代码合并：Merge还是Rebase

> BY 童仲毅(geeeeeeeeek@github)
>
> 这是一篇在原文(BY atlassian)基础上演绎的译文。除非另行注明，页面上所有内容采用知识共享-署名(CC BY 2.5 AU)协议共享。

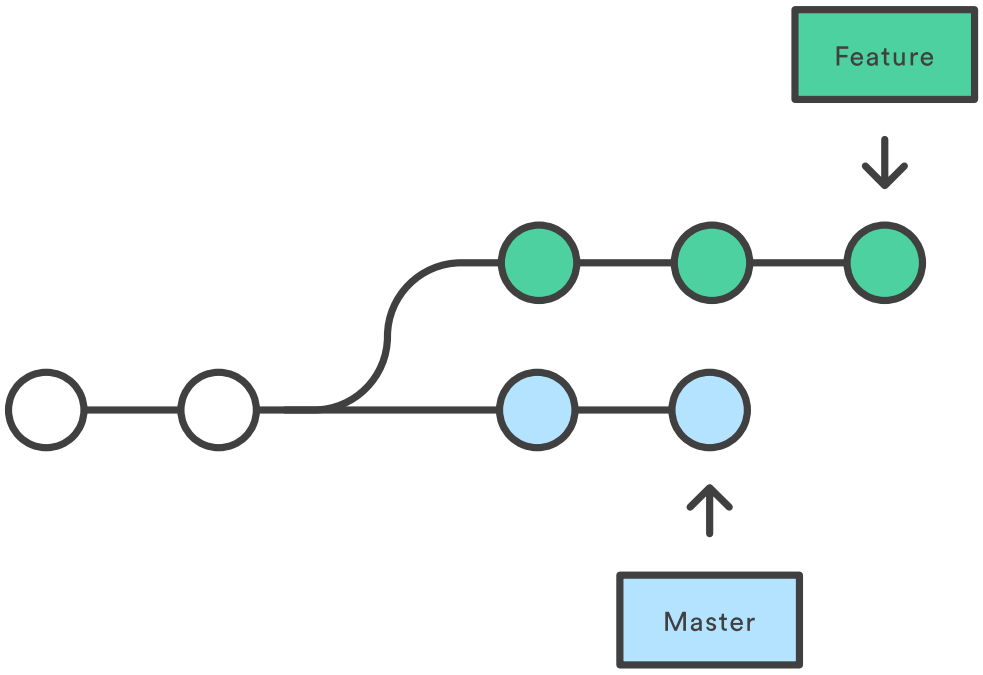
`git rebase` 这个命令经常被人认为是一种Git巫术，初学者应该避而远之。但如果使用得当的话，它能给你的团队开发省去太多烦恼。在这篇文章中，我们会比较`git rebase`和类似的`git merge`命令，找到Git工作流中rebase的所有用法。

概述

你要知道的第一件事是，`git rebase` 和`git merge` 做的事其实是一样的。它们都被设计来将一个分支的更改并入另一个分支，只不过方式有些不同。

想象一下，你刚创建了一个专门的分支开发新功能，然后团队中另一个成员在master分支上添加了新的提交。这就会造成提交历史被Fork一份，用Git来协作的开发者应该都很清楚。

A forked commit history



现在，如果master中新的提交和你的工作是相关的。为了将新的提交并入你的分支，你有两个选择：merge或rebase。

Merge

将master分支合并到feature分支最简单的办法就是用下面这些命令：

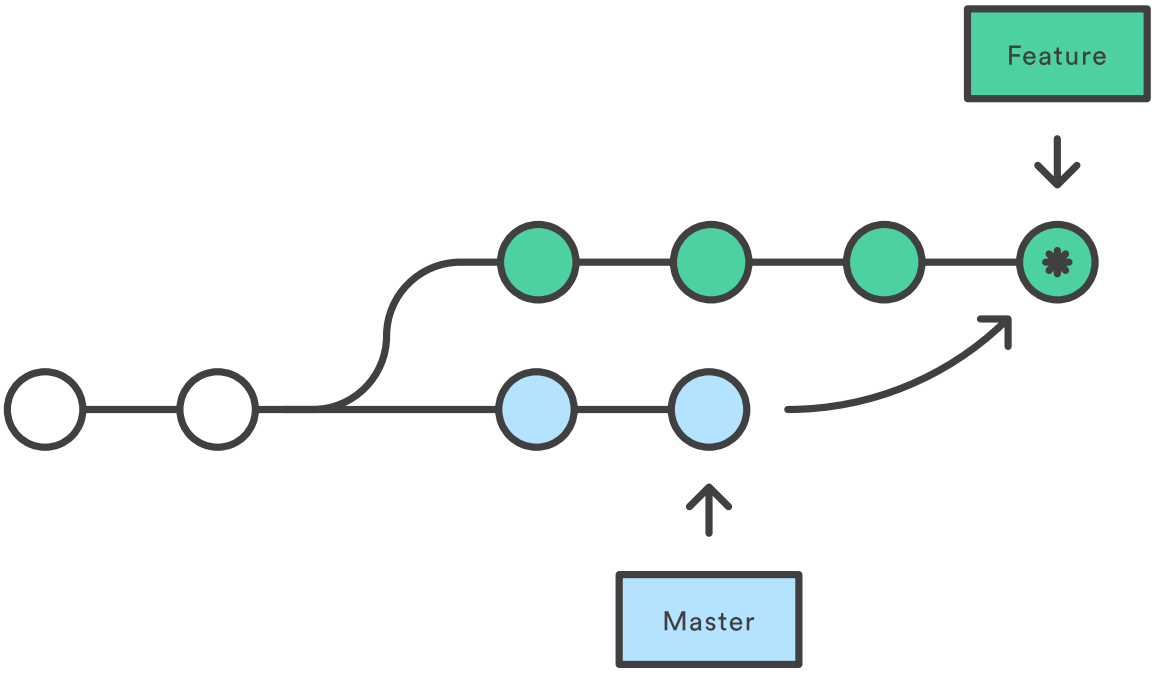
```
git checkout feature
git merge master
```

