

Git 菜单



高质量的 Git 中文教程，源于国外社区的优秀文章和个人实践.



下载手机APP
畅享精彩阅读

目 录

致谢

Git 菜单

1. 果壳中的Git

2.1 Git简易指南

2.2 创建代码仓库

2.3 保存你的更改

2.4 查看仓库状态

2.5 检出以前的提交

2.6 回滚错误的修改

2.7 重写项目历史

3.2 保持代码同步

3.3 创建PullRequest

3.4 使用分支

3.5 常见工作流比较

4. Git 图解

5.1 代码合并Merge还是Rebase

5.2 回滚命令Reset、Checkout、Revert辨析

5.3 Git_log高级用法

5.4 Git钩子

5.5 Git提交引用

致谢

当前文档 《Git 菜单》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2006-01-02。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：童仲毅 <https://github.com/geeeeeeeek/git-recipes>

文档地址：<http://www.bookstack.cn/books/git-recipes>

书栈官网：<https://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Git 菜单

高质量的 *Git* 中文教程，源于国外社区的优秀文章和个人实践

第1篇 果壳中的 Git

- 第1章 [什么是 Git](#)

第2篇 从零搭建本地代码仓库

本篇完全面向入门者。我假设你从零开始创建一个项目并且想用 Git 来进行版本控制，我们会讨论如何在你的个人项目中使用 Git，比如如何初始化你的项目，如何管理新的或者已有的文件，如何在远端仓库中储存你的代码。

- 第1章 [快速指南](#)
- 第2章 [创建代码仓库](#)
- 第3章 [保存你的更改](#)
- 第4章 [检查仓库状态](#)
- 第5章 [检出之前的提交](#)
- 第6章 [回滚错误的修改](#)
- 第7章 [重写项目历史](#)

第3篇 远程团队协作和管理

- 第1章 [快速指南](#)
- 第2章 [保持同步](#)
- 第3章 [创建 Pull Request](#)
- 第4章 [使用分支](#)
- 第5章 [常见工作流比较](#)

第4篇 Git 命令详解

- 第1章 [图解 Git 命令](#)

如果你稍微理解 Git 的工作原理，这篇文章能够让你理解的更透彻。

第5篇 Git 实用贴士

- 第1章 [代码合并：Merge、Rebase 的选择](#)

`git rebase` 和 `git merge` 都是用来合并分支，只不过方式不太相同。`git rebase` 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，它能省去太多烦恼。在这篇文章中，我们会通过比较找到 Git 工作流中所有可以使用 rebase 的机会。

- 第2章 [代码回滚：Reset、Checkout、Revert 的选择](#)

`git reset`、`git checkout` 和 `git revert` 都是用来撤销代码仓库中的某些更改，所以我们经常弄混。在这篇文章中，我们比较最常见的用法，分析在什么场景下该用哪个命令。

- 第3章 [Git log 高级用法](#)

任何一个版本控制系统设计的目的都是为了记录你代码的变化——谁贡献了什么，找出 bug 是什么时候引入的，以及撤回一些有问题的更改。`git log` 可以格式化 commit 输出的形式，或过滤输出的 commit 从而找到项目中你需要的任何信息。


- 第4章 [Git 钩子：自定义你的工作流](#)

Git 钩子是在 Git 仓库中特定事件发生时自动运行的脚本。它可以让你自定义 Git 内部的行为，在开始周期中的关键点触发自定义的行为，自动化或者优化你开发工作流中任意部分。

- 第5章 [Git 提交引用和引用日志](#)

提交是 Git 的精髓所在，你无时不刻不在创建和缓存提交、查看以前的提交，或者用各种 Git 命令在仓库间转移你的提交。在这章中，我们研究提交的各种引用方式，以及涉及到的 Git 命令的工作原理。我们还会学到如何使用 Git 的引用日志查看看似已经删除的提交。

版权说明

-  [童仲毅 \(geeeeeeeeek@github\)](#)
- 除非另行注明，这个项目中的所有内容采用知识共享-署名 (CC BY 2.5 AU) 协议共享。
- 不少文章在原基础上翻译或演绎而来，页面上方标注了原作者、原文链接以及原文采用的协议。如有版权疑问，请在 Issue 中提出。
- 欢迎通过 Issue 或者 Pull Request 推荐你认为合适的资料，让这份菜单更充实一些。

为什么要做这份菜单

在整理 Git 资料的时候，我发现社区贡献了非常多高质量的博客文章、指南等等。尤其英文的那些资料，除了大家熟知的「Git 图解」，还有好多优秀的文章仍无人翻译。此外，这些资料往往只涉及某些特定的话题，如果能有一份菜单将这些菜谱以特定的方式串起来，那么对于 Git 学习者来说将会是极大的便利。尤其对于我这样热爱查阅社区资料胜过出版物的懒人：] 随着我的学习节奏还会不断有新的菜谱加入进来，或许不会很频繁，不过也没有确定的终点。

写于 2015 年

什么是 Git

houkensisjt | 董仲毅

© 本文演绎自 Atlassian 编写的 *What is Git*。页面上所有内容采用知识共享-署名 (CC BY 2.5 AU) 许可协议。

到目前为止，Git 是世界上使用最为广泛的现代化版本控制系统。Git 最初由 Linux 系统内核的作者 Linus Torvalds 在 2005 年开始开发，目前已经是一个持续维护的成熟开源项目。如今，大量软件项目依赖 Git 进行版本管理，其中既有开源软件，也有商业软件。Git 在很多操作系统和集成开发环境 (IDE) 上都表现良好。绝大多数软件开发者或多或少都使用过 Git。

Git 是分布式版本管理 (DVCS) 的一种。CVS 和 Subversion (SVN) 等集中式的版本管理软件将完整的版本历史存放在同一个地方。而在 Git 中，每个开发者的代码仓库都包含了所有变更历史。

除了分布式之外，Git 在设计之初也考虑了性能、安全性和灵活性。

高性能

Git 的底层性能相较于其他版本管理软件有强大的优势。提交修改、创建分支、合并分支和比较版本都针对性能进行了优化。Git 中实现的算法利用了现实中代码树的特点以及它们被修改和访问的常见模式。

不同于某些版本管理软件，Git 在决定文件树的储存和版本历史时，不会被文件名的变化所愚弄——Git 关注的是文件的内容本身。毕竟，代码文件经常会被重命名、拆分和重新编排。Git 仓库中的文件对象通过差分编码 (delta encoding，仅保存代码修改的差分) 和压缩技术储存，并且直接保存文件夹中的内容和版本控制元数据。

分布式架构也给 Git 带来了巨大的性能优势。

比如说，有一名开发成员 Alice 修改了代码，添加了一些准备在 2.0 版本中发布的功能，然后提交了这些修改及其描述。随后，她又编写并提交了另一个新功能。很自然地，这两次修改是版本历史中两份独立的工作。Alice 又切换到了 1.3 版本的分支，修复了一个只影响这个旧版本的 bug。这次修复的目的是为了让团队在 2.0 版本还没有完成之前，发布一个 1.3.1 版本来解决旧版本中的一些 bug。Alice 可以立刻回到 2.0 版本分支，继续新功能开发。这一切都不需要网络连接，非常快速可靠，甚至可以在飞机中完成。当她准备好将这些单独提交的更改发送到远程仓库时，她只需要一个“推送” (push) 命令。

安全

Git 设计时就托托管代码的完好性作为重中之重。文件内容以及文件、目录、版本、标签和提交的关联，都通过安全的加密哈希校验算法 (SHA1) 保护。这可以避免代码和修改历史被不小心或者恶意改变，并且保证修改历史完全可追溯。

你可以相信在 Git 中源代码的修改历史是真实可靠的。

有一些版本管理软件无法防止版本历史之后被篡改。这对于任何依赖软件开发的团队来说都是严重的安全漏洞。

灵活

Git 的关键设计目标之一就是灵活。Git 在很多方面都展现出了其灵活性：支持多种非线性的工作流，对不同规模的

项目来说都很高效，并且兼容多个操作系统和协议。

Git 在设计时最重要的功能便是分支和标签（不同于 SVN），因此所有影响分支和标签的操作也都会被保存到修改历史中。不是所有的版本管理软件关注的都是这个层面的版本追踪。

使用 Git 进行版本管理

Git 对于绝大多数软件开发团队来说都是最好的选择。虽然每个团队都需要考虑自身的情况，但我们依然可以列举一些 Git 比其他版本控制系统更好的理由：

Git 很好用

Git 兼具大多数团队和个人开发者需要的功能、性能，安全性和灵活性。我们已经具体讨论过了这些特点。对很多团队来说，它们发现 Git 在这几点上都表现的更优秀。

Git 已经成为了事实上的行业标准

Git 使用最广泛的版本管理软件。这使得 Git 在以下这些方面具有极大的吸引力。在 Atlassian（作者所在的公司），大多数代码都是通过 Git 管理的。

大量开发者都有过 Git 的使用经历，很大一部分大学毕业生甚至只用过 Git 进行版本管理。虽然迁移到 Git 的过程中或许会经历比较陡峭的学习曲线，但是大多数员工以及未来的员工都已经具备了使用 Git 的基本技能，这意味着他们不需要额外的培训。

除了拥有大量使用者之外，Git 的普及还意味着很多第三方的服务和 IDE 都已经集成了 Git。比如我们的 DVCS 桌面客户端 [Source Tree](#)、项目开发管理软件 [JIRA](#) 和代码托管服务 [Bitbucket](#)。

如果你是一个想要积累软件开发工具使用技能的新人，Git 毫无疑问是你在版本管理方面的第一选择。

Git 是一个高质量的开源项目

Git 本身是一个经历多年良好支持和管理的开源软件项目。Git 的维护者很好地平衡了长远的用户需求，和改进可用性和功能性的例行更新。这个开源项目的质量久经考验，无数企业都极度依赖于此。

Git 还拥有良好的社区支持和庞大的用户群体。你可以找到各种深入浅出的学习资料，包括书籍，教程，以及专题网站。你也可以找到相关的播客节目和视频教程。

开源降低了业余开发者的成本，因为他们不需要花一分钱来使用 Git。对于开源项目来说，Git 无疑是 SVN 和 CVS 等上一代流行版本管理软件的接班人。

对 Git 的批评

对于 Git 的一个常见批评是它学起来不那么容易。Git 中的某些术语对于新手或者是使用其他系统的朋友可能会比较陌生。比如说，`revert` 这个命令在 Git、SVN、CVS 中具有不同的含义。不过，Git 向用户提供了非常强大的功能。学习掌握这些功能也许会花一些时间，但是一旦你学会了这些技能，它们会帮助你大大提高团队的开发效率。

对于曾经使用非分布式版本管理的团队来说，他们可能不想放弃中央服务器。不过，虽然 Git 被设计成分布式的架构，你依然可以建立一个“官方”的代码库来存放所有的修改。使用 Git 时，由于所有的开发者都拥有完整的代码库

拷贝，所以他们的工作不会被中央服务器的状态和性能所影响。即使遇到故障，他们依然可以查看完整的项目历史。得益于 Git 的灵活性和分布式特点，你可以在保持原有工作方式的同时还可以得到 Git 带来的额外好处，而你以前甚至不会意识到这些好处。

现在你已经明白了什么是版本管理，什么是 Git 以及为什么要使用 Git ，你可以选择继续阅读下一节，了解 Git 在整个组织层面带来的好处。

这篇文章是「[Git Recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [协作说明](#)。

Git 简易指南(上)

BY 董仲毅 (geeeeeeeeek@github)

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-guide](#) 项目对本文亦有贡献。

这节是完全面向入门者的，我假设你从零开始创建一个项目并且想用 Git 来进行版本控制，因此本文会避开分支这些相对复杂的概念。

在这节中，我会介绍如何在你的个人项目中使用 Git，我们会讨论 Git 最基本的操作——如何初始化你的项目，如何管理新的或者已有的文件，如何在远端仓库中储存你的代码。

安装 Git

- Mac 用户: Xcode Command Line Tools 自带 Git (`xcode-select --install`)
- Linux 用户: `sudo apt-get install git`
- Windows 用户: 下载 [Git SCM](#)

- 对于 Windows 用户，安装后如果希望在全局的 cmd 中使用 Git，需要把 git.exe 加入 PATH 环境变量中，或在 Git Bash 中使用 Git。

检出仓库

执行如下命令以创建一个本地仓库的克隆版本：

```
git clone /path/to/repository
```

如果是远端服务器上的仓库，你的命令会是这个样子：

```
git clone username@host:/path/to/repository
```

 （通过 SSH）

或者：

```
git clone https://path/to/repository.git
```

 （通过 https）

比如说 `git clone https://github.com/geeeeeeeeek/git-recipes.git` 可以将 git 教程 clone 到你指定的目录。

创建新仓库

创建新文件夹，打开，然后执行 `git init` 以创建新的 git 仓库。

下面每一步中，你都可以通过 `git status` 来查看你的git仓库状态。

工作流

你的本地仓库由 Git 维护的三棵「树」组成。第一个是你的 **工作目录**，它持有实际文件；第二个是 **缓存区 (Index)**，它像个缓存区域，临时保存你的改动；最后是 **HEAD**，指向你最近一次提交后的结果。



事实上，第三个阶段是 commit history 的图。HEAD 一般是指向最新一次 commit 的引用。现在暂时不必究其细节。

添加与提交

你可以计划改动（把它们添加到缓存区），使用如下命令：

```
1. git add < filename >
2. git add *
```

这是 Git 基本工作流程的第一步。使用如下命令以实际提交改动：

```
1. git commit -m "代码提交信息"
```

现在，你的改动已经提交到了 HEAD，但是还没到你的远端仓库。

在开发时，良好的习惯是根据工作进度及时 commit，并务必注意附上有意义的 commit message。创建完项目目录后，第一次提交的 commit message 一般为「Initial commit」。

推送改动

你的改动现在已经在本地仓库的 HEAD 中了。执行如下命令以将这些改动提交到远端仓库：

```
1. git push origin master
```

可以把 master 换成你想要推送的任何分支。

如果你还没有克隆现有仓库，并欲将你的仓库连接到某个远程服务器，你可以使用如下命令添加：

```
1. git remote add origin <server>
```

如此你就能够将你的改动推送到所添加的服务器上去了。

- 这里 origin 是 <server> 的别名，取什么名字都可以，你也可以在 push 时将 <jserver> 替换为 origin。但为了以后 push 方便，我们第一次一般都会先 remote add。
- 如果你还没有 Git 仓库，可以在 Github 等代码托管平台上创建一个空（不要自动生成 README.md）的仓库，然后将代码 push 到远端仓库。

至此，你应该可以顺利地提交你的项目了。在下一节中，我们将涉及更多的命令，来完成更有用的操作。比如从远端的仓库拉取更新并且合并到你的本地，如何通过分支多人协作，如何处理不同分支的冲突等等。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 **Star** :star2: 或 **Fork** :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

创建代码仓库

BY 童仲毅 (geeeeeeeeeek@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。

这一章简要地带你了解一些最重要的 Git 命令。在这节中，我会向你介绍开始一个新的版本控制项目需要的所有工具，后面的几节包含了你每天都会用到的Git操作。

在这节之后，你应该能够创建一个新的 Git 仓库，缓存你的项目以免丢失，以及查看你项目的历史。

git init

`git init` 命令创建一个新的 Git 仓库。它用来将已存在但还没有版本控制的项目转换成一个 Git 仓库，或者创建一个空的新仓库。大多数Git命令在未初始化的仓库中都是无法使用的，所以这就是你运行新项目的第一个命令了。

运行 `git init` 命令会在你项目的根目录下创建一个新的 `.git` 目录，其中包含了你项目必需的所有元数据。除了 `.git` 目录之外，已经存在的项目不会被改变（就像 SVN 一样，Git 不强制每个子目录中都有一个 `.git` 目录）。

用法

```
1. git init
```

将当前的目录转换成一个 Git 仓库。它在当前的目录下增加了一个 `.git` 目录，于是就可以开始记录项目版本了。

```
1. git init <directory>
```

在指定目录创建一个空的 Git 仓库。运行这个命令会创建一个名为 `directory`，只包含 `.git` 子目录的空目录。

```
1. git init --bare <directory>
```

初始化一个裸的 Git 仓库，但是忽略工作目录。共享的仓库应该总是用 `--bare` 标记创建（见下面的讨论）。一般来说，用 `--bare` 标记初始化的仓库以 `.git` 结尾。比如，一个叫 `my-project` 的仓库，它的空版本应该保存在 `my-project.git` 目录下。

