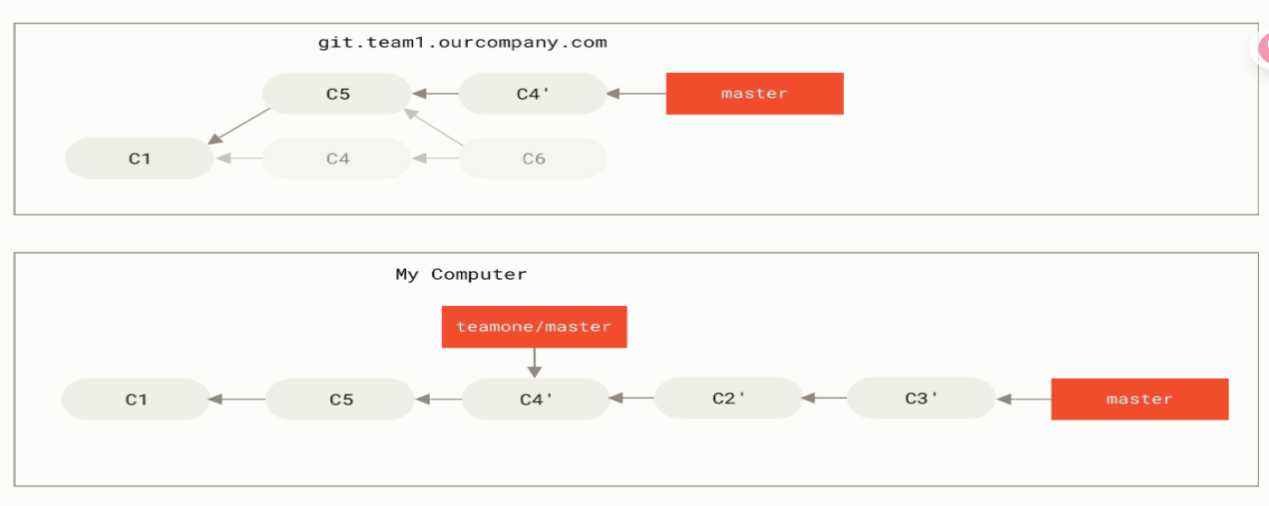
用变基解决变基

如果你 **真的** 遭遇了类似的处境，Git 还有一些高级魔法可以帮到你。 如果团队中的某人强制推送并覆盖了一些你所基于的提交，你需要做的就是检查你做了哪些修改，以及他们覆盖了哪些修改。

如果你拉取被覆盖过的更新并将你手头的工作基于此进行变基的话，一般情况下 Git 都能成功分辨出哪些是你的修改，并把它们应用到新分支上。

举个例子，如果遇到前面提到的 [有人推送了经过变基的提交，并丢弃了你的本地开发所基于的一些提交](https://git-scm.com/book/zh/v2/ch00/_pre_merge_rebase_work) 那种情境，如果我们不是执行合并，而是执行 git rebase teamone/master, Git 将会：

* 检查哪些提交是我们的分支上独有的（C2，C3，C4，C6，C7）
* 检查其中哪些提交不是合并操作的结果（C2，C3，C4）
* 检查哪些提交在对方覆盖更新时并没有被纳入目标分支（只有 C2 和 C3，因为 C4 其实就是 C4'）
* 把查到的这些提交应用在 teamone/master 上面

从而我们将得到与 [你将相同的内容又合并了一次，生成了一个新的提交](https://git-scm.com/book/zh/v2/ch00/_merge_rebase_work) 中不同的结果，如图 [在一个被变基然后强制推送的分支上再次执行变基](https://git-scm.com/book/zh/v2/ch00/_rebase_rebase_work) 所示。

要想上述方案有效，还需要对方在变基时确保 C4' 和 C4 是几乎一样的。 否则变基操作将无法识别，并新建另一个类似 C4 的补丁（而这个补丁很可能无法整洁的整合入历史，因为补丁中的修改已经存在于某个地方了）。

在本例中另一种简单的方法是使用 git pull --rebase 命令而不是直接 git pull。 又或者你可以自己手动完成这个过程，先 git fetch，再 git rebase teamone/master。、

如果你习惯使用 git pull ，同时又希望默认使用选项 --rebase，你可以执行这条语句 git config --global pull.rebase true 来更改 pull.rebase 的默认配置。

如果你只对不会离开你电脑的提交执行变基，那就不会有事。 如果你对已经推送过的提交执行变基，但别人没有基于它的提交，那么也不会有事。 如果你对已经推送至共用仓库的提交上执行变基命令，并因此丢失了一些别人的开发所基于的提交， 那你就有大麻烦了，你的同事也会因此鄙视你。

如果你或你的同事在某些情形下决意要这么做，请一定要通知每个人执行 git pull --rebase 命令，这样尽管不能避免伤痛，但能有所缓解。