或者，你也可以把它们压缩在一行里。

```
git merge master feature
```

feature分支中新的合并提交(merge commit)将两个分支的历史连在了一起。你会得到下面这样的分支结构：

Merging master into the feature branch



Merge Commit

Merge好在它是一个安全的操作。现有的分支不会被更改，避免了rebase潜在的缺点（后面会说）。

另一方面，这同样意味着每次合并上游更改时feature分支都会引入一个外来的合并提交。如果master非常活跃的话，这或多或少会污染你的分支历史。虽然高级的`git log`选项可以减轻这个问题，但对于开发者来说，还是会增加理解项目历史的难度。

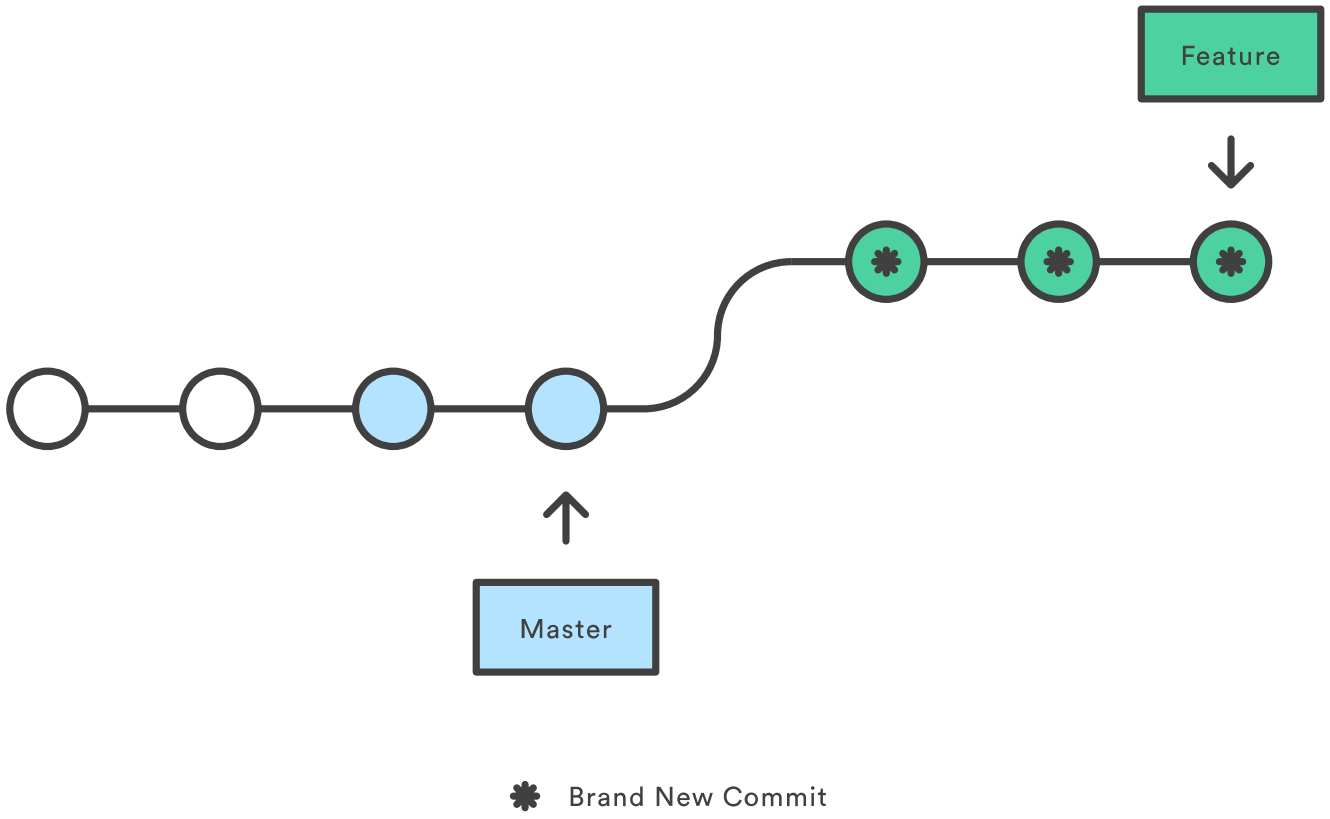
Rebase

作为merge的替代选择，你可以像下面这样将feature分支并入master分支：

```
git checkout feature
git rebase master
```

它会把整个feature分支移动到master分支的后面，有效地把所有master分支上新的提交并入过来。但是，rebase为原分支上每一个提交创建一个新的提交，重写了项目历史，并且不会带来合并提交。

Rebasing the feature branch onto master



rebase最大的好处是你的项目历史会非常整洁。首先，它不像`git merge`那样引入不必要的合并提交。其次，如上图所示，rebase导致最后的项目历史呈现出完美的线性——你可以从项目终点到起点浏览而不需要任何的Fork。这让你更容易使用`git log`、`git bisect`和`gitk`来查看项目历史。

不过，这种简单的提交历史会带来两个后果：安全性和可跟踪性。如果你违反了Rebase黄金法则，重写项目历史可能会给你的协作工作流带来灾难性的影响。此外，rebase不会有合并提交中附带的信息——你看不到feature分支中并入了上游的哪些更改。