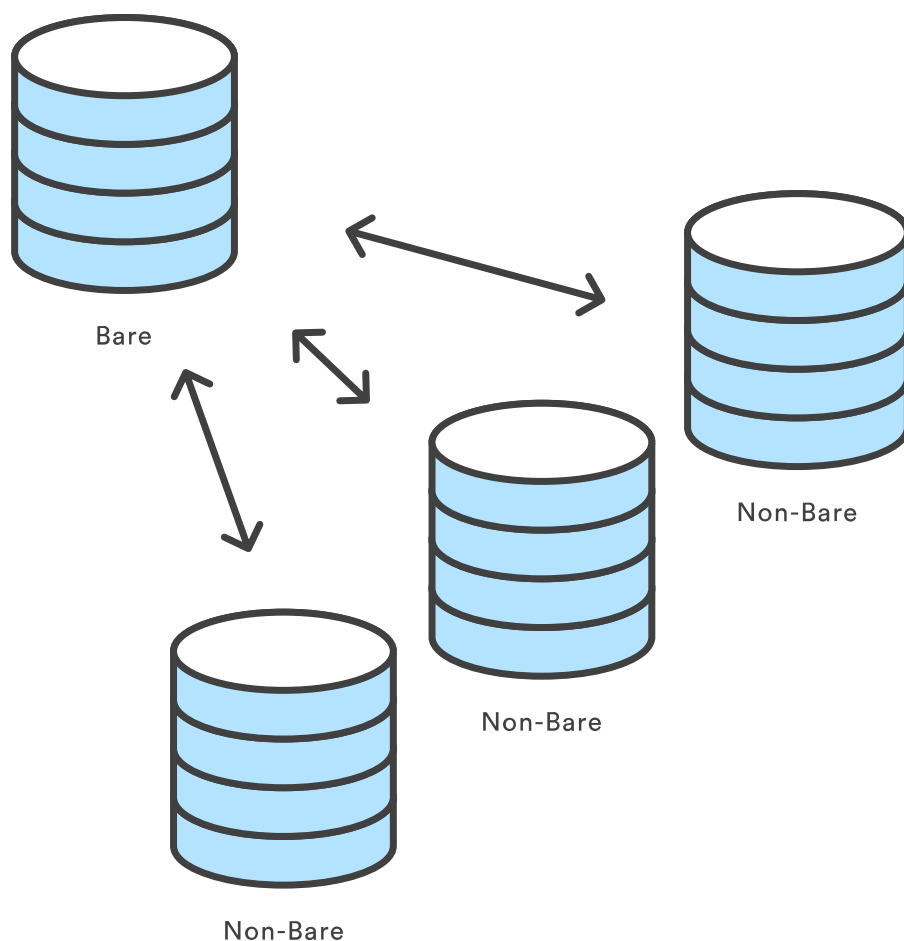
讨论

和 SVN 相比，`git init` 命令是一个创建新的版本控制项目非常简单的途径。Git 不需要你创建仓库，导入文件，检查正在修改的拷贝。你只需要 `cd` 到你的项目目录下，运行 `git init`，你就有了一个功能强大的 Git 仓库。

但是，对大多数项目来说，`git init` 只需要在创建中央仓库时执行一次——开发者通常不会使用 `git init` 来创建他们的本地仓库。他们往往使用 `git clone` 来将已存在的仓库拷贝到他们的机器中去。

裸仓库

`--bare` 标记创建了一个没有工作目录的仓库，这样我们在仓库中更改文件并且提交了。中央仓库应该总是创建成裸仓库，因为向非裸仓库推送分支有可能会覆盖已有的代码变动。将 `--bare` 看成是用来将仓库标记为储存设施，而不是一个开发环境。也就是说，对于所有的 Git 工作流，中央仓库是裸仓库，开发者的本地仓库是非裸仓库。



栗子

因为 `git clone` 创建项目的本地拷贝更为方便，`git init` 最常见的使用情景就是用于创建中央仓库：

```
1. ssh <user>@<host>
2.
3. cd path/above/repo
4.
5. git init --bare my-project.git
```

首先，你用SSH连入存放中央仓库的服务器。然后，来到任何你想存放项目的地方，最后，使用 `--bare` 标记来创建一个中央存储仓库。开发者会将 `my-project.git` 克隆到本地的开发环境中。

git clone

`git clone` 命令拷贝整个 Git 仓库。这个命令就像 `svn checkout` 一样，除了「工作副本」是一个完备的 Git 仓库——它包含自己的历史，管理自己的文件，以及环境和原仓库完全隔离。

为了方便起见，`clone` 自动创建了一个名为 `origin` 的远程连接，指向原有仓库。这让和中央仓库之间的交互更加简单。

用法

```
1. git clone <repo>
```

将位于 `<repo>` 的仓库克隆到本地机器。原仓库可以在本地文件系统中，或是通过 HTTP 或 SSH 连接的远程机器。

```
1. git clone <repo> <directory>
```

将位于 `<repo>` 的仓库克隆到本地机器上的 `<directory>` 目录。

讨论

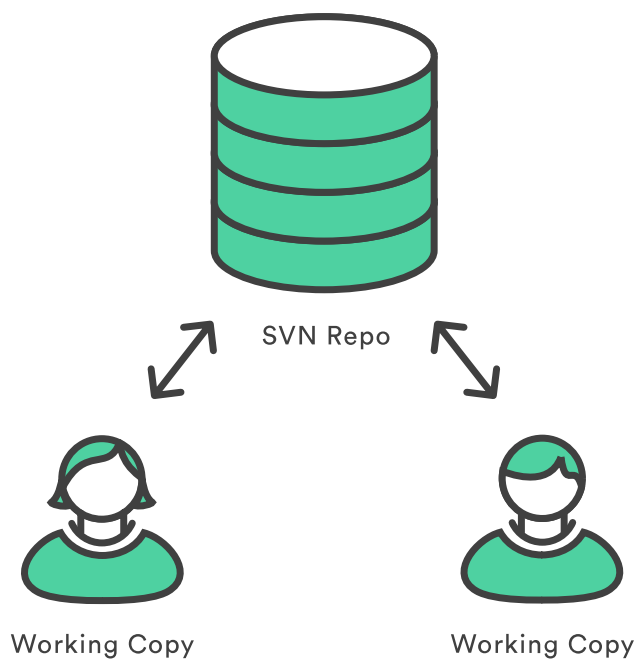
如果项目在远程仓库已经设置完毕，`git clone` 是用户获取开发副本最常见的方式。和 `git init` 相似，`clone` 通常也是一次性的操作——只要开发者获得了一份工作副本，所有版本控制操作和协作管理都是在本地仓库中完成的。

仓库间协作

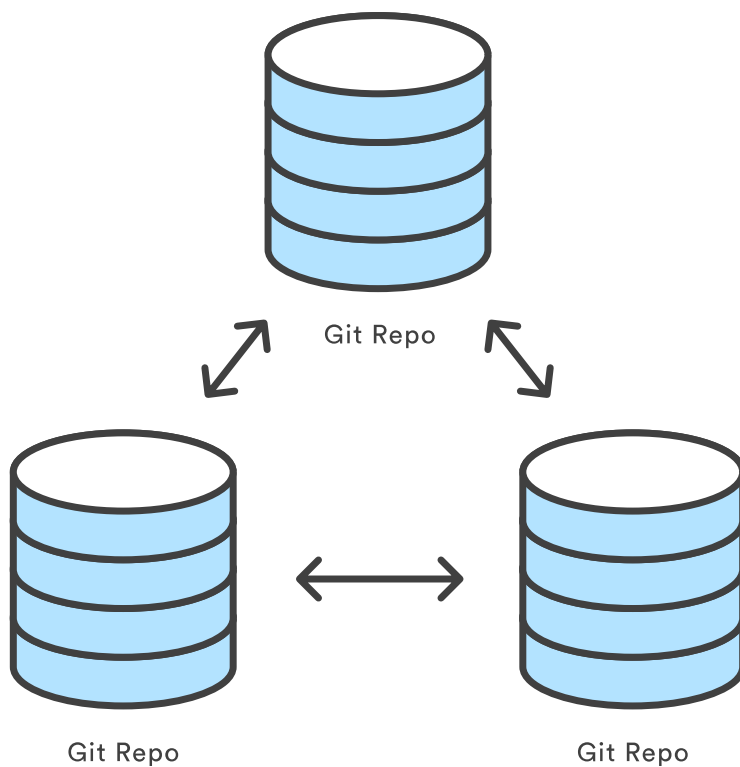
这一点很重要，你要理解 Git 中「工作副本」的概念和 SVN 仓库 check out 下来的「工作副本」是很不一样的。和 SVN 不同的是，Git 不会区分工作副本和中央仓库——它们都是功能完备的 Git 仓库。

这就使得 Git 的协作和 SVN 截然不同。SVN 依赖于中央仓库和工作副本之间的关系，而 Git 协作模型是基于仓库和仓库之间的交互的。相对于 SVN 的提交流程，你可以在 Git 仓库之间 `push` 或 `pull` 提交。

Central-Repo-to-Working-Copy Collaboration



Repo-To-Repo Collaboration



当然，你也完全可以给予某个特定的仓库一些特殊的含义。比如，指定某个 Git 仓库为中央仓库，你就可以用 Git 进行中央化的工作流。重点是，这是通过约定实现的，而不是写死在版本控制系统本身。

栗子

下面这个例子演示用 SSH 用户名 john 连接到 example.com，获取远程服务器上中央仓库的本地副本：

```
1. git clone ssh://john@example.com/path/to/my-project.git
2.
3. cd my-project
4.
5. # 开始工作
```

第一行命令在本地机器的 `my-project` 目录下初始化了一个新的 Git 仓库，并且导入了中央仓库中的文件。接下来，你 `cd` 到项目目录，开始编辑文件、缓存提交、和其它仓库交互。同时注意 `.git` 拓展名克隆时会被去除。它表明了本地副本的非裸状态。

```
1. git config
```

`git config` 命令允许你在命令行中配置你的 Git 安装（或是一个独立仓库）。这个命令定义了所有配置，从用户信息到仓库行为等等。一些常见的配置命令如下所列。

用法

```
1. git config user.name <name>
```

定义当前仓库所有提交使用的作者姓名。通常来说，你希望使用 `--global` 标记设置当前用户的配置项。

```
1. git config --global user.name <name>
```

定义当前用户所有提交使用的作者姓名。

```
1. git config --global user.email <email>
```

定义当前用户所有提交使用的作者邮箱。

```
1. git config --global alias.<alias-name> <git-command>
```

为Git命令创建一个快捷方式（别名）。

```
1. git config --system core.editor <editor>
```

定义当前机器所有用户使用命令时用到的文本编辑器，如 `git commit`。 `<editor>` 参数用编辑器的启动命令（如 `vi`）替代。

```
1. git config --global --edit
```


用文本编辑器打开全局配置文件，手动编辑。

讨论

所有配置项都储存在纯文本文件中，所以 `git config` 命令其实只是一个提供便捷的命令行接口。通常，你只需要在新机器上配置一次 Git 安装，以及，你通常会想要使用 `--global` 标记。

Git 将配置项保存在三个单独的文件中，允许你分别对单个仓库、用户和整个系统设置。

- `/.git/config` - 特定仓库的设置。
- `~/.gitconfig` - 特定用户的设置。这也是 `--global` 标记的设置项存放的位置。
- `$(prefix)/etc/gitconfig` - 系统层面的设置。

当这些文件中的配置项冲突时，本地仓库设置覆盖用户设置，用户设置覆盖系统设置。如果你打开期中一份文件，你会看到下面这些：

```
1. [user]
2.
3. name = John Smith
4.
5. email = john@example.com
6.
7. [alias]
8.
9. st = status
10.
11. co = checkout
12.
13. br = branch
14.
15. up = rebase
16.
17. ci = commit
18.
19. [core]
20.
21. editor = vim
```

你可以用 `git config` 手动编辑这些值。

栗子

你在安装 Git 之后想要做的第一件事是告诉它你的名字和邮箱，个性化一些默认设置。一般初始的设置过程看上去是这样的：

```
1. # 告诉Git你是谁
2.
3. git config --global user.name "John Smith"
```

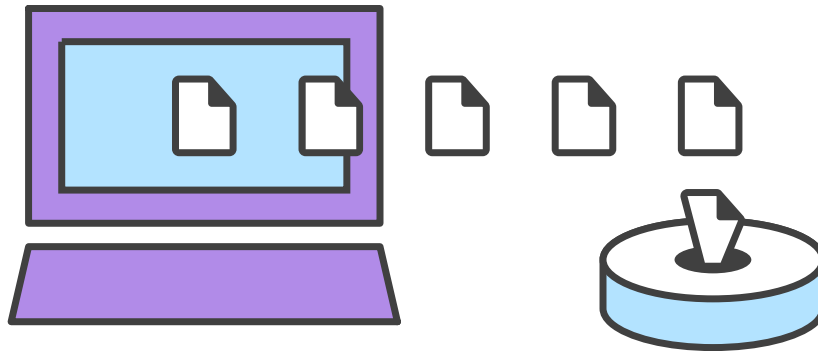
```
4.  
5. git config --global user.email john@example.com  
6.  
7. # 选择你喜欢的文本编辑器  
8.  
9. git config --global core.editor vim  
10.  
11. # 添加一些快捷方式(别名)  
12.  
13. git config --global alias.st status  
14.  
15. git config --global alias.co checkout  
16.  
17. git config --global alias.br branch  
18.  
19. git config --global alias.up rebase  
20.  
21. git config --global alias.ci commit
```

它会生成上一节中所说的 `~/.gitconfig` 文件。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。



`git add` / `git commit` / `git diff` / `git stash` / `.gitignore`

童仲毅 | 2018 年 10 月 26 日（部分章节未更新）

© 本文演绎自 Atlassian 编写的 [Saving Changes](#)。页面上所有内容采用知识共享-署名 (CC BY 2.5 AU) 许可协议。

“保存”这个概念在 Git 等版本控制系统和 Word 等文本编辑应用中不太一样。传统软件里的“保存”在 Git 里被叫做“提交”（commit）。我们常说的保存可以理解成在文件系统中覆盖一个已有的文件或者创建一个新的文件。而在 Git 中，提交这个操作作用于若干个文件和目录。

在 Git 和 SVN 里保存更改也不一样。SVN 提交或检入（check-in）将会推送到远端的中央服务器。也就是说 SVN 的提交需要联网才能完全“保存”项目更改。Git 提交可以在本地完成，然后再使用 `git push -u origin master` 命令推送到远端服务器。这两种方法的区别体现了两种架构设计的本质区别。Git 是一个分布式的应用，而 SVN 是一个中心化的应用。分布式应用一般来说更可靠，因为它们不存在中央服务器这样的单点故障。

`git add`、`git status` 和 `git commit` 这三个命令通常一起使用，将 Git 项目当前的状态保存成一份快照。

Git 还有另一个保存机制：“储藏”（stash）。储藏是一个临时的储存区域，保存还没准备好提交的更改。储藏操作作用于工作目录，三个文件树中的第一棵。它有很多用法，访问 [git stash](#) 页面了解更多。

Git 仓库可以通过设置忽略一些文件或目录。Git 将不会保存这些文件的任何更改。Git 有多种方式管理忽略文件列表。访问 [git ignore](#) 页面了解更多 Git 忽略文件设置。

git add

`git add` 命令将工作目录中的变化添加到暂存区。它告诉 Git 你想要在下一次提交时包含这个文件的更新。但是，`git add` 不会实质上影响你的仓库—在你运行 `git commit` 前更改都还没有真正被记录。

使用这些命令的同时，你还需要 `git status` 来查看工作目录和暂存区的状态。

用法

```
1. git add <file>
```

将 `<file>` 中的更改加入下次提交的缓存。

```
1. git add <directory>
```

将 `<directory>` 下的更改加入下次提交的缓存。

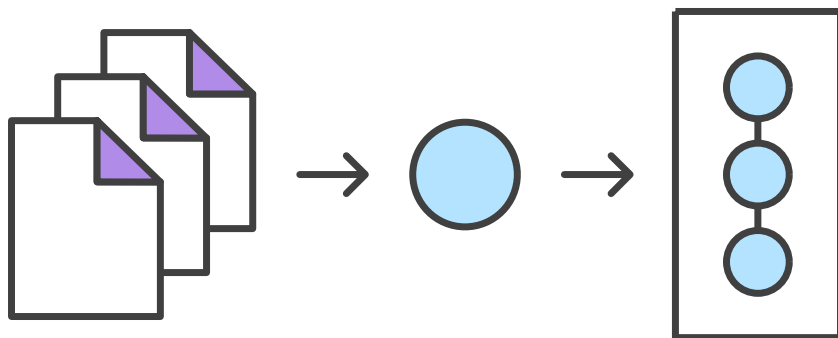
```
1. git add -i
```

开始交互式的缓存，你可以选择文件的一部分加入到下次提交缓存。它会向你展示一堆更改，等待你输入一个命令。`y` 将这块更改加入缓存，`n` 忽略这块更改，`s` 将它分割成更小的块，`e` 手动编辑这块更改，以及`q` 退出。

讨论

`git add` 和 `git commit` 这两个命令组成了最基本的 Git 工作流。每一个 Git 用户都需要理解这两个命令，不管他们团队的协作模型是如何的。我有一千种方式可以将项目版本记录在仓库的历史中。

在一个只有编辑、缓存、提交这样基本流程的项目上开发。首先，你要在工作目录中编辑你的文件。当你准备备份项目的当前状态时，你通过 `git add` 来缓存更改。当你对缓存的快照满意之后，你通过 `git commit` 将它提交到你的项目历史中去。



`git add` 命令不能和 `svn add` 混在一起理解，后者将文件添加到仓库中。而 `git add` 发生于更抽象的更改层面。也就是说，`git add` 在每次你修改一个文件时都需要被调用，而 `svn add` 只需要每个文件调用一次。这听上去很多余，但这样的工作流使得一个项目更容易组织。

缓存区

缓存区是 Git 更为独特的地方之一，如果你是从 SVN (甚至是 Mercurial) 迁移而来，那你可得花点时间理解了。你可以简单地把它想成是工作目录和项目历史之间的缓冲区。

缓存允许你在实际提交到项目历史之前，将相关的更改组合成一份高度专注的快照，而不是将你上次提交以后产生的所有更改一并提交。也就是说你可以更改各种不相关的文件，然后回过去将它们按逻辑切分，将相关的更改添加到缓存，一份一份提交。在任何修改控制系统中，很重要的一点是提交必须是原子性的，以便于追踪 bug，并用最小的代价回滚更改。

栗子

当你开始新项目的时候，`git add` 和 `svn import` 类似。为了创建当前目录的初始提交，使用下面两个命令：

```
1. git add .
2. git commit
```

当你项目设置好之后，新的文件可以通过路径传递给 `git add` 来添加：

```
1. git add hello.py
2. git commit
```

上面的命令同样可以用于记录已有文件的更改。重复一次，Git 不会区分缓存的更改来自新文件，还是仓库中已有的文件。

git commit

`git commit` 命令将缓存的快照提交到项目历史。提交的快照可以认为是项目安全的版本，Git 永远不会改变它们，除非你这么要求。和 `git add` 一样，这是最重要的 Git 命令之一。

尽管和它和 `svn commit` 名字一样，但实际上它们毫无关联。快照被提交到本地仓库，不会和其他 Git 仓库有任何交互。

用法

```
1. git commit
```

提交已经缓存的快照。它会运行文本编辑器，等待你输入提交信息。当你输入信息之后，保存文件，关闭编辑器，创建实际的提交。

```
1. git commit -m "<message>"
```