交互式的rebase

交互式的rebase允许你更改并入新分支的提交。这比自动的rebase更加强大，因为它提供了对分支上提交历史完整的控制。一般来说，这被用于将feature分支并入master分支之前，清理混乱的历史。

把`-i`传入`git rebase`选项来开始一个交互式的rebase过程：

```
git checkout feature
git rebase -i master
```

它会打开一个文本编辑器，显示所有将被移动的提交：

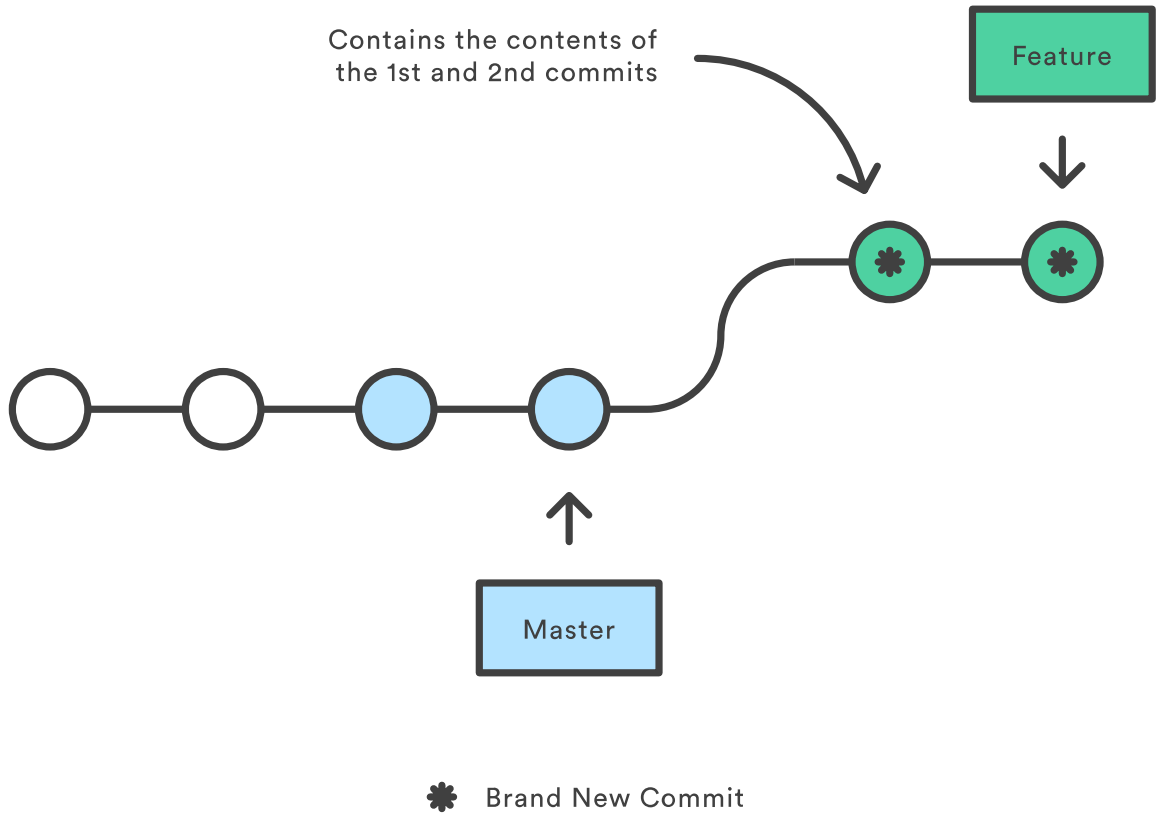
```
pick 33d5b7a Message for commit #1
pick 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

这个列表定义了rebase将被执行后分支会是什么样的。更改`pick`命令或者重新排序，这个分支的历史就能如你所愿了。比如说，如果第二个提交修复了第一个提交中的小问题，你可以用`fixup`命令把它们合到一个提交中：