提交已经缓存的快照。但将 `<message>` 作为提交信息，而不是运行文本编辑器。

```
1. git commit -a
```

提交一份包含工作目录所有更改的快照。它只包含跟踪过的文件的更改（那些之前已经通过 `git add` 添加过的文件）。

讨论

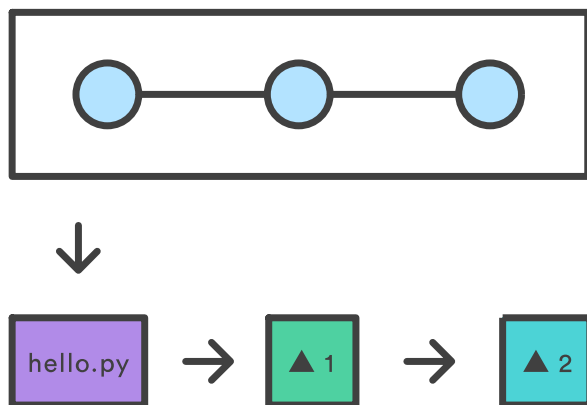
快照总是提交到 本地 仓库。这一点和 SVN 截然不同，后者的工作拷贝提交到中央仓库。而 Git 不会强制你和中央仓库进行交互，直到你准备好了。就像缓存区是工作目录和项目历史之间的缓冲地带，每个开发者的本地仓库是他们贡献的代码和中央仓库之间的缓冲地带。

这一点改变了 Git 用户基本的开发模型。Git 开发者可以在本地仓库中积累一些提交，而不是一发生更改就直接提交到中央仓库。这对于 SVN 风格的协作有着诸多优点：更容易将功能切分成原子性的提交，让相关的提交组合在一起，发布到中央仓库之前整理好本地的历史。开发者得以在一个隔离的环境中工作，直到他们方便的时候再整合代码。

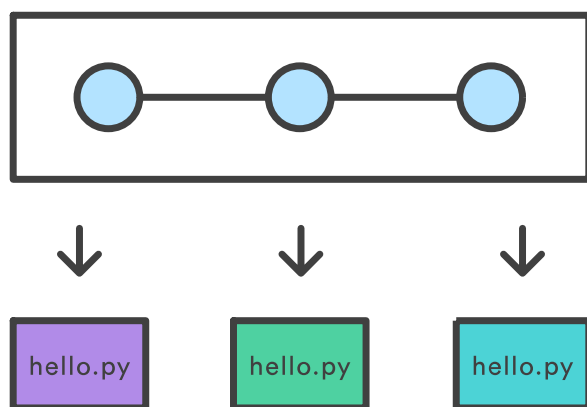
记录快照，而不是记录差异

SVN 和 Git 除了使用上存在巨大差异，它们底层的实现同样遵循截然不同的设计哲学。SVN 追踪文件的 变化 ，而 Git 的版本控制模型基于 快照 。比如说，一个 SVN 提交由仓库中原文件相比的差异（diff）组成。而 Git 在每次提交中记录文件的 完整内容 。

Recording File Diffs (SVN)



Recording Snapshots (Git)



这让很多 Git 操作比 SVN 来的快得多，因为文件的某个版本不需要通过版本间的差异组装得到——每个文件完整的修改能立刻从 Git 的内部数据库中得到。

Git 的快照模型对它版本控制模型的方方面面都有着深远的影响，从分支到合并工具，再到协作工作流，以至于影响了所有特性。

栗子

下面这个栗子假设你编辑了 `hello.py` 文件的一些内容，并且准备好将它提交到项目历史。首先，你需要用 `git add` 缓存文件，然后提交缓存的快照。

```
1. git add hello.py
2. git commit
```

它会打开一个文件编辑器（可以通过 `git config` 设置）询问提交信息，同时列出将被提交的文件。

```
1. # Please enter the commit message for your changes. Lines starting
2. # with '#' will be ignored, and an empty message aborts the commit.
3. # On branch master
4. # Changes to be committed:
5. # (use "git reset HEAD <file>..." to unstage)
6. #
7. #modified: hello.py
```

Git 对提交信息没有特定的格式限制，但约定俗成的格式是：在第一行用 50 个以内的字符总结这个提交，留一空行，然后详细阐述具体的更改。比如：

```
1. Change the message displayed by hello.py
2.
3. - Update the sayHello() function to output the user's name
4. - Change the sayGoodbye() function to a friendlier message
```

注意，很多开发者倾向于在提交信息中使用一般现在时态。这样看起来更像是对仓库进行的操作，让很多改写历史的操作更加符合直觉。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

检查仓库状态

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 status

`git status` 命令显示工作目录和缓存区的状态。你可以看到哪些更改被缓存了，哪些还没有，以及哪些还未被 Git 追踪。status 的输出 不会 告诉你任何已提交到项目历史的信息。如果你想看的话，应该使用 `git log` 命令。

用法

```
1. git status
```

列出已缓存、未缓存、未追踪的文件。

讨论

`git status` 是一个相对简单的命令。 它告诉你 `git add` 和 `git commit` 的进展。status 信息还包括了添加缓存和移除缓存的相关指令。样例输出显示了三类主要的 `git status` 输出：

```
1. # On branch master
2. # Changes to be committed:
3. # (use "git reset HEAD <file>..." to unstage)
4. #
5. #modified: hello.py
6. #
7. # Changes not staged for commit:
8. # (use "git add <file>..." to update what will be committed)
9. # (use "git checkout -- <file>..." to discard changes in working directory)
10. #
11. #modified: main.py
12. #
13. # Untracked files:
14. # (use "git add <file>..." to include in what will be committed)
15. #
16. #hello.pyc
```

忽略文件

未追踪的文件通常有两类。它们要么是项目新增但还未提交的文件，要么是像 `.pyc` 、 `.obj` 、 `.exe` 等编译后的二进制文件。显然前者应该出现在 `git status` 的输出中，而后者会让我们困惑究竟发生了什么。

因此，Git 允许你完全忽略这些文件，只需要将路径放在一个特定的 `.gitignore` 文件中。所有想要忽略的文件应该分别写在单独一行，`*` 字符用作通配符。比如，将下面这行加入项目根目录的 `.gitignore` 文件可以避免编译后的 Python 模块出现在 `git status` 中：

```
1. *.pyc
```

栗子

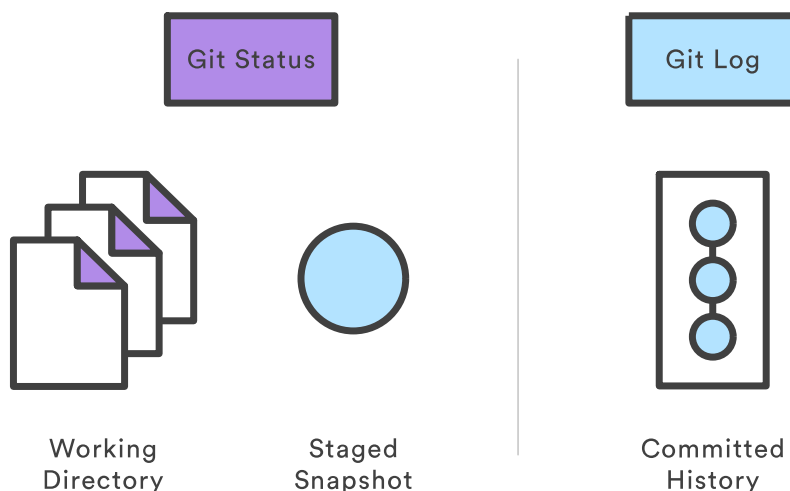
在提交更改前检查仓库状态是一个良好的实践，这样你就不会不小心提交什么奇怪的东西。这个例子显示了缓存和提交快照前后的仓库状态：

```
1. # Edit hello.py
2. git status
3. # hello.py is listed under "Changes not staged for commit"
4. git add hello.py
5. git status
6. # hello.py is listed under "Changes to be committed"
7. git commit
8. git status
9. # nothing to commit (working directory clean)
```

第一个 `status` 的输出显示文件还未缓存。`git add` 操作会影响第二个 `git status`，最后的 `status` 输出告诉你已经没有可以提交的东西了——工作目录和最近的提交一致。一些 Git 命令（比如 `git merge`）需要工作目录整洁，以免意外覆盖更改。

git log

`git log` 命令显示已提交的快照。你可以列出项目历史，筛选，以及搜索特定更改。`git status` 允许你查看工作目录和缓存区，而 `git log` 只作用于提交的项目历史。



`log` 输出可以有很多种自定义的方式，从简单地筛选提交，到用完全自定义的格式显示。其中一些最常用的 `git log` 配置如下所示。

用法

```
1. git log
```

使用默认格式显示完整地项目历史。如果输出超过一屏，你可以用 `空格键` 来滚动，按 `q` 退出。

```
1. git log -n <limit>
```

用 `<limit>` 限制提交的数量。比如 `git log -n 3` 只会显示 3 个提交。

```
1. git log --oneline
```

将每个提交压缩到一行。当你需要查看项目历史的上层情况时这会很有用。

```
1. git log --stat
```

除了 `git log` 信息之外，包含哪些文件被更改了，以及每个文件相对的增删行数。

```
1. git log -p
```

显示代表每个提交的一堆信息。显示每个提交全部的差异（diff），这也是项目历史中最详细的视图。

```
1. git log --author="<pattern>"
```

搜索特定作者的提交。`<pattern>` 可以是字符串或正则表达式。

```
1. git log --grep="<pattern>"
```

搜索提交信息匹配特定 `<pattern>` 的提交。`<pattern>` 可以是字符串或正则表达式。

```
1. git log <since>..<until>
```

只显示发生在 `<since>` 和 `<until>` 之间的提交。两个参数可以是提交 ID、分支名、`HEAD` 或是任何一种引用。

```
1. git log <file>
```

只显示包含特定文件的提交。查找特定文件的历史这样做会很方便。

```
1. git log --graph --decorate --oneline
```

还有一些有用的选项。`--graph` 标记会绘制一幅字符组成的图形，左边是提交，右边是提交信息。`--decorate` 标记会加上提交所在的分支名称和标签。`--oneline` 标记将提交信息显示在同一行，一目了然。

讨论

`git log` 命令是 Git 查看项目历史的基本工具。当你要寻找项目特定的一个版本或者弄明白合并功能分支时引入了哪些变化，你就会用到这个命令。

```
1. commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7
2. Author: John Smith
```

大多数时候都很简单直接。但是，第一行需要解释下。`commit` 后面 40 个字的字符串是提交内容的 SHA-1 校验总和（checksum）。它有两个作用。一是保证提交的正确性——如果它被损坏了，提交会生成一个不同的校验总和。第二，它是提交唯一的标识 ID。

这个 ID 可以用于 `git log` 这样的命令中来引用具体的提交。比如，`git log 3157e..5ab91` 会显示所有ID在 `3157e` 和 `5ab91` 之间的提交。除了校验总和之外，分支名、HEAD 关键字也是常用的引用提交的方法。`HEAD` 总是指向当前的提交，无论是分支还是特定提交也好。

`~`字符用于表示提交的父节点的相对引用。比如，`3157e~1` 指向 `3157e` 前一个提交，`HEAD~3` 是当提交回溯3个节点的提交。

所有这些标识方法的背后都是为了让你对特定提交进行操作。`git log` 命令一般是这些交互的起点，因为它让你找到你想要的提交。

栗子

用法 一节提供了 `git log` 很多的栗子，但请记住，你可以将很多选项用在同一个命令中：

```
1. git log --author="John Smith" -p hello.py
```

这个命令会显示 `John Smith` 作者对 `hello.py` 文件所做的所有更改的差异比较（diff）。

`..`句法是比较分支很有用的工具。下面的栗子显示了在 `some-feature` 分支而不在 `master` 分支的所有提交的概览。

```
1. git log --oneline master..some-feature
```

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

检出之前的提交

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 checkout

`git checkout` 这个命令有三个不同的作用：检出文件、检出提交和检出分支。在这一章中，我们只关心前两种用法。

检出提交会使工作目录和这个提交完全匹配。你可以用它来查看项目之前的状态，而不改变当前的状态。检出文件使你能够查看某个特定文件的旧版本，而工作目录中剩下的文件不变。

用法

```
1. git checkout master
```

回到 `master` 分支。分支会在下一节中讲到，而现在，你只需要将它视为回到项目「当前」状态的一种方式。

```
1. git checkout <commit> <file>
```

查看文件之前的版本。它将工作目录中的 `<file>` 文件变成 `<commit>` 中那个文件的拷贝，并将它加入缓存区。

```
1. git checkout <commit>
```

更新工作目录中的所有文件，使得和某个特定提交中的文件一致。你可以将提交的哈希字符串，或是标签作为 `<commit>` 参数。这会使你处在分离 `HEAD` 的状态。

讨论

版本控制系统背后的思想就是「安全」地储存项目的拷贝，这样你永远不用担心什么时候不可复原地破坏了你的代码库。当你建立了项目历史之后，`git checkout` 是一种便捷的方式，来将保存的快照「加载」到你的开发机器上去。

检出之前的提交是一个只读操作。在查看旧版本的时候绝不会损坏你的仓库。你项目「当前」的状态在 `master` 上不会变化。在开发的正常阶段，`HEAD` 一般指向 `master` 或是其他的本地分支，但当你检出之前提交的时候，`HEAD` 就不再指向一个分支了——它直接指向一个提交。这被称为「分离 `HEAD`」状态，可以用下图可视化：



在另一方面，检出旧文件不影响你仓库的当前状态。你可以在新的快照中像其他文件一样重新提交旧版本。所以，在效果上，`git checkout` 的这个用法可以用来将单个文件回滚到旧版本。

栗子

查看之前的版本

这个栗子假定你开始了一个疯狂的实验，但你不确定你是否想要保留它。为了帮助你决定，你想看一看你开始实验之前的项目状态。首先，你需要找到你想要看的那个版本的 ID。

```
1. git log --oneline
```

假设你的项目历史看上去和下面一样：

```
1. b7119f2 继续做些丧心病狂的事
2. 872fa7e 做些丧心病狂的事
3. a1e8fb5 对 hello.py 做了一些修改
4. 435b61d 创建 hello.py
5. 9773e52 初始导入
```

你可以这样使用 `git checkout` 来查看「对 hello.py 做了一些修改」这个提交：

```
1. git checkout a1e8fb5
```

这让你的工作目录和 `a1e8fb5` 提交所处的状态完全一致。你可以查看文件，编译项目，运行测试，甚至编辑文件而不需要考虑是否会影响项目的当前状态。你所做的一切 都不会 被保存到仓库中。为了继续开发，你需要回到你项目的「当前」状态：

```
1. git checkout master
```

这里假定了你默认在 `master` 分支上开发，我们会在以后的分支模型中详细讨论。

一旦你回到 `master` 分支之后，你可以使用 `git revert` 或 `git reset` 来回滚任何不想要的更改。

检出文件

如果你只对某个文件感兴趣，你也可以用 `git checkout` 来获取它的一个旧版本。比如说，如果你只想从之前的提交中查看 `hello.py` 文件，你可以使用下面的命令：

```
1. git checkout a1e8fb5 hello.py
```

记住，和检出提交不同，这里 确实 会影响你项目的当前状态。旧的文件版本会显示为「需要提交的更改」，允许你回滚到文件之前的版本。如果你不想保留旧的版本，你可以用下面的命令检出到最近的版本：

```
1. git checkout HEAD hello.py
```

2.5 检出以前的提交

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 **Star** :star2: 或 **Fork** :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

回滚错误的修改

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。

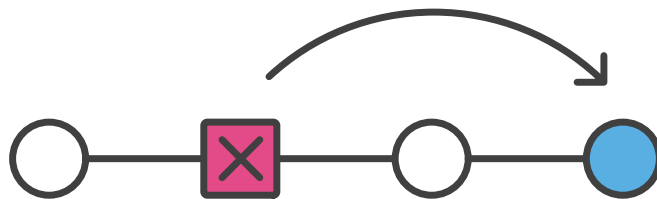
这章教程提供了和项目旧版本打交道所需要的所有技巧。首先，你会知道如何浏览旧的提交，然后了解回滚项目历史中的公有提交和回滚本地机器上的私有更改之间的区别。

git checkout

见上一章「[2.5 检出之前的提交](#)」。

git revert

`git revert` 命令用来撤销一个已经提交的快照。但是，它是通过搞清楚如何撤销这个提交引入的更改，然后在最后加上一个撤销了更改的 新 提交，而不是从项目历史中移除这个提交。这避免了Git丢失项目历史，这一点对于你的版本历史和协作的可靠性来说是很重要的。



.svg)

用法

```
1. git revert <commit>
```

生成一个撤消了 `<commit>` 引入的修改的新提交，然后应用到当前分支。

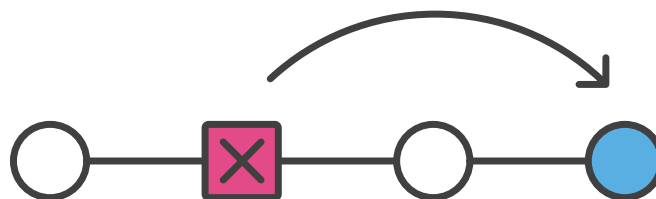
讨论

撤销 (revert) 应该用在你想要在项目历史中移除一整个提交的时候。比如说，你在追踪一个 bug，然后你发现它是由一个提交造成的，这时候撤销就很有用。与其说自己去修复它，然后提交一个新的快照，不如用 `git revert`，它帮你做了所有的事情。

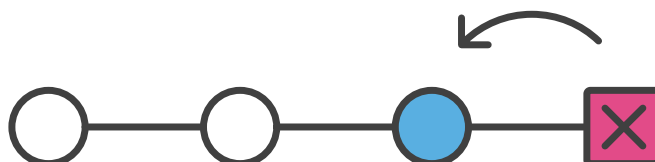
撤销 (revert) 和重设 (reset) 对比

理解这一点很重要。`git revert` 回滚了「单独一个提交」，它没有移除后面的提交，然后回到项目之前的状态。在Git中，后者实际上被称为 `reset`，而不是 `revert`。