```
pick 33d5b7a Message for commit #1
fixup 9480b3d Message for commit #2
pick 5c67e61 Message for commit #3
```

保存后关闭文件，Git会根据你的指令来执行rebase，项目历史看上去会是这样：

Squashing a commit with an interactive rebase



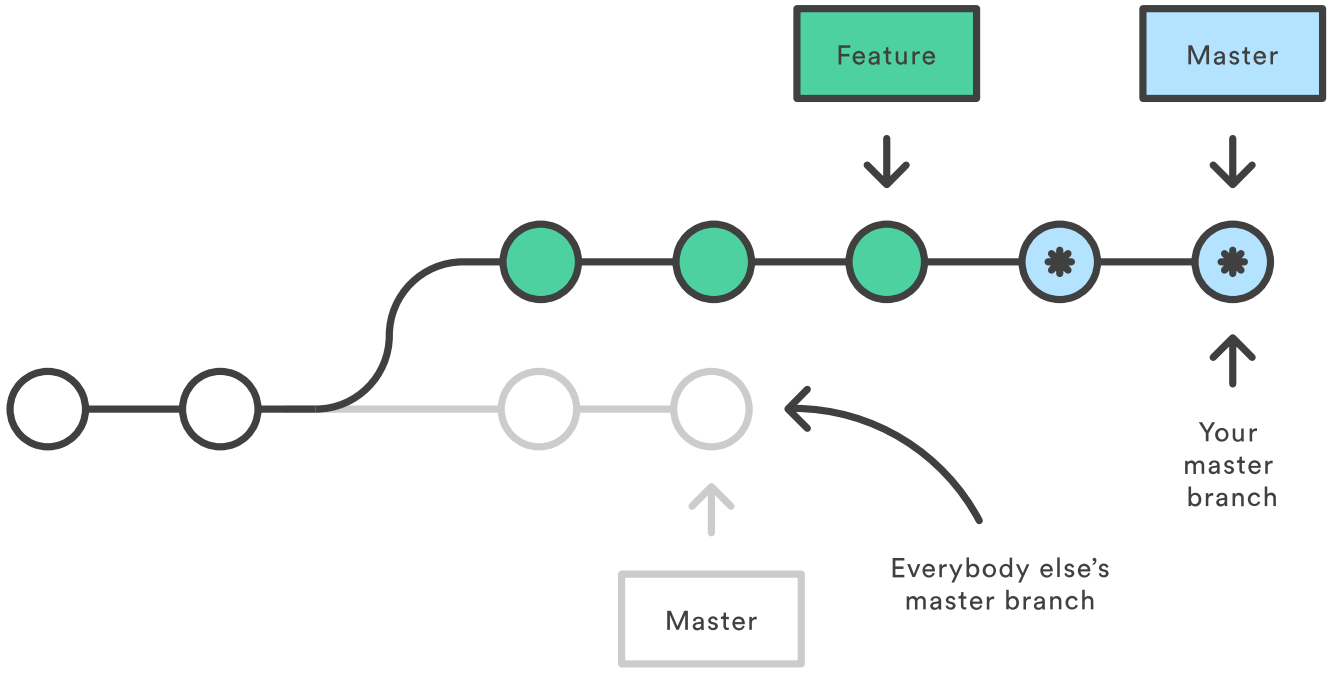
忽略不重要的提交会让你的feature分支的历史更清晰易读。这是`git merge`做不到的。

Rebase的黄金法则

当你理解rebase是什么的时候，最重要的就是什么时候 不能 用rebase。`git rebase`的黄金法则便是，绝不要在公共的分支上使用它。

比如说，如果你把master分支rebase到你的feature分支上会发生什么：

Rebasing the master branch



Brand New Commit

这次rebase将master分支上的所有提交都移到了feature分支后面。问题是它只发生在你的代码仓库中，其他所有的开发者还在原来的master上工作。因为rebase引起了新的提交，Git会认为你的master分支和其他人的master已经分叉了。

同步两个master分支的唯一办法是把它们merge到一起，导致一个额外的合并提交和两堆包含同样更改的提交。不用说，这会让人非常困惑。

所以，在你运行`git rebase`之前，一定要问问你自己“有没有别人正在这个分支上工作？”。如果答案是肯定的，那么把你的爪子放回去，重新找到一个无害的方式（如`git revert`）来提交你的更改。不然的话，你可以随心所欲地重写历史。

强制推送

如果你想把rebase之后的master分支推送到远程仓库，Git会阻止你这么做，因为两个分支包含冲突。但你可以传入`--force`标记来强行推送。就像下面一样：

```
# 小心使用这个命令！
git push --force
```

它会重写远程的master分支来匹配你仓库中rebase之后的master分支，对于团队中其他成员来

说这看上去很诡异。所以，务必小心这个命令，只有当你知道你在做什么的时候再使用。

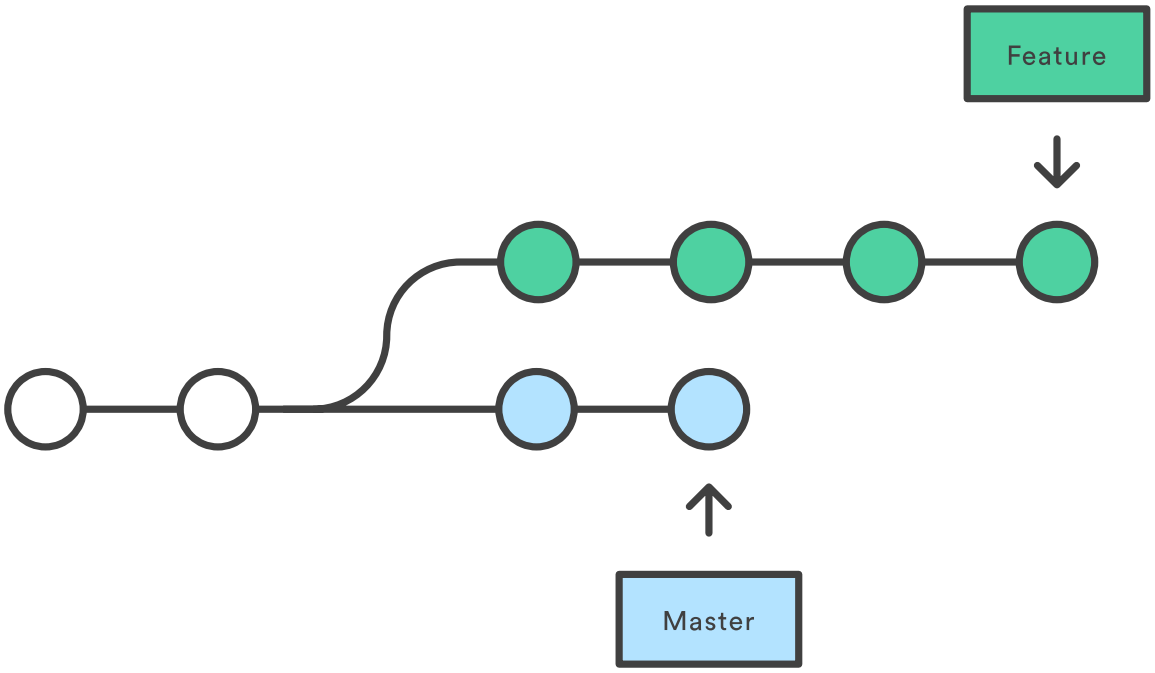
仅有的几个强制推送的使用场景之一是，当你在想向远程仓库推送了一个私有分支之后，执行了一个本地的清理（比如说为了回滚）。这就像是在说“哦，其实我并不想推送之前那个feature分支的。用我现在的版本替换掉吧。”同样，你要注意没有别人正在这个feature分支上工作。

工作流

rebase可以或多或少应用在你们团队的Git工作流中。在这一节中，我们来看看在feature分支开发的各个阶段中，rebase有哪些好处。

第一步是在任何和`git rebase`有关的工作流中为每一个feature专门创建一个分支。它会给你带来安全使用rebase的分支结构：

Developing a feature in a dedicated branch



本地清理

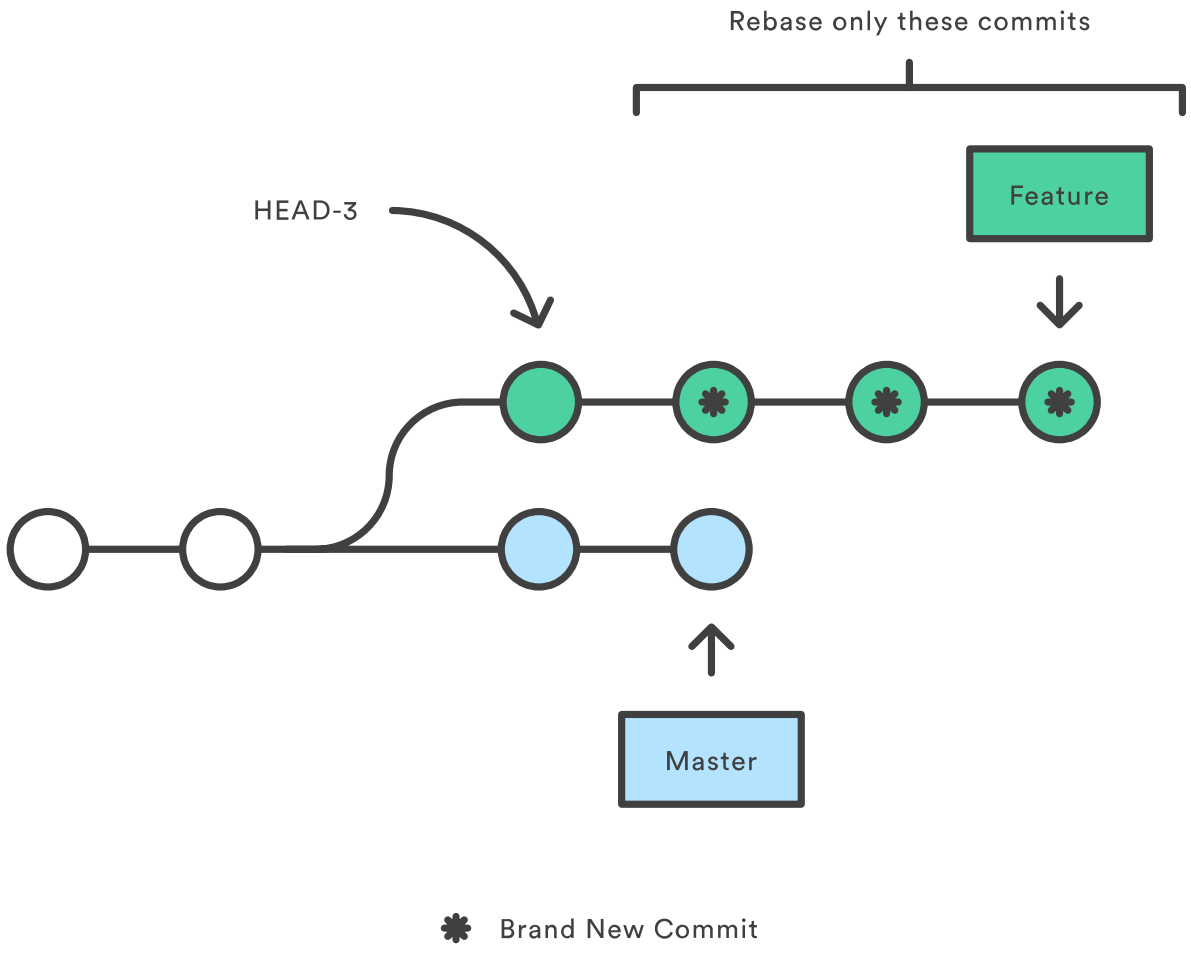
在你工作流中使用rebase最好的用法之一就是清理本地正在开发的分支。隔一段时间执行一次交互式rebase，你可以保证你feature分支中的每一个提交都是专注和有意义的。你在写代码时不用担心造成孤立的提交——因为你后面一定能修复。

调用`git rebase`的时候，你有两个基(base)可以选择：上游分支（比如master）或者你feature分支中早先的一个提交。我们在“交互式rebase”一节看到了第一种的例子。后一种在当你只需要修改最新几次提交时也很有用。比如说，下面的命令对最新的3次提交进行了交互式rebase：

```
git checkout feature
git rebase -i HEAD~3
```

通过指定`HEAD~3`作为新的基提交，你实际上没有移动分支——你只是将之后的3次提交重写了。注意它不会把上游分支的更改并入到feature分支中。

Rebasing onto HEAD-3



如果你想用这个方法重写整个feature分支，`git merge-base`命令非常方便地找出feature分支开始分叉的基。下面这段命令返回基提交的ID，你可以接下来将它传给`git rebase`：