Reverting



Resetting



.svg)

撤销和重设相比有两个重要的优点。首先，它不会改变项目历史，对那些已经发布到共享仓库的提交来说这是一个安全的操作。至于为什么改变共享的历史是危险的，请参阅 `git reset` 一节。

其次，`git revert` 可以针对历史中任何一个提交，而 `git reset` 只能从当前提交向前回溯。比如，你想用 `git reset` 重设一个旧的提交，你不得不移除那个提交后的所有提交，再移除那个提交，然后重新提交后面的所有提交。不用说，这并不是一个优雅的回滚方案。

栗子

下面的这个栗子是 `git revert` 一个简单的演示。它提交了一个快照，然后立即撤销这个操作。

```
1. # 编辑一些跟踪的文件
2.
3. # 提交一份快照
4. git commit -m "Make some changes that will be undone"
5.
6. # 撤销刚刚的提交
7. git revert HEAD
```

这个操作可以用下图可视化：

注意第四个提交在撤销后依然在项目历史中。`git revert` 在后面增加了一个提交来撤销修改，而不是删除它。因此，第三和第五个提交表示同样的代码，而第四个提交依然在历史中，以备以后我们想要回到这个提交。

git reset

如果说 `git revert` 是一个撤销更改安全的方式，你可以将 `git reset` 看做一个 危险 的方式。当你用 `git reset` 来重设更改时(提交不再被任何引用或引用日志所引用)，我们无法获得原来的样子—这个撤销是永远的。使用这个工具的时候务必要小心，因为这是少数几个可能会造成工作丢失的命令之一。

和 `git checkout` 一样，`git reset` 有很多种用法。它可以被用来移除提交快照，尽管它通常被用来撤销缓存区和工作目录的修改。不管是哪种情况，它应该只被用于 本地 修改—你永远不应该重设和其他开发者共享的快照。

用法

```
1. git reset <file>
```

从缓存区移除特定文件，但不改变工作目录。它会取消这个文件的缓存，而不覆盖任何更改。

```
1. git reset
```

重设缓冲区，匹配最近的一次提交，但工作目录不变。它会取消 所有 文件的缓存，而不会覆盖任何修改，给你一个重设缓存快照的机会。

```
1. git reset --hard
```

重设缓冲区和工作目录，匹配最近的一次提交。除了取消缓存之外，`--hard` 标记告诉 Git 还要重写所有工作目录中的更改。换句话说：它清除了所有未提交的更改，所以在使用前确定你想扔掉你所有本地的开发。

```
1. git reset <commit>
```

将当前分支的末端移到 `<commit>`，将缓存区重设到这个提交，但不改变工作目录。所有 `<commit>` 之后的更改会保留在工作目录中，这允许你用更干净、原子性的快照重新提交项目历史。

```
1. git reset --hard <commit>
```

将当前分支的末端移到 `<commit>`，将缓存区和工作目录都重设到这个提交。它不仅清除了未提交的更改，同时还清除了 `<commit>` 之后的所有提交。

讨论

上面所有的调用都是用来移除仓库中的修改。没有 `--hard` 标记时 `git reset` 通过取消缓存或取消一系列的提交，然后重新构建提交来清理仓库。而加上 `--hard` 标记对于作了大死之后想要重头再来尤其方便。

撤销(revert)被设计为撤销 公开 的提交的安全方式， `git reset` 被设计为重设 本地 更改。因为两个命令的目的不同，它们的实现也不一样：重设完全地移除了一堆更改，而撤销保留了原来的更改，用一个新的提交来实现撤销。

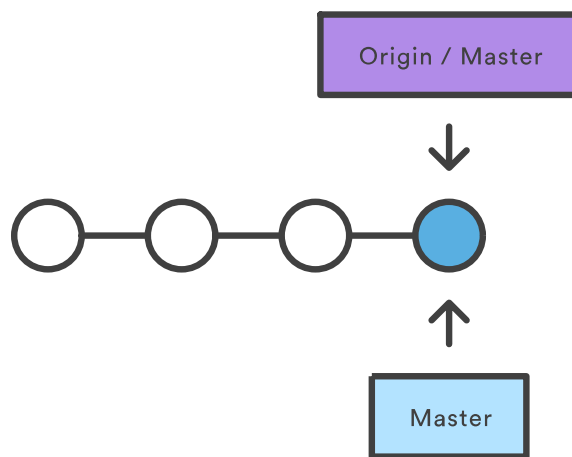


不要重设公共历史

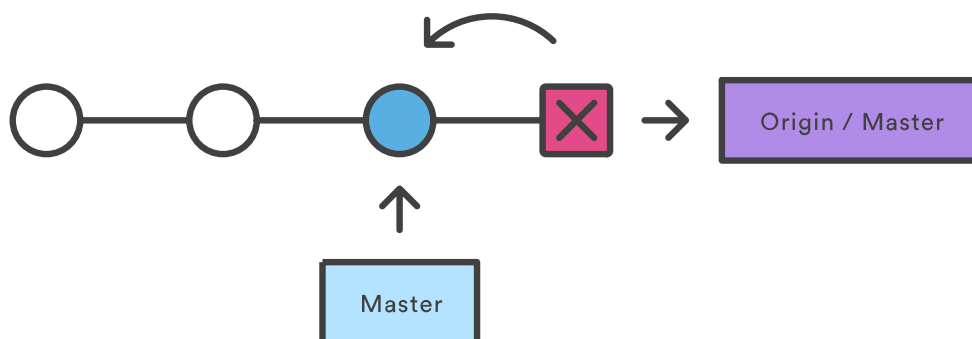
当有 `<commit>` 之后的提交被推送到公共仓库后，你绝不应该使用 `git reset` 。发布一个提交之后，你必须假设其他开发者会依赖于它。

移除一个其他团队成员在上面继续开发的提交在协作时会引发严重的问题。当他们试着和你的仓库同步时，他们会发现项目历史的一部分突然消失了。下面的序列展示了如果你尝试重设公共提交时会发生什么。 `origin/master` 是你本地 `master` 分支对应的中央仓库中的分支。

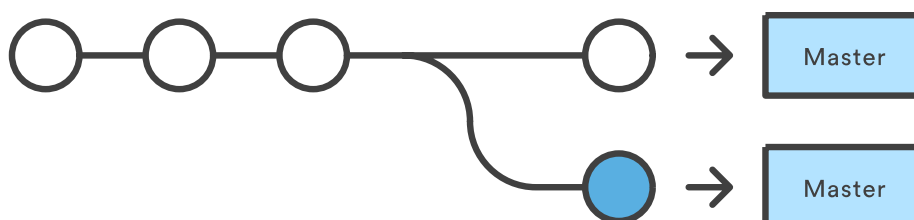
Resetting



After Resetting



After Committing



.svg)

一旦你在重设之后又增加了新的提交，Git 会认为你的本地历史已经和 `origin/master` 分叉了，同步你的仓库时的合并提交（merge commit）会使你的同事困惑。

重点是，确保你只对本地的修改使用 `git reset`，而不是公共更改。如果你需要修复一个公共提交，`git revert` 命令正是被设计来做这个的。

栗子

取消文件缓存

`git reset` 命令在准备缓存快照时经常被用到。下面的例子假设你有两个文件，`hello.py` 和 `main.py` 它们已经被加入了仓库中。

```
1. # 编辑了hello.py和main.py
2.
3. # 缓存了目录下所有文件
4. git add .
5.
6. # 意识到hello.py和main.py中的修改
7. # 应该在不同的快照中提交
8.
9. # 取消main.py缓存
10. git reset main.py
11.
12. # 只提交hello.py
13. git commit -m "Make some changes to hello.py"
14.
15. # 在另一份快照中提交main.py
16. git add main.py
17. git commit -m "Edit main.py"
```

如你所见，`git reset` 帮助你取消和这次提交无关的修改，让提交能够专注于某一特定的范围。

移除本地修改

下面的这个栗子显示了一个更高端的用法。它展示了你作了大死之后应该如何扔掉那几个更新。

```
1. # 创建一个叫`foo.py`的新文件，增加代码
2.
3. # 提交到项目历史
4. git add foo.py
5. git commit -m "Start developing a crazy feature"
6.
7. # 再次编辑`foo.py`，修改其他文件
8.
9. # 提交另一份快照
10. git commit -a -m "Continue my crazy feature"
11.
```

```
12. # 决定废弃这个功能，并删除相关的更改
13. git reset --hard HEAD~2
```

`git reset HEAD~2` 命令将当前分支向前倒退两个提交，相当于在项目历史中移除刚创建的这两个提交。记住，这种重设只能用在 非公开 的提交中。绝不要在将提交推送到共享仓库之后执行上面的操作。

git clean

`git clean` 命令将未跟踪的文件从你的工作目录中移除。它只是提供了一条捷径，因为用 `git status` 查看哪些文件还未跟踪然后手动移除它们也很方便。和一般的 `rm` 命令一样，`git clean` 是无法撤消的，所以在删除未跟踪的文件之前想清楚，你是否真的要这么做。

`git clean` 命令经常和 `git reset --hard` 一起使用。记住，`reset` 只影响被跟踪的文件，所以还需要一个单独的命令来清理未被跟踪的文件。这两个命令相结合，你就可以将工作目录回到之前特定提交时的状态。

用法

```
1. git clean -n
```

执行一次`git clean`的『演习』。它会告诉你那些文件在命令执行后会被移除，而不是真的删除它。

```
1. git clean -f
```

移除当前目录下未被跟踪的文件。`-f`（强制）标记是必需的，除非 `clean.requireForce` 配置项被设为了 `false`（默认为 `true`）。它不会删除 `.gitignore` 中指定的未跟踪的文件。

```
1. git clean -f <path>
```

移除未跟踪的文件，但限制在某个路径下。

```
1. git clean -df
```

移除未跟踪的文件，以及目录。

```
1. git clean -xf
```

移除当前目录下未跟踪的文件，以及 Git 一般忽略的文件。

讨论

如果你在本地仓库中作死之后想要毁尸灭迹，`git reset --hard` 和 `git clean -f` 是你最好的选择。运行这两个命令使工作目录和最近的提交相匹配，让你在干净的状态下继续工作。

`git clean` 命令对于 build 后清理工作目录十分有用。比如，它可以轻易地删除 C 编译器生成的 `.o` 和 `.exe` 二进制文件。这通常是打包发布前需要的一步。`-x` 命令在这种情况下特别方便。

请牢记，和 `git reset` 一样，`git clean` 是仅有的几个可以永久删除提交的命令之一，所以要小心使用。事实上，它太容易丢掉重要的修改了，以至于 Git 厂商 强制 你用 `-f` 标志来进行最基本的操作。这可以避免你用一个 `git clean` 就不小心删除了所有东西。

栗子

下面的栗子清除了工作目录中的所有更改，包括新建还没加入缓存的文件。它假设你已经提交了一些快照，准备开始一些新的实验。

```
1. # 编辑了一些文件
2. # 新增了一些文件
3. # 『糟糕』
4.
5. # 将跟踪的文件回滚回去
6. git reset --hard
7.
8. # 移除未跟踪的文件
9. git clean -df
```

在执行了 `reset/clean` 的流程之后，工作目录和缓存区和最近一次提交看上去一模一样，而 `git status` 会认为这是一个干净的工作目录。你可以重新过来了。

注意，不像 `git reset` 的第二个栗子，新的文件没有被加入到仓库中。因此，它们不会受到 `git reset --hard` 的影响，需要 `git clean` 来删除它们。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

重写项目历史

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。

概述

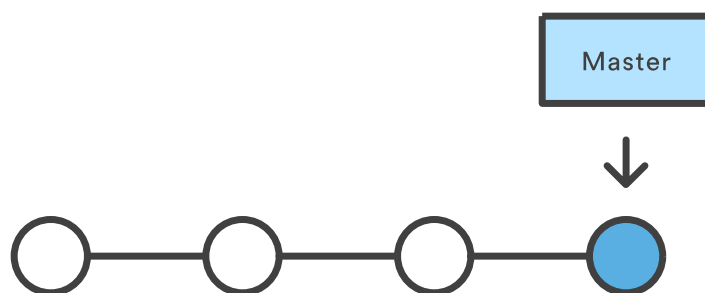
Git 的主要职责是保证你不会丢失提交的修改。但是，它同样被设计成让你完全掌控开发工作流。这包括了让你自定义你的项目历史，而这也创造了丢失提交的可能性。Git 提供了可以重写项目历史的命令，但也警告你这些命令可能会让你丢失内容。

这份教程讨论了重写提交快照的一些常见原因，并告诉你如何避免不好的影响。

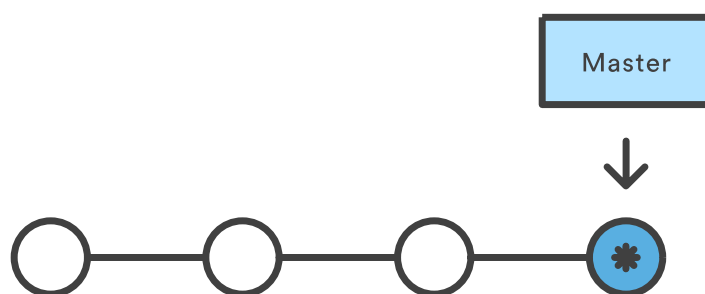
git commit --amend

`git commit --amend` 命令是修复最新提交的便捷方式。它允许你将缓存的修改和之前的提交合并到一起，而不是提交一个全新的快照。它还可以用来简单地编辑上一次提交的信息而不改变快照。

Initial History



Amended History



* Brand New Commits

但是，`amend` 不只是修改了最新的提交——它进行了一次替换。对于 `Git` 来说，这看上去像一个全新的提交，即上图中用星号表示的那一个。在公共仓库工作时一定要牢记这一点。

用法

```
1. git commit --amend
```

合并缓存的修改和上一次的提交，用新的快照替换上一个提交。缓存区没有文件时运行这个命令可以用来编辑上次提交的提交信息，而不会更改快照。

讨论

仓促的提交在你日常开发过程中时常会发生。很容易就忘记了缓存一个文件或者弄错了提交信息的格式。标记是修复这些小意外的便捷方式。

`--amend`

不要修复公共提交

在 `git reset` 这节中，我们说过永远不要重设和其他开发者共享的提交。对于修复也是一样：永远不要修复一个已经推送到公共仓库中的提交。

修复过的提交事实上是全新的提交，之前的提交会被移除出项目历史。这和重设公共快照的后果是一样的。如果你修复了其他开发者在之后继续开发的一个提交，看上去他们的工作基础从项目历史中消失了一样。对于在这上面的开发者来说这是很困惑的，而且很难恢复。

栗子

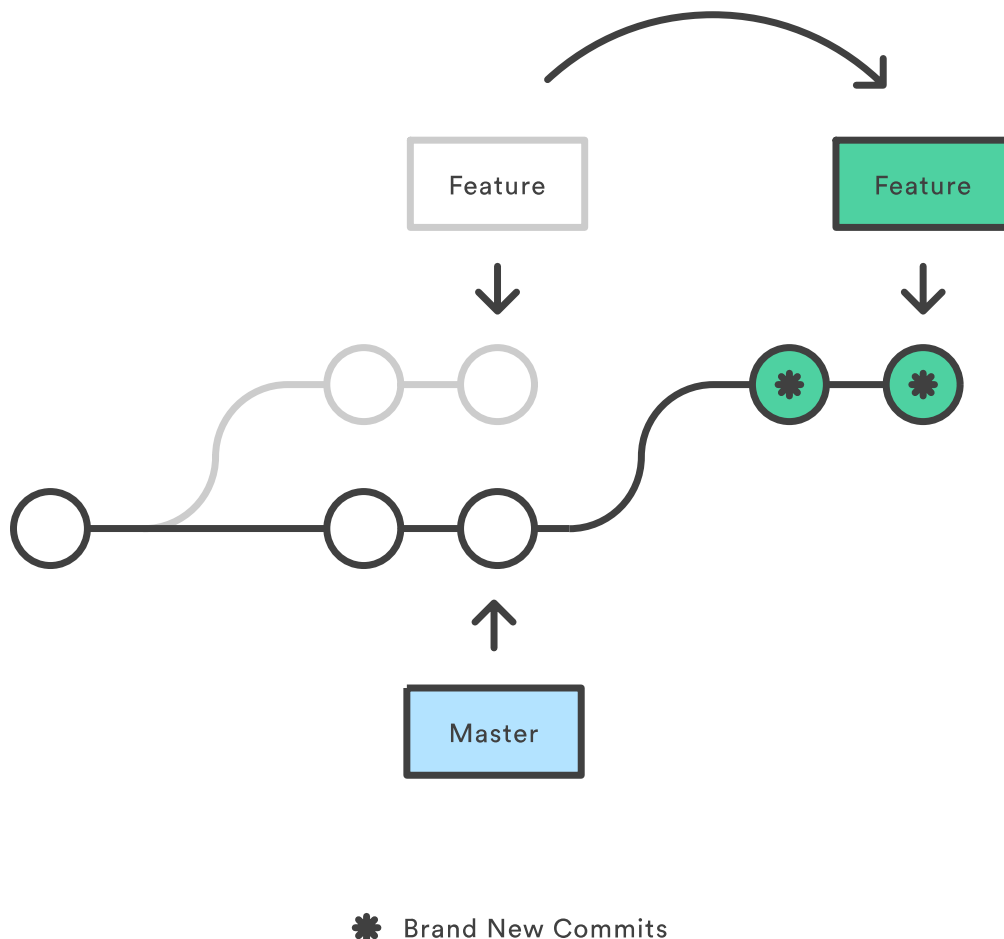
下面这个 展示了 Git 开发工作流中的一个常见情形。我们编辑了一些希望在同一个快照中提交的文件，但我们忘记添加了其中的一个。修复错误只需要缓存那个文件并且用 `--amend` 标记提交：

```
1. # 编辑 hello.py 和 main.py
2. git add hello.py
3. git commit
4.
5. # 意识到你忘记添加 main.py 的更改
6. git add main.py
7. git commit --amend --no-edit
```