```
git merge-base feature master
```

交互式rebase是在你工作流中引入`git rebase`的的好办法，因为它只影响本地分支。其他开发者只能看到你已经完成的结果，那就是一个非常整洁、易于追踪的分支历史。

但同样的，这只能用在私有分支上。如果你在同一个feature分支和其他开发者合作的话，这个分支是公开的，你不能重写这个历史。

用带有交互式的rebase清理本地提交，这是无法用`git merge`命令代替的。

将上游分支上的更改并入feature分支

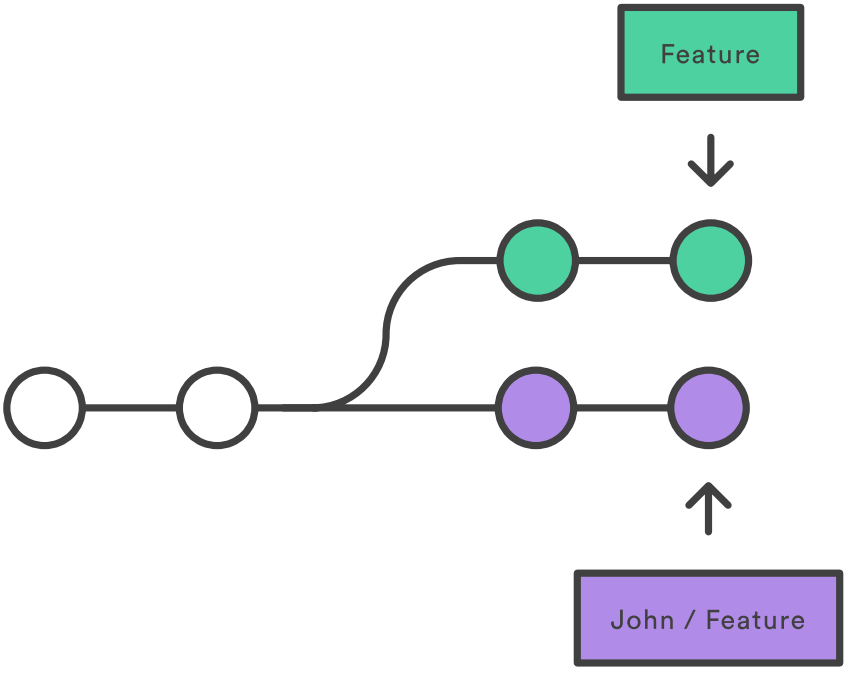
在概览一节，我们看到了feature分支如何通过`git merge`或`git rebase`来并入上游分支。merge是保留你完整历史的安全选择，rebase将你的feature分支移动到master分支后面，创建一个线性的历史。

`git rebase`的用法和本地清理非常类似（而且可以同时使用），但之间并入了master上的上游更改。

记住，rebase到远程分支而不是master也是完全合法的。当你和另一个开发者在同一个feature分之上协作的时候，你会用到这个用法，将他们的更改并入你的项目。

比如说，如果你和另一个开发者——John——往feature分支上添加了几个提交，在从John的仓库中fetch之后，你的仓库可能会像下面这样：

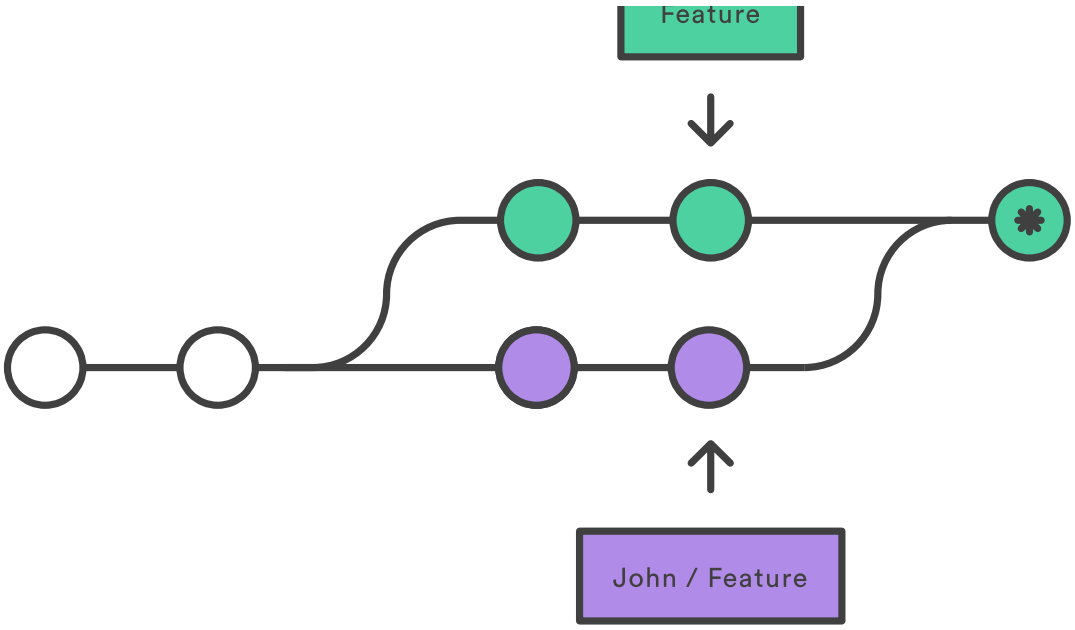
Collaborating on the same feature branch



就和并入master上的上游更改一样，你可以这样解决这个Fork：要么merge你的本地分支和John的分支，要不把你的本地分支rebase到John的分支后面。