编辑器会弹出上一次提交的信息，加入 `--no-edit` 标记会修复提交但不修改提交信息。需要的话你可以修改，不然的话就像往常一样保存并关闭文件。完整的提交会替换之前不完整的提交，看上去就像我们在同一个快照中提交了 `hello.py` 和 `main.py`。

git rebase

变基（rebase，事实上这个名字十分诡异，所以在大多数时候直接用英文术语）是将分支移到一个新的基提交的过程。过程一般如下所示：



从内容的角度来看，rebase 只不过是将分支从一个提交移到了另一个。但从内部机制来看，Git 是通过在选定的基上创建新提交来完成这件事的——它事实上重写了你的项目历史。理解这一点很重要，尽管分支看上去是一样的，但它包含了全新的提交。

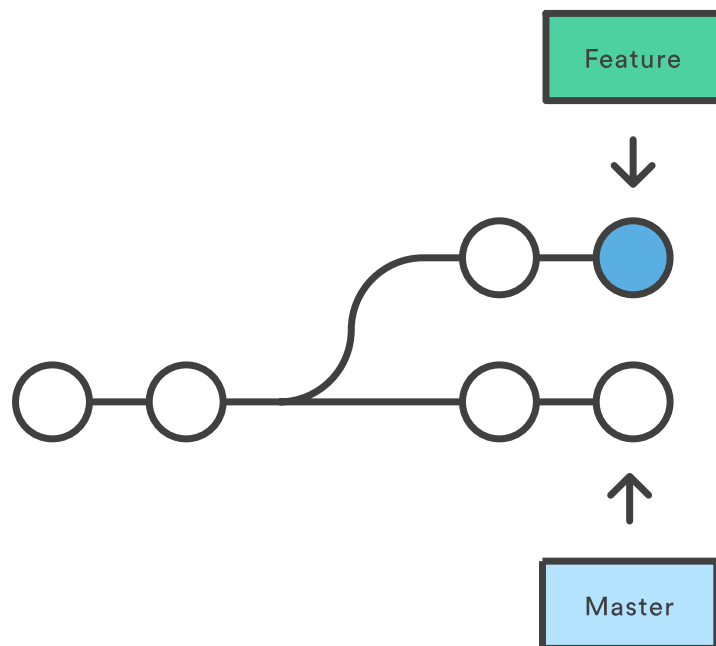
用法

```
1. git rebase <base>
```

将当前分支 rebase 到 `<base>`，这里可以是任何类型的提交引用（ID、分支名、标签，或是 `HEAD` 的相对引用）。

讨论

rebase 的主要目的是为了保持一个线性的项目历史。比如说，当你在 feature 分支工作时 master 分支取得了一些进展：



要将你的 feature 分支整合进 `master` 分支，你有两个选择：直接 merge，或者先 rebase 后 merge。前者会产生一个三路合并（3-way merge）和一个合并提交，而后者产生的是一个快速向前的合并以及完美的线性历史。下图展示了为什么 rebase 到 `master` 分支会促成一个快速向前的合并。



rebase 是将上游更改合并进本地仓库的通常方法。你每次想查看上游进展时，用 `git merge` 拉取上游更新会导致一个多余的合并提交。在另一方面，rebase 就好像是说「我想将我的更改建立在其他人的进展之上」。

不要 rebase 公共历史

和我们讨论过的 `git commit --amend` 和 `git reset` 一样，你永远不应该 rebase 那些已经推送到公共仓库的提交。rebase 会用新的提交替换旧的提交，你的项目历史会像突然消失了一样。

栗子

下面这个 同时使用 `git rebase` 和 `git merge` 来保持线性的项目历史。这是一个确认你的合并都是快速向前的方法。

```
1. # 开始新的功能分支
2. git checkout -b new-feature master
3. # 编辑文件
4. git commit -a -m "Start developing a feature"
```

在 feature 分支开发了一半的时候，我们意识到项目中有一个安全漏洞：

```
1. # 基于master分支创建一个快速修复分支
2. git checkout -b hotfix master
3. # 编辑文件
4. git commit -a -m "Fix security hole"
5. # 合并回master
```

```
6. git checkout master
7. git merge hotfix
8. git branch -d hotfix
```

将 hotfix 分支并回之后 master，我们有了一个分叉的项目历史。我们用 rebase 整合 feature 分支以获得线性的历史，而不是使用普通的 git merge。

```
1. git checkout new-feature
2. git rebase master
```

它将 new-feature 分支移到了 master 分支的末端，现在我们可以对 master 上进行标准的快速向前合并了：

```
1. git checkout master
2. git merge new-feature
```

git rebase -i

用 `-i` 标记运行 `git rebase` 开始交互式 rebase。交互式 rebase 给你在过程中修改单个提交的机会，而不是盲目地将所有提交都移到新的基上。你可以移除、分割提交，更改提交的顺序。它就像是打了鸡血的 `git commit --amend` 一样。

用法

```
1. git rebase -i <base>
```

将当前分支 rebase 到 `base`，但使用可交互的形式。它会打开一个编辑器，你可以为每个将要 rebase 的提交输入命令（见后文）。这些命令决定了每个提交将会怎样被转移到新的基上去。你还可以对这些提交进行排序。

讨论

交互式 rebase 给你了控制项目历史的完全掌控。它给了开发人员很大的自由，因为他们可以提交一个「混乱」的历史而只需专注于写代码，然后回去恢复干净。

大多数开发者喜欢在并入主代码库之前用交互式 rebase 来完善他们的 feature 分支。他们可以将不重要的提交合在一起，删除不需要的，确保所有东西在提交到「正式」的项目历史前都是整齐的。对其他人来说，这个功能的开发看上去是由一系列精心安排的提交组成的。

栗子

下面这个 是 `非交互式rebase` 一节中的可交互升级版。

```
1. # 开始新的功能分支
2. git checkout -b new-feature master
3. # 编辑文件
4. git commit -a -m "Start developing a feature"
```

```
5. # 编辑更多文件
6. git commit -a -m "Fix something from the previous commit"
7.
8. # 直接在 master 上添加文件
9. git checkout master
10. # 编辑文件
11. git commit -a -m "Fix security hole"
12.
13. # 开始交互式 rebase
14. git checkout new-feature
15. git rebase -i master
```

最后的那个命令会打开一个编辑器，包含 new-feature 的两个提交，和一些指示：

```
1. pick 32618c4 Start developing a feature
2. pick 62eed47 Fix something from the previous commit
```

你可以更改每个提交前的 pick 命令来决定在 rebase 时提交移动的方式。在我们的例子中，我们只需要用 squash 命令把两个提交并在一起就可以了：

```
1. pick 32618c4 Start developing a feature
2. squash 62eed47 Fix something from the previous commit
```

保存并关闭编辑器以开始 rebase。另一个编辑器会打开，询问你合并后的快照的提交信息。在定义了提交信息之后，rebase 就完成了，你可以在 `git log` 输出中看到那个提交。整个过程可以用下图可视化：



注意缩并的提交和原来的两个提交的 ID 都不一样，告诉我们这确实是个新的提交。

最后，你可以执行一个快速向前的合并，来将完善的 feature 分支整合进主代码库：

```
1. git checkout master
2. git merge new-feature
```

交互式 rebase 强大的能力可以从整合后的 master 分支看出——额外的 62eed47 提交找不到了。对其他人来说，你就像是一个天才，用完美数量的提交完成了 new-feature。这就是交互式提交如何保持项目历史干净和合意。

git reflog

Git 用引用日志这种机制来记录分支顶端的更新。它允许你回到那些不被任何分支或标签引用的更改。在重写历史后，引用日志包含了分支旧状态的信息，有需要的话你可以回到这个状态。

用法

```
1. git reflog
```

显示本地仓库的引用日志。

```
1. git reflog --relative-date
```

用相对的日期显示引用日志。（如 2 周前）。

讨论

每次当前的 HEAD 更新时（如切换分支、拉取新更改、重写历史或只是添加新的提交），引用日志都会添加一个新条目。

栗子

为了理解 `git reflog`，我们来看一个。

```
1. 0a2e358 HEAD@{0}: reset: moving to HEAD~2
2. 0254ea7 HEAD@{1}: checkout: moving from 2.2 to master
3. c10f740 HEAD@{2}: checkout: moving from master to 2.2
```

上面的引用日志显示了 master 和 2.2 的 branch 之间的相互切换。还有对一个更老的提交的强制重设。最近的活动用 `HEAD@{0}` 标记在上方显示。

如果事实上你是不小心切换回去的，引用日志包含了你意外地丢掉两个提交之前 master 指向的提交 0254ea7。

```
1. git reset --hard 0254ea7
```

使用 `git reset`，就有可能能将master变回之前的那个提交。它提供了一张安全网，以防历史发生意外更改。

务必记住，引用日志提供的安全网只对提交到本地仓库的更改有效，而且只有移动操作会被记录。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

保持同步

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。

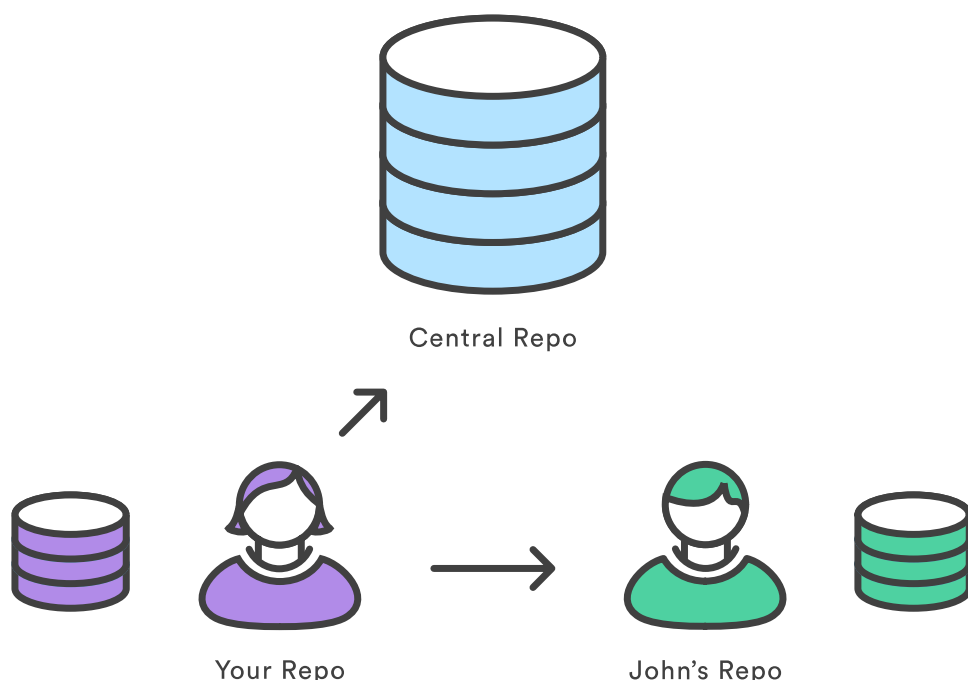
SVN 使用唯一的中央仓库作为开发者之间沟通的桥梁，在开发者的工作拷贝和中央仓库之间传递变更集合 (changeset)，协作得以发生。这和Git的协作模型有所不同，Git 给予每个开发者一份自己的仓库拷贝，拥有自己完整的本地历史和分支结构。用户通常共享一系列的提交而不是单个变更集合。Git 允许你在仓库间共享整个分支，而不是从工作副本提交一个差异集合到中央仓库。

下面的命令让你管理仓库之间的连接，将分支「推送」到其他仓库来发布本地历史，或是将分支「拉取」到本地仓库来查看其它开发者的贡献。

git remote

`git remote` 命令允许你创建、查看和删除和其它仓库之间的连接。远程连接更像是书签，而不是直接跳转到其他仓库的链接。它用方便记住的别名引用不那么方便记住的 URL，而不是提供其他仓库的实时连接。

例如，下图显示了你的仓库和中央仓库以及另一个开发者仓库之间的远程连接。你可以向 Git 命令传递 `origin` 和 `john` 的别名来引用这些仓库，替代完整的 URL。



用法

1. git remote

列出你和其他仓库之间的远程连接。

```
1. git remote -v
```

和上个命令相同，但同时显示每个连接的 URL。

```
1. git remote add <name> <url>
```

创建一个新的远程仓库连接。在添加之后，你可以将 `<name>` 作为 `<url>` 便捷的别名在其他 Git 命令中使用。

```
1. git remote rm <name>
```

移除名为的远程仓库的连接。

```
1. git remote rename <old-name> <new-name>
```

将远程连接从 `<old-name>` 重命名为 `<new-name>`。

讨论

Git 被设计为给每个开发者提供完全隔离的开发环境。也就是说信息并不是自动地在仓库之间传递。开发者需要手动将上游提交拉取到本地，或手动将本地提交推送到中央仓库中去。`git remote` 命令正是将 URL 传递给这些「共享」命令的快捷方式。

名为 origin 的远程连接

当你用 `git clone` 克隆仓库时，它自动创建了一个名为 origin 的远程连接，指向被克隆的仓库。当开发者创建中央仓库的本地副本时非常有用，因为它提供了拉取上游更改和发布本地提交的快捷方式。这也是为什么大多数基于 Git 的项目将它们的中央仓库取名为 origin。

仓库的 URL

Git 支持多种方式来引用一个远程仓库。其中两种最简单的方式便是 HTTP 和 SSH 协议。HTTP 是允许匿名、只读访问仓库的简易方式。比如：

```
1. http://host/path/to/repo.git
```

但是，直接将提交推送到一个 HTTP 地址一般是不可行的（你不太可能希望匿名用户也能随意推送）。如果希望对仓库进行读写，你需要使用 SSH 协议：

```
1. ssh://user@host/path/to/repo.git
```

你需要在托管的服务器上有一个有效的 SSH 账户，但不用麻烦了，Git 支持开箱即用的 SSH 认证连接。

栗子

除了 `origin` 之外，添加你同事的仓库连接通常会带来一些便利。比如，如果你的同事 John 在 `dev.example.com/john.git` 上维护了一个公开的仓库，你可以这样添加连接：

```
1. git remote add john http://dev.example.com/john.git
```

通过这种方式访问每个开发者的仓库，中央仓库之外的协作变得可能。这给维护大项目的小团队带来了极大的便利。

git fetch

`git fetch` 命令将提交从远程仓库导入到你的本地仓库。拉取下来的提交储存为远程分支，而不是我们一直使用的普通的本地分支。你因此可以在整合进你的项目副本之前查看更改。

用法

```
1. git fetch <remote>
```

拉取仓库中所有的分支。同时会从另一个仓库中下载所有需要的提交和文件。

```
1. git fetch <remote> <branch>
```

和上一个命令相同，但只拉取指定的分支。

讨论

当你希望查看其他人的工作进展时，你需要 `fetch`。`fetch` 下来的内容表示为一个远程分支，因此不会影响你的本地开发。这是一个安全的方式，在整合进你的本地仓库之前，检查那些提交。类似于 `svn update`，你可以看到中央仓库的历史进展如何，但它不会强制你将这些进展合并入你的仓库。

远程分支

远程分支和本地分支一样，只不过它们代表这些提交来自于其他人的仓库。你可以像查看本地分支一样查看远程分支，但你会处于分离 HEAD 状态（就像查看旧的提交时一样）。你可以把它们视作只读的分支。如果想要查看远程分支，只需要向 `git branch` 命令传入 `-r` 参数。远程分支拥有 `remote` 的前缀，所以你不会将它们和本地分支混起来。比如，下面的代码片段显示了从 `origin` 拉取之后，你可能想要查看的分支：

```
1. git branch -r
2. # origin/master
3. # origin/develop
4. # origin/some-feature
```

同样，你可以用寻常的 `git checkout` 和 `git log` 命令来查看这些分支。如果你接受远程分支包含的更改，你可以使用 `git merge` 将它并入本地分支。所以，不像 SVN，同步你的本地仓库和远程仓库事实上是一个分两步的操作：先 `fetch`，然后 `merge`。`git pull` 命令是这个过程的快捷方式。

栗子

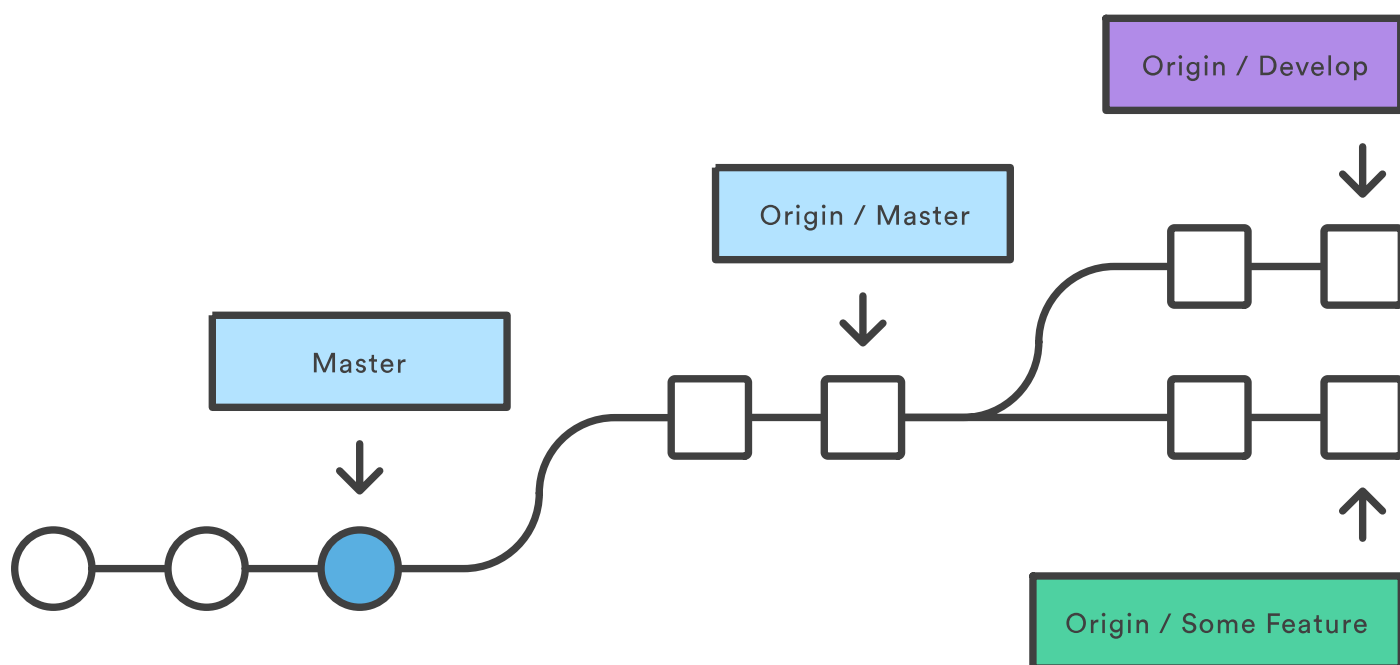
这个例子回顾了同步本地和远程仓库 `master` 分支的常见工作流：

```
1. git fetch origin
```