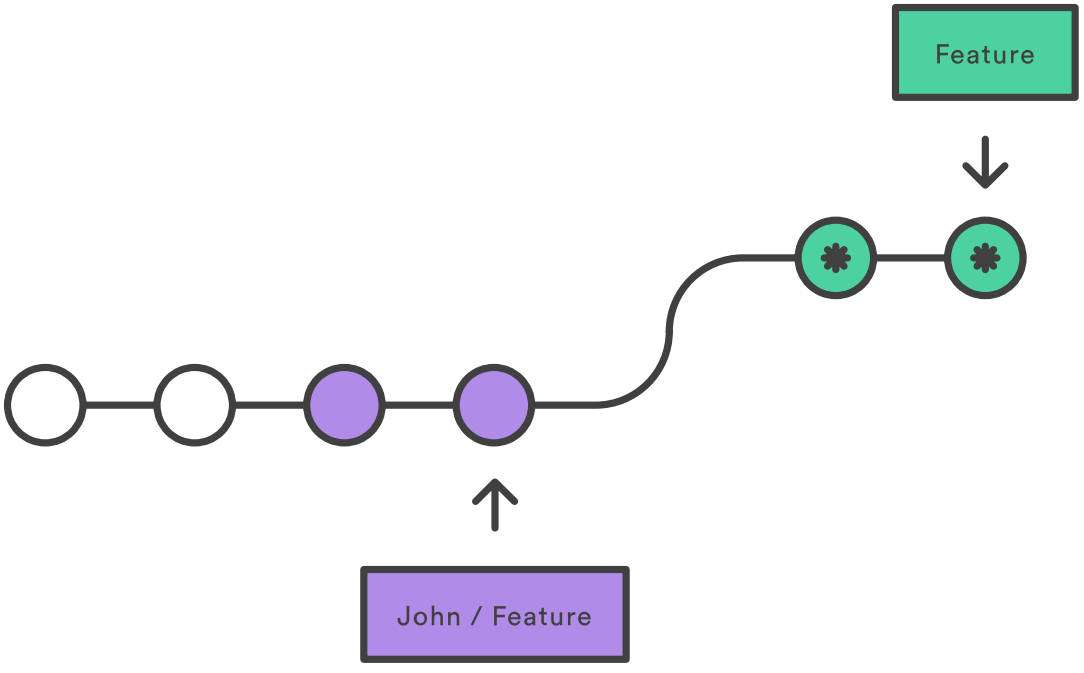
Merging vs. rebasing onto a remote branch

Merging



Merge Commit

Rebasing



Brand New Commits

注意，这里的rebase没有违反Rebase黄金法则，因为只有你的本地分支上的commit被移动了，之前的所有东西都没有变。这就像是在说“把我的改动加到John的后面去”。在大多数情况下，这比通过合并提交来同步远程分支更符合直觉。

默认情况下，`git pull` 命令会执行一次merge，但你可以传入 `--rebase` 来强制它通过rebase来整合远程分支。

用Pull Request进行审查

如果你将pull request作为你代码审查过程中的一环，你需要避免在创建pull request之后使用 `git rebase` 。只要你发起了pull request，其他开发者能看到你的代码，也就是说这个分支变成了公共分支。重写历史会造成Git和你的同事难以找到这个分支接下来的任何提交。

来自其他开发者的任何更改都应该用 `git merge` 而不是 `git rebase` 来并入。

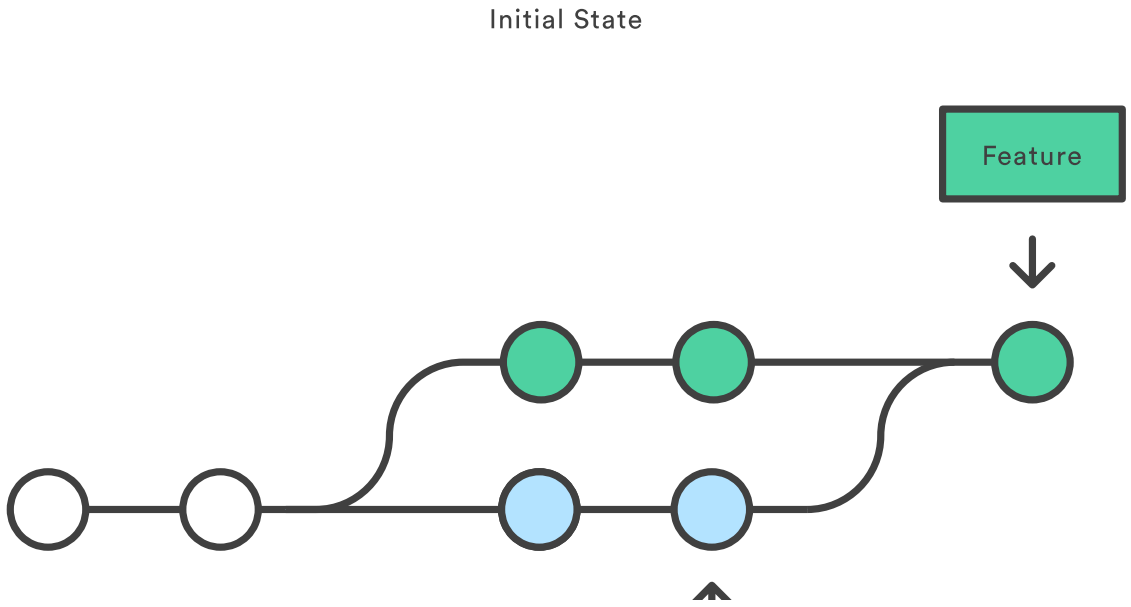
因此，在提交pull request前用交互式的rebase进行代码清理通常是一个好的做法。

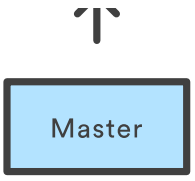
并入通过的功能分支

如果某个功能被你们团队通过了，你可以选择将这个分支rebase到master分支之后，或是使用 `git merge` 来将这个功能并入主代码库中。

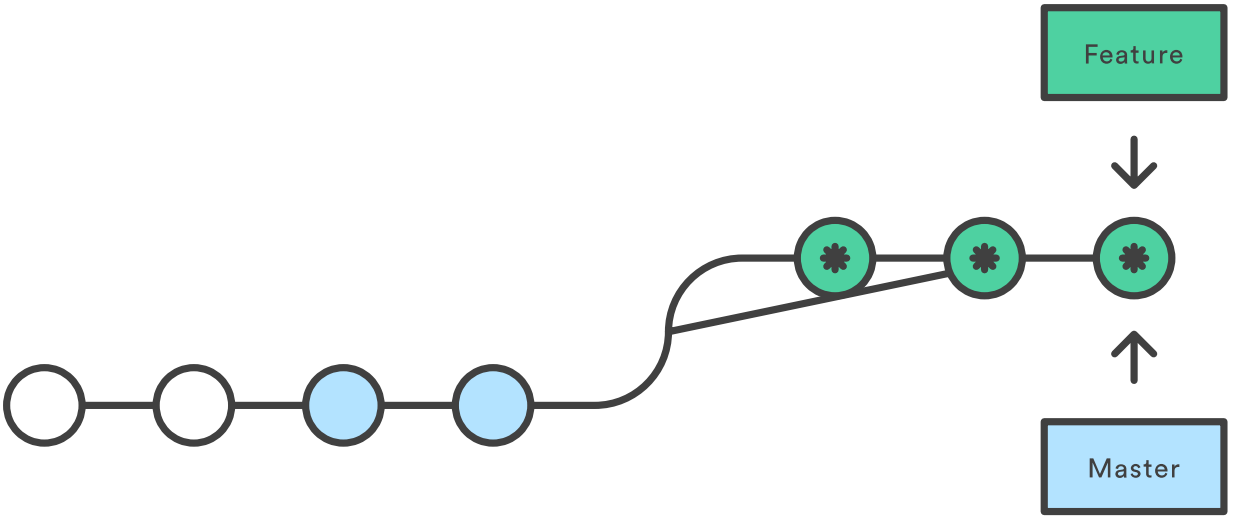