它会显示会被下载的分支：

```
1. a1e8fb5..45e66a4 master -> origin/master
2. a1e8fb5..9e8ab1c develop -> origin/develop
3. * [new branch] some-feature -> origin/some-feature
```

在下图中，远程分支中的提交显示为方块，而不是圆圈。正如你所见，`git fetch` 让你看到了另一个仓库完整的分支结构。



若想查看添加到上游 `master` 上的提交，你可以运行 `git log`，用 `origin/master` 过滤：

```
1. git log --oneline master..origin/master
```

用下面这些命令接受更改并并入你的本地 `master` 分支：

```
1. git checkout master
2. git log origin/master
```

我们可以使用 `git merge origin/master`：

```
1. git merge origin/master
```

`origin/master` 和 `master` 分支现在指向了同一个提交，你已经和上游的更新保持了同步。

git pull

在基于 Git 的协作工作流中，将上游更改合并到你的本地仓库是一个常见的工作。我们已经知道应该使用 `git fetch`，然后是 `git merge`，但是 `git pull` 将这两个命令合二为一。

用法

```
1. git pull <remote>
```

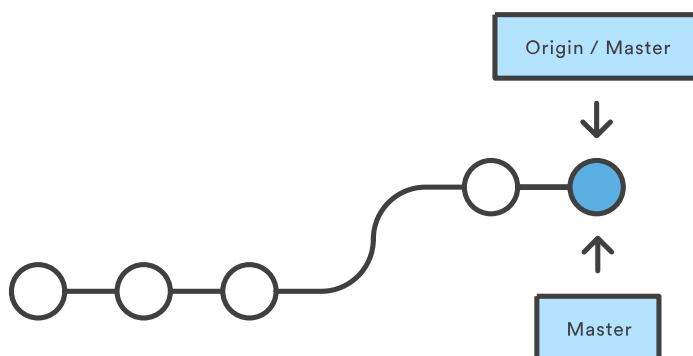
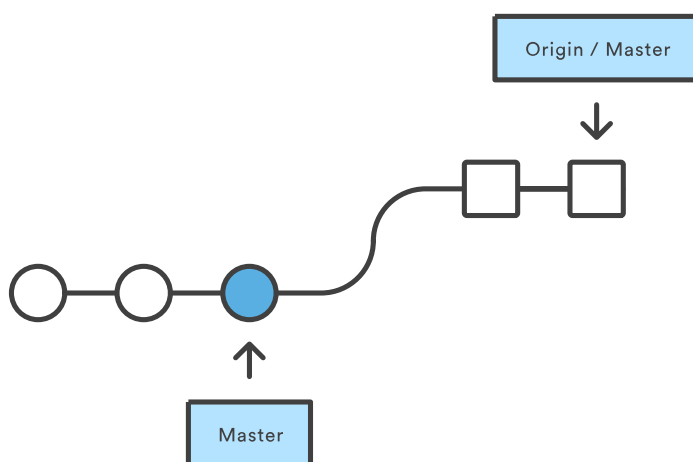
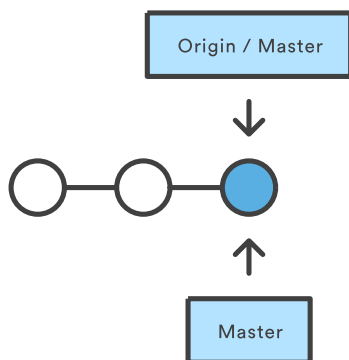
拉取当前分支对应的远程副本中的更改，并立即并入本地副本。效果和 `git fetch` 后接 `git merge origin/.` 一致。

```
1. git pull --rebase <remote>
```

和上一个命令相同，但使用 `git rebase` 合并远程分支和本地分支，而不是使用 `git merge`。

讨论

你可以将 `git pull` 当做 Git 中对应 `svn update` 的命令。这是同步你本地仓库和上游更改的简单方式。下图揭示了 pull 过程中的每一步。



你认为你的仓库已经同步了，但 `git fetch` 发现 origin 中 `master` 的版本在上次检查后已经有了新进展。接着 `git merge` 立即将 `remote master` 并入本地的分支。

基于 Rebase 的 Pull

`--rebase` 标记可以用来保证线性的项目历史，防止合并提交（merge commits）的产生。很多开发者倾向于使用 rebase 而不是 merge，因为「我想要把我的更改放在其他人完成的工作之后」。在这种情况下，与普通的 `git pull` 相比而言，使用带有 `--rebase` 标记的 `git pull` 甚至更像 `svn update`。

事实上，使用 `--rebase` 的 pull 的工作流是如此普遍，以致于你可以直接在配置项中设置它：

```
1. git config --global branch.autosetuprebase always # In git < 1.7.9
2. git config --global pull.rebase true              # In git >= 1.7.9
```

在运行这个命令之后，所有的 `git pull` 命令将使用 `git rebase` 而不是 `git merge`。

栗子

下面的栗子演示了如何和一个中央仓库的 `master branch` 同步：

```
1. git checkout master
2. git pull --rebase origin
```

简单地将你本地的更改放到其他人已经提交的更改之后。

git push

Push 是你将本地仓库中的提交转移到远程仓库中时要做的事。它和 `git fetch` 正好相反，fetch 将提交导入到本地分支，而 push 将提交导出到远程分支。它可以覆盖已有的更改，所以你需要小心使用。这些情况请见下面的讨论。

用法

```
1. git push <remote> <branch>
```

将指定的分支推送到 `<remote>` 上，包括所有需要的提交和提交对象。它会在目标仓库中创建一个本地分支。为了防止你覆盖已有的提交，如果会导致目标仓库非快速向前合并时，Git 不允许你 push。

```
1. git push <remote> --force
```

和上一个命令相同，但即使会导致非快速向前合并也强制推送。除非你确定你所做的事，否则不要使用 `--force` 标记。

```
1. git push <remote> --all
```

将所有本地分支推送到指定的远程仓库。

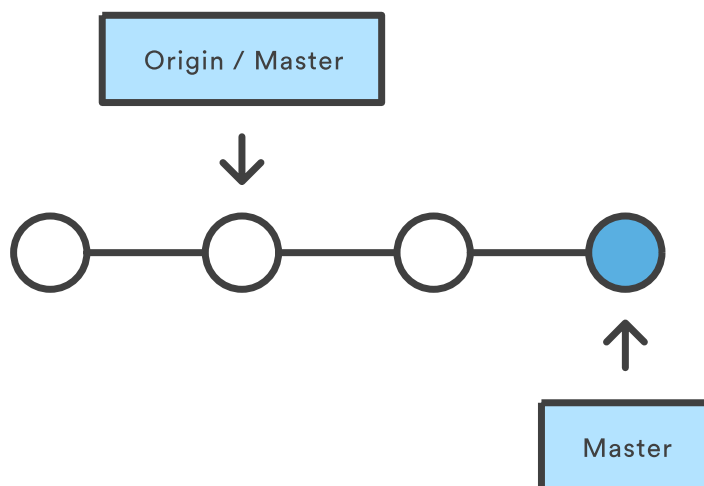
```
1. git push <remote> --tags
```

当你推送一个分支或是使用 `--all` 选项时，标签不会被自动推送上去。`--tags` 将你所有的本地标签推送到远程仓库中去。

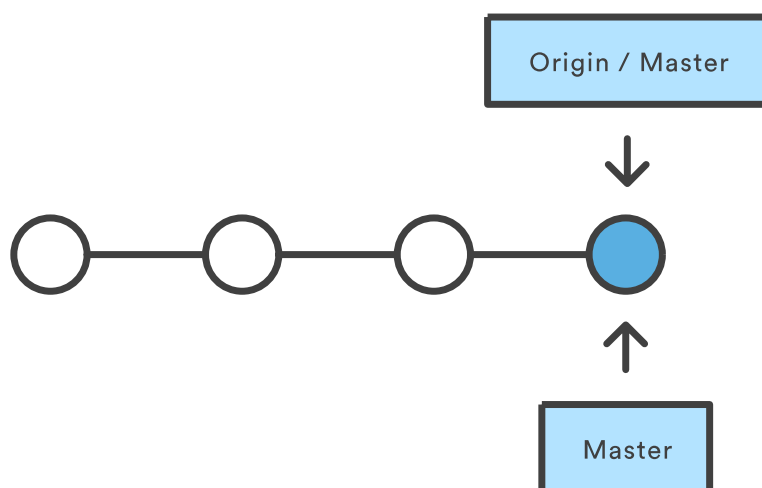
讨论

`git push` 最常见的用法是将你的本地更改发布到中央仓库。在你积累了一些本地提交，准备和同事们共享时，你（可以）用交互式 rebase 来清理你的提交，然后推送到中央仓库去。

Before Pushing



After Pushing



上图显示了当你本地的 `master` 分支进展超过了中央仓库的 `master` 分支，当你运行 `git push origin master` 发布更改时发生的事情。注意，`git push` 和在远程仓库内部运行 `git merge master` 事实上是一样的。

强制推送

Git 为了防止你覆盖中央仓库的历史，会拒绝你会导致非快速向前合并的推送请求。所以，如果远程历史和你本地历史已经分叉，你需要将远程分支 `pull` 下来，在本地合并后再尝试推送。这和 `SVN` 让你在提交更改集合之前要和中央仓库同步是类似的。

`--force` 这个标记覆盖了这个行为，让远程仓库的分支符合你的本地分支，删除你上次 `pull` 之后可能的上游更改。只有当你意识到你刚刚共享的提交不正确，并用 `git commit --amend` 或者交互式 `rebase` 修复之后，你才需要用到强制推送。但是，你必须绝对确定在你使用 `--force` 标记前你的同事们都没有 `pull` 这些提交。

只推送到裸仓库

此外，你只应该推送到那些用 `--bare` 标记初始化的仓库。因为推送会弄乱远程分支结构，很重要的一点是，永远不要推送到其他开发者的仓库。但因为裸仓库没有工作目录，不会发生打断别人的开发之类的事情。

栗子

下面的栗子描述了将本地提交推送到中央仓库的一些标准做法。首先，确保你本地的 `master` 和中央仓库的副本是一致的，提前 `fetch` 中央仓库的副本并在上面 `rebase`。交互式 `rebase` 同样是共享之前清理提交的好机会。接下来，`git push` 命令将你本地 `master` 分支上的所有提交发送给中央仓库。

```
1. git checkout master
2. git fetch origin master
3. git rebase -i origin/master
4. # Squash commits, fix up commit messages etc.
5. git push origin master
```

因为我们已经确信本地的 `master` 分支是最新的，它应该导致快速向前的合并，`git push` 不应该抛出非快速向前之类的问题。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 `Star` :star2: 或 `Fork` :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) 协作说明。

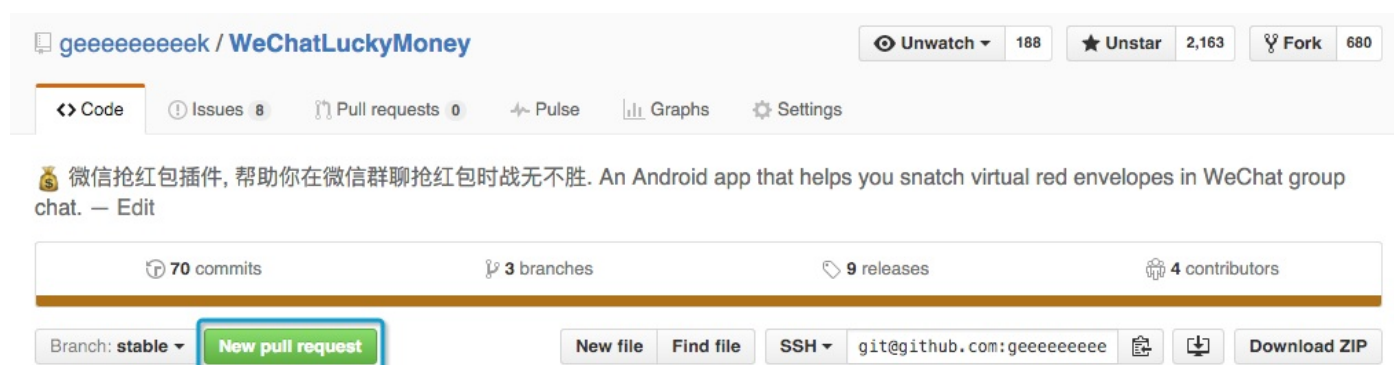
创建Pull Request

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。

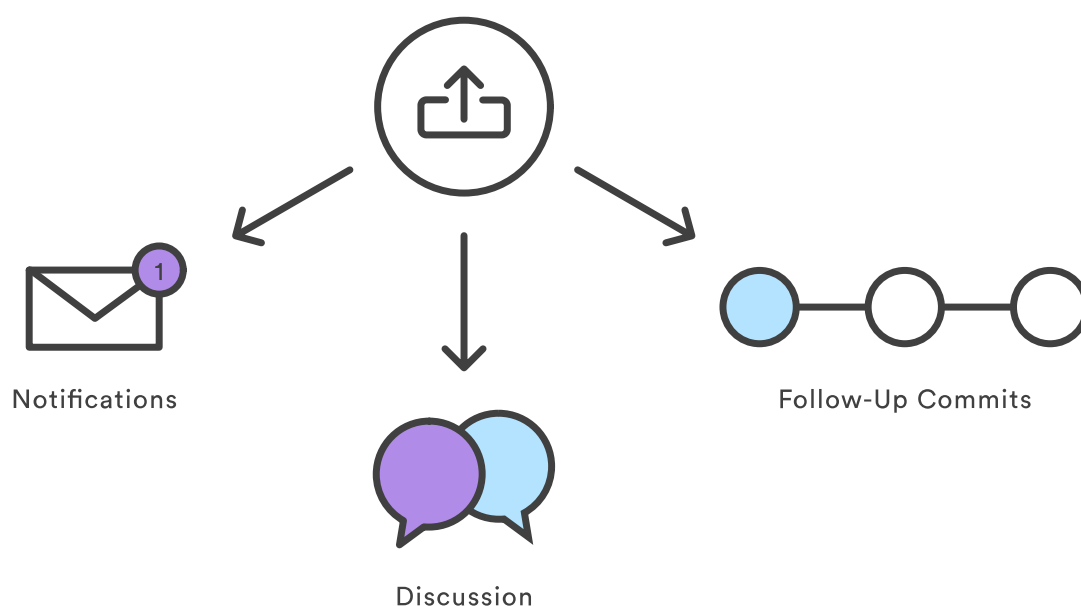
原文以 Bitbucket 为例，考虑到[git-recipes](#)主要面向 GitHub 用户，因此栗子替换成了 GitHub。Pull Request 在 GitLab 等平台上也有，用法和本教程基本一致。

Pull Request 是开发者使用 GitHub 进行协作的利器。这个功能为用户提供了友好的页面，让提议的更改在并入官方项目之前，可以得到充分的讨论。



最简单地来说，Pull Request 是一种机制，让开发者告诉项目成员一个功能已经完成。一旦 feature 分支开发完毕，开发者使用 GitHub 账号提交一个 Pull Request。它告诉所有参与者，他们需要审查代码，并将代码并入 `master` 分支。

但是，Pull Request 不只是一个通知，还是一个专注于某个提议功能的讨论版。如果更改导致了任何问题，团队成员可以在 Pull Request 下发布反馈，甚至推送后续提交来修改这个 Pull Request。所有的活动都在这个 Pull Request 里之间追踪。

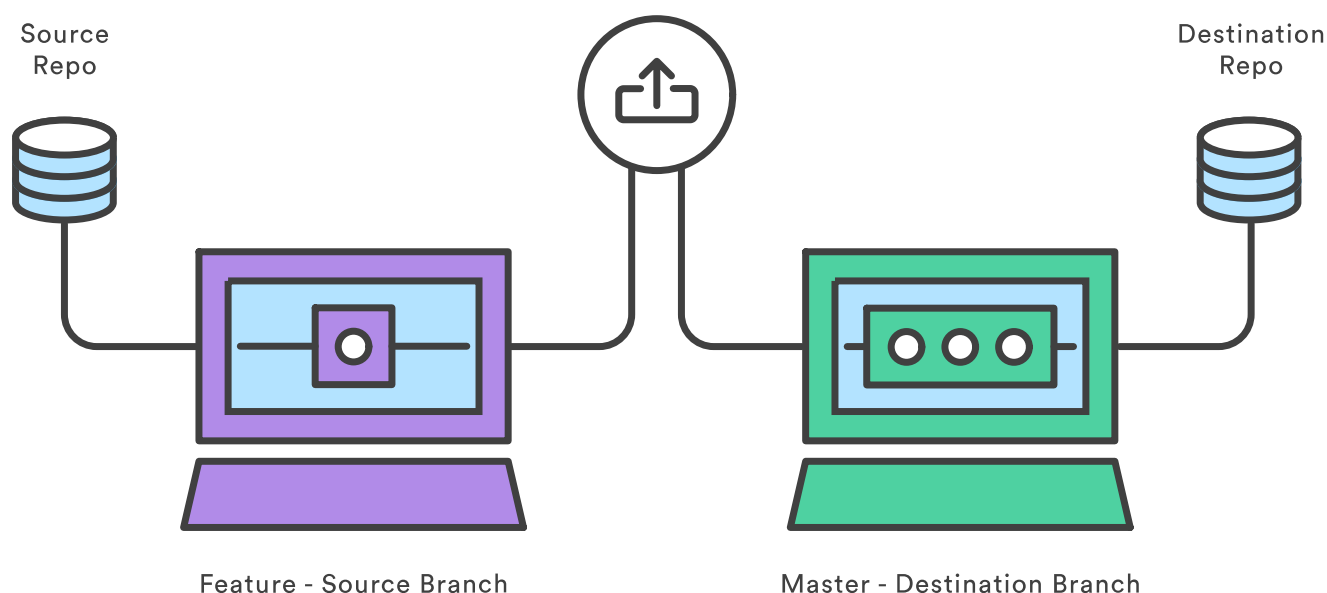


和其他协作模型相比，这种共享提交的解决方案形成了更加线性的工作流。SVN 和 Git 都能通过一个简单的脚本发送通知邮件；但是，如果要讨论更改，开发者不得不在邮件里回复。这会变得愈发杂乱无章，尤其是后续提交出现

时。Pull Request 将所有这些功能放入了一个友好的网页，在每个 GitHub 仓库上方都能找到。

剖析一个 Pull Request

当你提交一个 Pull Request 的时候，你做的事情是 请求 (*request*) 另一个开发者（比如项目维护者）来 拉取 (*pull*) 你仓库中的一个分支到他们的仓库。也就是说你需要提供 4 个信息来完成一个 Pull Request：源仓库、源分支、目标仓库、目标分支。



GitHub 会机智地帮你将一些值设为默认值。但是，取决于你的协作工作流，你的团队可能需要设置不同的值。上图显示了一个请求从 feature 分支合并到官方 master 分支的一个 Pull Request，但除此之外还有好多种使用 Pull Request 的方式。

Pull Request是如何工作的

Pull Request 可以和 feature 分支工作流、GitFlow 工作流或者 Fork 工作流一起使用。但 Pull Request 需要两个不同的分支或是两个不同的仓库，因此它们不能和中心化的工作流一起使用。在不同的工作流中使用 Pull Request 有些不同，但大致的流程如下：

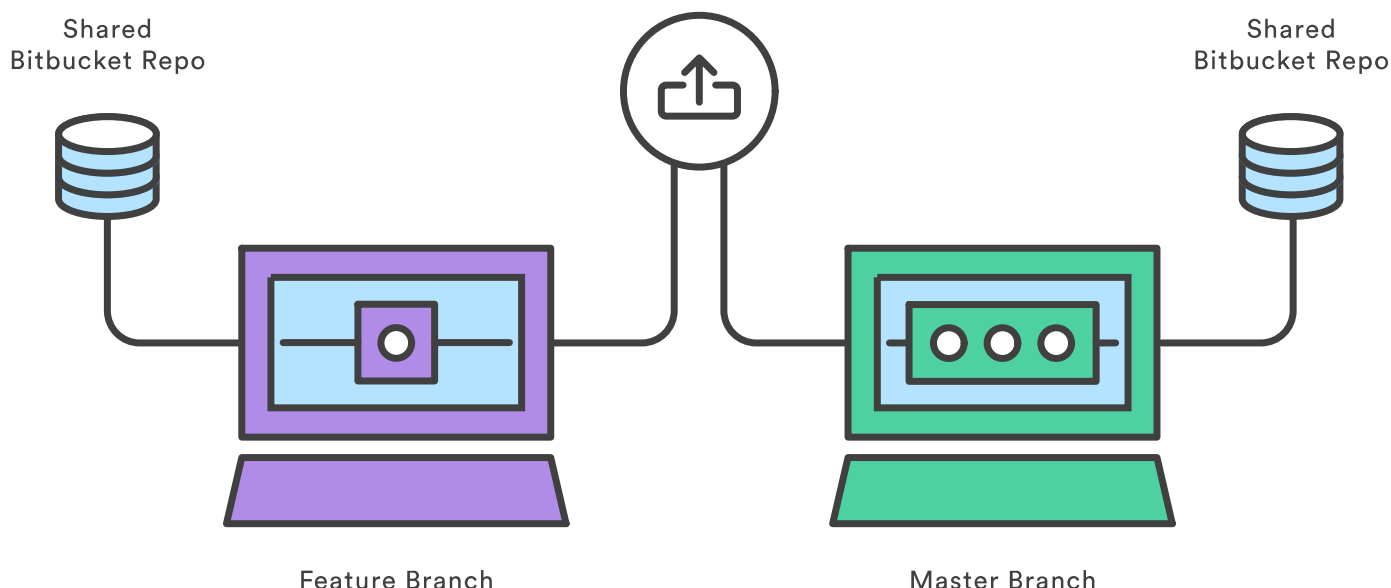
1. 开发者在他们的本地仓库中为某个功能创建一个专门的分支。
2. 开发者将分支推送到公共的 GitHub 仓库。
3. 开发者用 GitHub 发起一个 Pull Request。
4. 其余的团队成员审查代码，讨论并且做出修改。
5. 项目维护者将这个功能并入官方的仓库，然后关闭这个 Pull Request。

下面的章节讨论 Pull Request 在不同的协作工作流中有哪些不同。

Feature 分支工作流中的 Pull Request

Feature 分支工作流使用共享的 GitHub 仓库来管理协作，开发者在单独的 feature 分支中添加功能。开发者在将代码并入主代码库之前，应该发起一个 Pull Request 来启动这个功能的讨论，而不是直接将它们合并到

`master`。



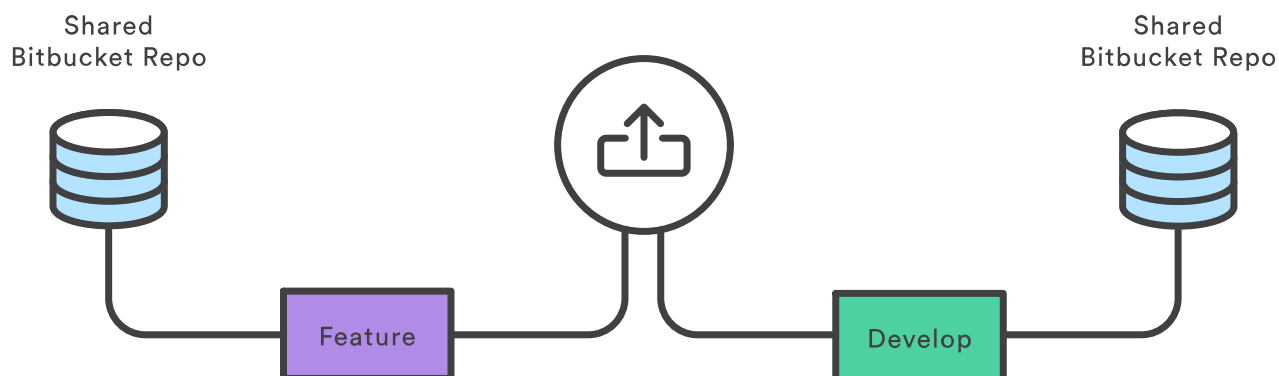
在 Feature 分支工作流中只有一个公共的仓库，因此 Pull Request 的目标和源仓库永远是同一个。一般来说，开发者会将他们的 feature分支作为源分支，`master` 作为目标分支。

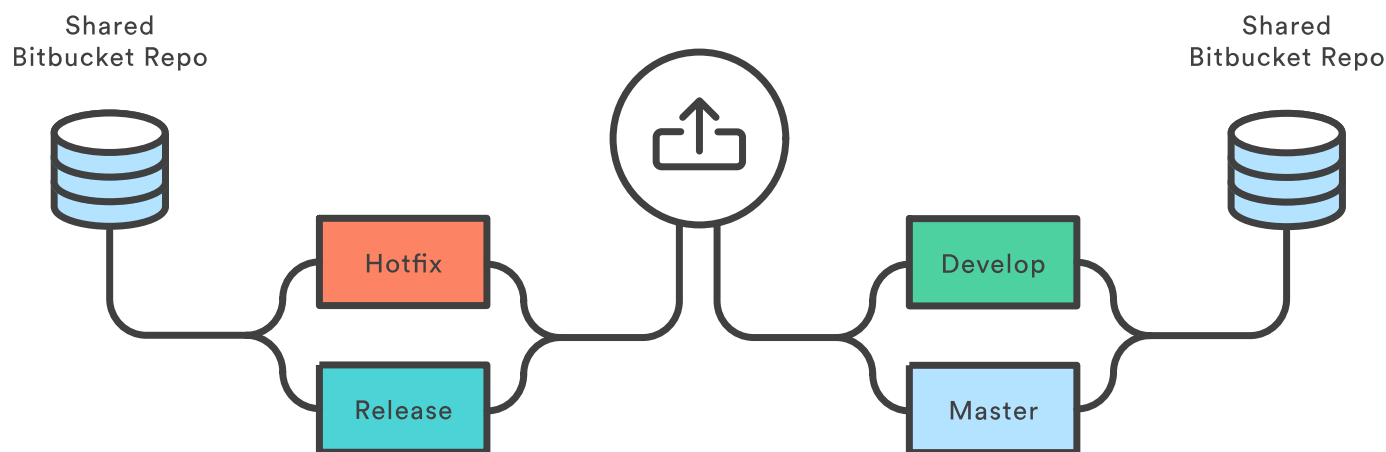
在收到 Pull Request 之后，项目维护者将会做出决定。如果这个功能可以立即发布，他们只需要将代码合并进 `master`，然后关闭 Pull Request 即可。但是，如果提议的更改有一些问题，他们可以在 Pull Request 下发布反馈。后续提交将会显示在相关评论的下方。

你也可以发布一个未成功能的 Pull Request。例如，如果开发者在实现一个特殊的需求时遇到了问题，同样可以发布一个包含工作进展的 Pull Request。其他开发者可以在这个 Pull Request 后面提供建议，甚至自己发布后续的提交来解决这个问题。

GitFlow 工作流中的 Pull Request

GitFlow 工作流和 Feature 分支工作流类似，但定义了围绕项目发布的一个严格的分支模型。在 GitFlow 工作流之上添加 Pull Request 使得开发者方便地讨论发布分支或是所在的维护分支。





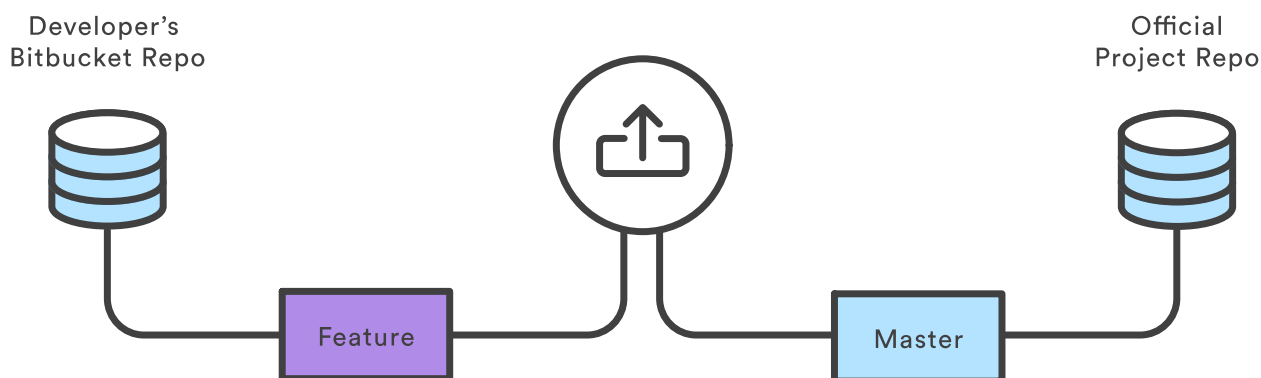
在 GitFlow 工作流中的 Pull Request 和上一节中的完全一致：开发者只需在功能、发布或是快速修复分支需要审查时发布一个 Pull Request，GitHub 会通知到其余的团队成员。

功能一般都会合并到 `develop` 分支，而发布和快速修复分支会被同时合并到 `develop` 和 `master` 当中。Pull Request 可以用来妥善管理这些合并。

Fork 工作流中的 Pull Request

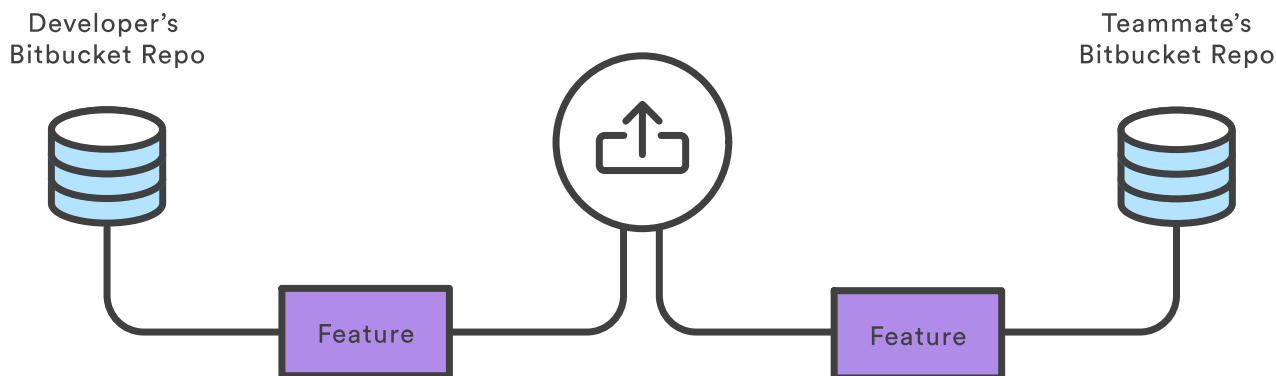
在 Fork 工作流中，开发者将一个完成的功能推送到 他们自己的 仓库，而不是公共的仓库。在这之后，他们发布一个 Pull Request，告诉项目维护者代码需要审查了。

在这个工作流中，Pull Request 的通知作用显得非常有用，因为项目维护者无法获知其他开发者什么时候向他们自己的 GitHub 仓库中添加了提交。



因为每个开发者都有他们自己的公共仓库，Pull Request 的源仓库和目标仓库不是同一个。源仓库是开发者的公开仓库，源分支是包含提议更改的那一个。如果开发者想要将功能合并到主代码库，目标仓库便是官方的项目仓库，目标分支为 `master`。

Pull Request 还可以用来和官方项目之外的开发者进行协作。比如说，一个开发者正在和同事一起开发一个功能，他们可以向 同事的 GitHub 仓库发起一个 Pull Request，而不是官方仓库。他们将 feature 分支同时作为源分支和目标分支。



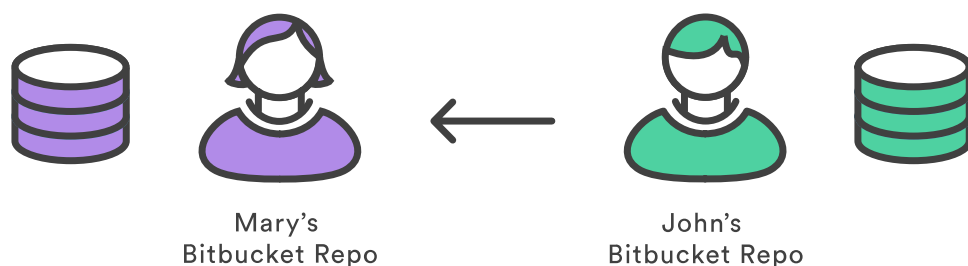
两个开发者可以在 Pull Request 中讨论和开发分支。当功能完成时，其中一位可以发起另一个 Pull Request，请求将功能合并到官方的 master 分支中去。这种灵活性使得 Pull Request 成为了 Fork 工作流中尤为强大的协作工具。

栗子

下面的 演示了如何将 Pull Request 用在 Fork 工作流中。小团队中的开发和向一个开源项目贡献代码都可以这样做。

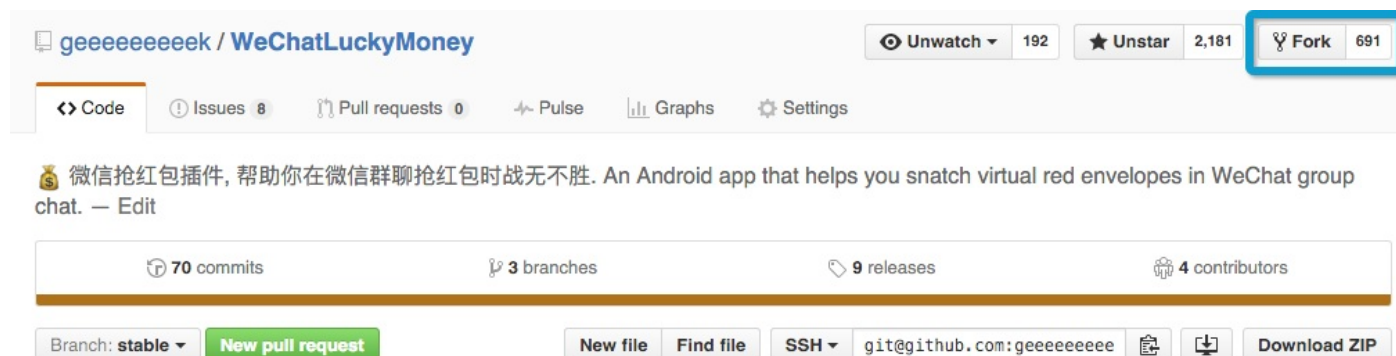
在这个栗子中，Mary 是一位开发者，John 是项目的维护者。他们都有自己公开的 GitHub 仓库，John 的仓库之一便是下面的官方项目。