这和将上游改动并入feature分支很相似，但是你不可以在master分支重写提交，你最后需要用 `git merge` 来并入这个feature。但是，在merge之前执行一次rebase，你可以确保merge是一直向前的，最后生成的是一个完全线性的提交历史。这样你还可以加入pull request之后的提交。

Integrating a feature into master with and without a rebase



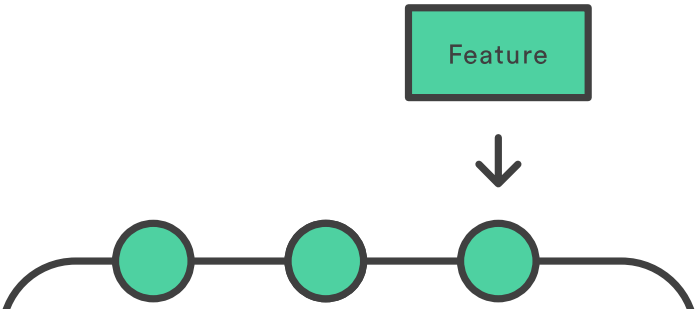


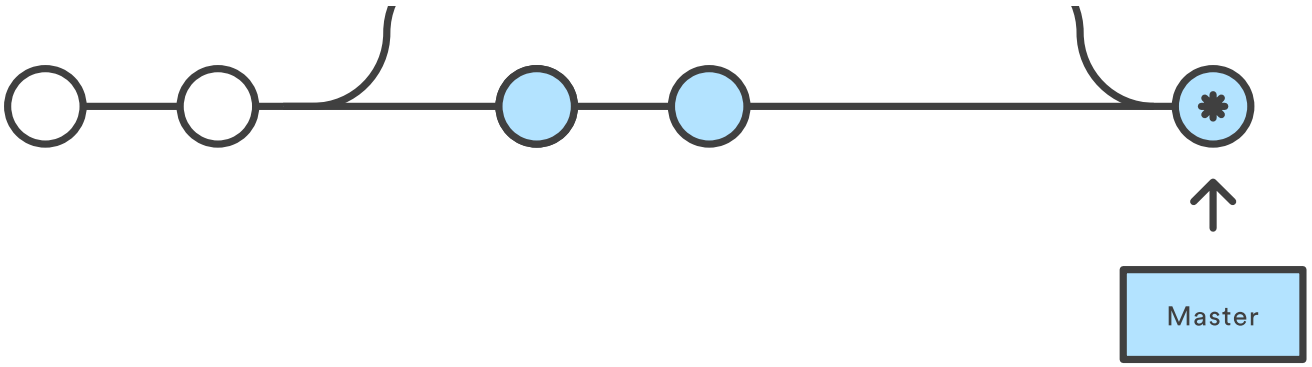
Rebase and Merge



✱ Brand New Commits

Merge without rebasing





Merge Commit

如果你还没有完全熟悉`git rebase`，你还可以在一个临时分支中执行rebase。这样的话，如果你意外地弄乱了你feature分支的历史，你还可以查看原来的分支然后重试。

比如说：

```
git checkout feature
git checkout -b temporary-branch
git rebase -i master
# [清理目录]
git checkout master
git merge temporary-branch
```

总结

你使用rebase之前需要知道的知识点都在这了。如果你想要一个干净的、线性的提交历史，没有不必要的合并提交，你应该使用`git rebase`而不是`git merge`来并入其他分支上的更改。

另一方面，如果你想要保存项目完整的历史，并且避免重写公共分支上的commit， 你可以使用`git merge`。两种选项都很好用，但至少你现在多了`git rebase`这个选择。