Mary fork了官方项目



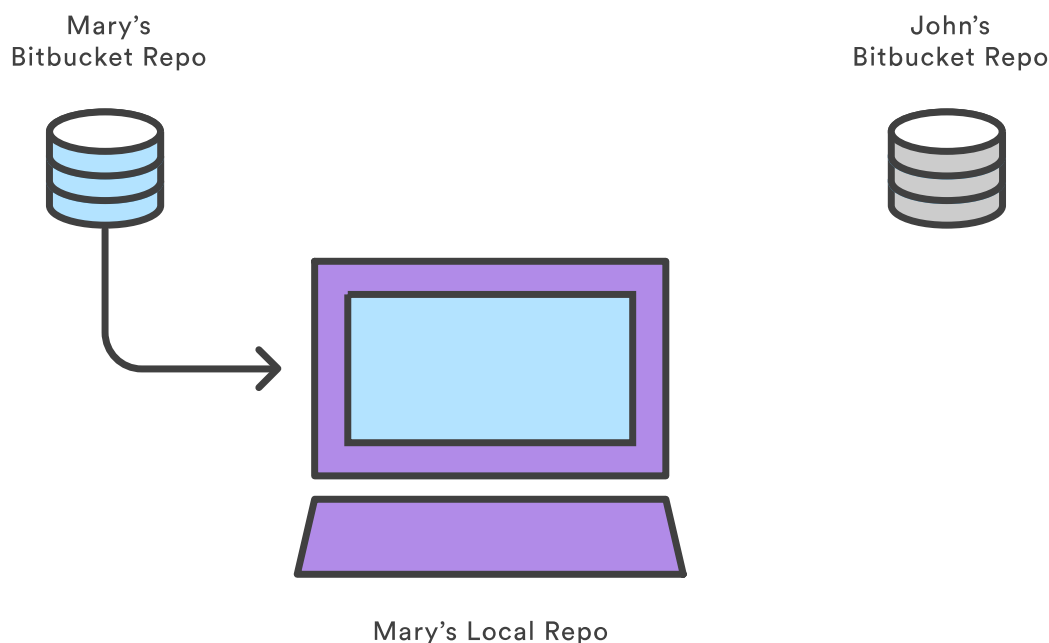
为了参与这个项目，Mary 首先要做的是 fork 属于 John 的 GitHub 仓库。她需要注册登录 GitHub，找到 John 的仓库，点击 Fork 按钮。

下图显示的是 geeeeeeeeek 的 WeChatLuckyMoney 仓库。



选好 fork 的目标位置之后，她在服务端就有了一个项目的副本。

Mary 克隆了她的 GitHub 仓库

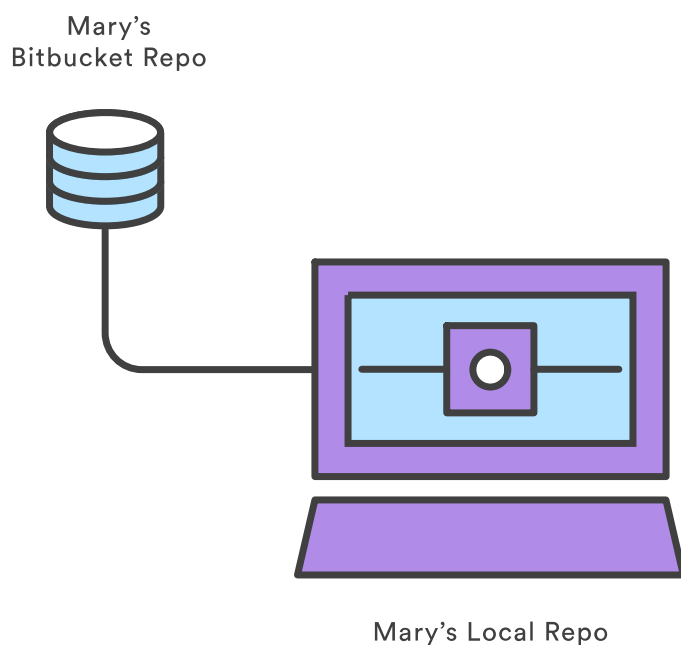


接下来，Mary 需要将她刚刚 fork 的 GitHub 仓库克隆下来。她在本地会有一份项目的副本。她需要运行下面这个命令：

```
1. git clone https://github.com/user/repo.git
```

请记住，`git clone` 自动创建了一个名为 `origin` 的远端连接，指向 Mary 所 fork 的仓库。

Mary 开发了一个新功能



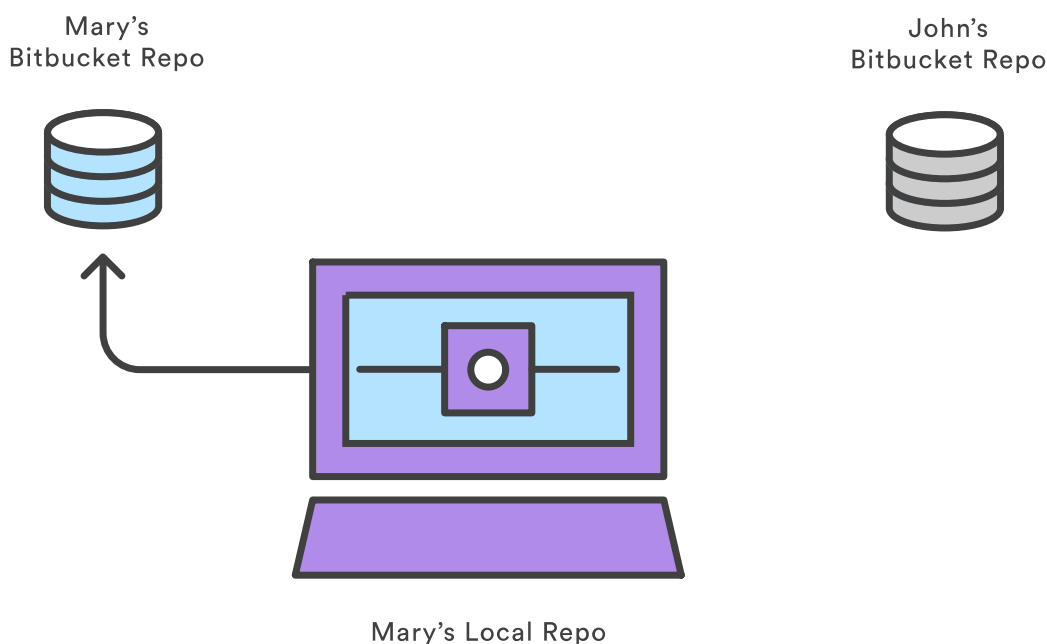
在她写任何代码之前，Mary 需要为这个功能创建一个新的分支。这个分支将是她随后发起 Pull Request 时要用

到的源分支。

```
1. git checkout -b some-feature
2. # 编辑一些代码
3. git commit -a -m "新功能的一些草稿"
```

为了完成这个新功能，Mary 想创建多少个提交都可以。如果 feature 分支的历史有些乱，她可以使用交互式的 rebase 来移除或者拼接不必要的提交。对于大项目来说，清理 feature 的项目历史使得项目维护者更容易看清楚 Pull Request 的所处的进展。

Mary 将 feature 分支推送到了她的 GitHub 仓库

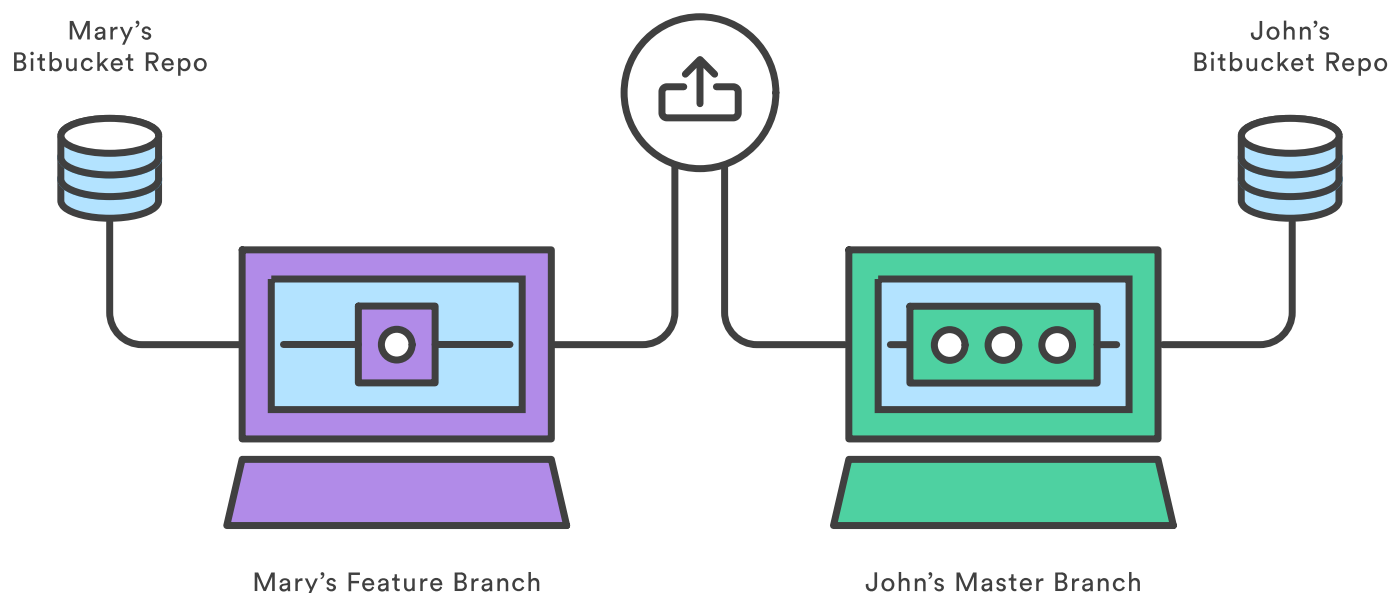


在功能完成后，Mary 使用简单的 `git push` 将 feature 分支推送到了她自己的 GitHub 仓库上（不是官方的仓库）：

```
1. git push origin some-branch
```

这样她的更改就可以被项目维护者看到了（或者任何有权限的协作者）。

Mary 创建了一个 Pull Request



GitHub 上已经有了她的 feature 分支之后，Mary 可以找到被她 fork 的仓库，点击项目简介下的 *New Pull Request* 按钮，用她的 GitHub 账号创建一个 Pull Request。Mary 的仓库会被默认设置为源仓库（head fork），询问她指定源分支（compare）、目标仓库（base fork）和目标分支（base）。

Mary 想要将她的功能并入主代码库，所以源分支就是她的 feature 分支，目标仓库就是 John 的公开仓库，目标分支为 `master`。她还需要提供一个 Pull Request 的标题和简介。

下图展示的是将 LitoMore/demo-project（源仓库）的 develop（源分支）合并到 LitoMore/demo-project（目标仓库）的 master（目标分支）。

LitoMore / demo-project Private

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 1 Projects 0 Wiki Insights Settings

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base: master compare: develop ✔ Able to merge. These branches can be automatically merged.

Add content #1
Add something View pull request

1 commit 1 file changed 0 commit comments 1 contributor

Commits on Feb 11, 2018

LitoMore Add content Verified ba71769

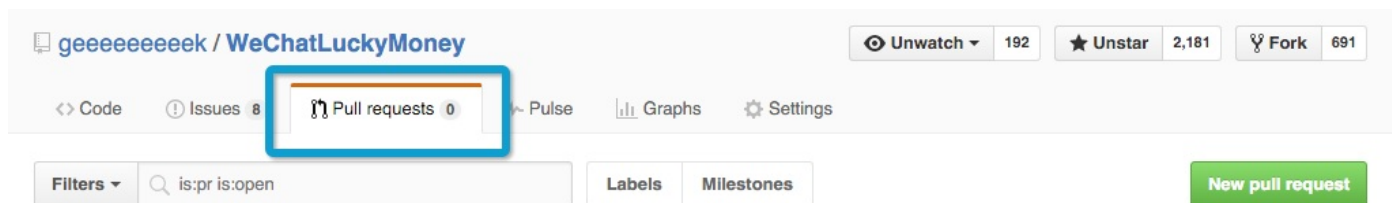
Showing 1 changed file with 1 addition and 0 deletions. Unified Split

```

1 file
... @@ -0,0 +1 @@
1 +test
  
```

在她创建了 Pull Request 之后，GitHub 会给 John 发送一条通知。

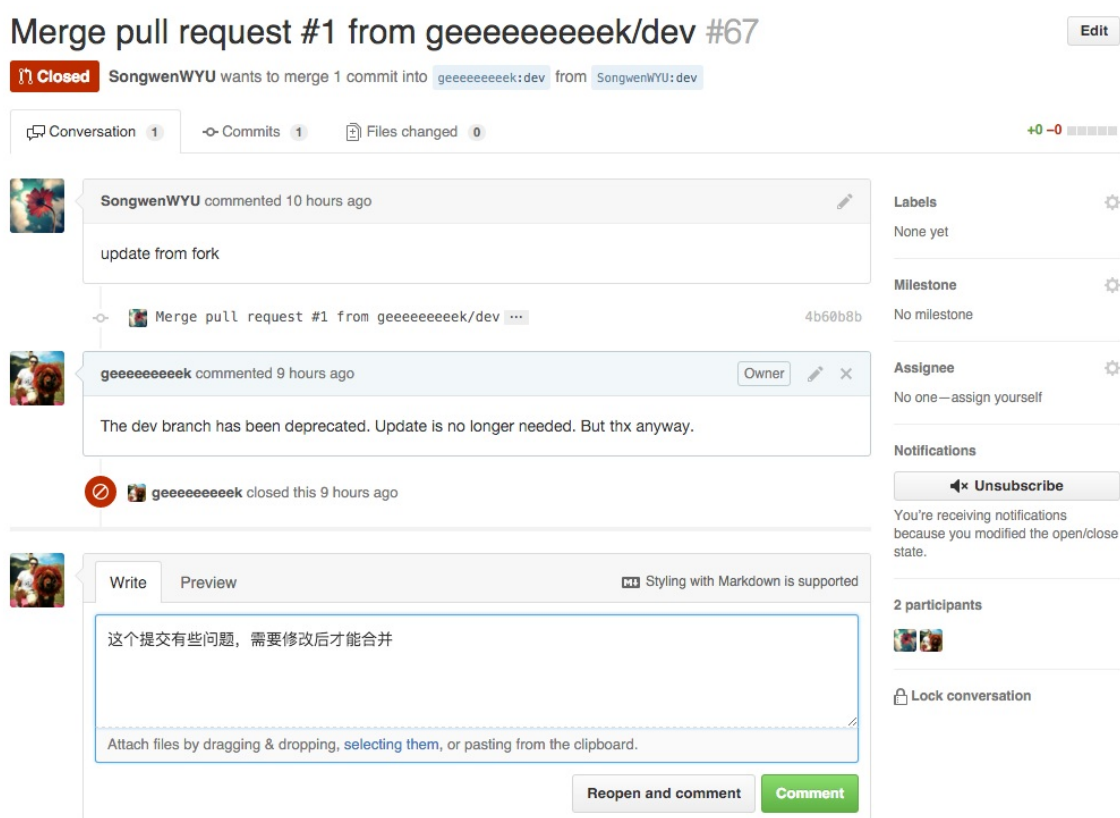
John 审查了这个 Pull Request



John 可以在他自己的 GitHub 仓库下的 *Pull Request* 选项卡中看到所有的 Pull Request。点击 Mary 的 Pull Request 会显示这个 Pull Request 的简介、feature 分支的提交历史，以及包含的更改。

如果他认为 feature 分支已经可以合并了，他只需点击 *Merge Pull Request* 按钮来通过这个 Pull Request，将 Mary 的 feature 分支并入他的 `master` 分支。

但是，在这里栗子中，假设 John 发现了 Mary 代码中的一个小 bug，需要她在合并前修复。他可以评论整个 Pull Request，也可以评论 feature 分支中某个特定的提交。



Mary 添加了一个后续提交

如果 Mary 对这个反馈感到困惑，她可以在 Pull Request 后回复，把这里当做是她的功能的讨论版。

为了修复错误，Mary 在她的 feature 分支后面添加了另一个提交，并将它推送到了她的 GitHub 仓库，就像她之前做的一样。这个提交被自动添加到原来的 Pull Request 后面，John 可以在他的评论下方再次审查这些修改。

John 接受了 Pull Request

最后，John 接受了这些修改，将 feature 分支并入了 master 分支，关闭了这个 Pull Request。功能现在

已经整合到了项目中，其他在 `master` 分支上工作的开发者可以使用标准的 `git pull` 命令将这些修改拉取到自己的本地仓库。

如果你希望实践一下，可以按照上面的流程向这个项目发起一个 Pull Request，修改任何你发现的错误 :smile:

接下来怎么做？

你现在应该已经掌握了如何将你的 Pull Request 整合到你的工作流。记住，Pull Request 不是替代任何 Git 工作流的万金油，而是一种让队员间协作锦上添花的工具。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。



使用分支

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。

这份教程是 Git 分支的综合介绍。首先，我们简单讲解如何创建分支，就像请求一份新的项目历史一样。然后，我们会看到 git checkout 是如何切换分支的。最后，学习一下 git merge 是如何整合独立分支的历史。

我们已经知道，Git 分支和 SVN 分支不同。SVN 分支只被用来记录偶尔大规模的开发效果，而 Git 分支是你日常工作中不可缺失的一部分。

git branch

分支代表了一条独立的开发流水线。分支是我们在第二篇中讨论过的「编辑/缓存/提交」流程的抽象。你可以把它看作请求全新「工作目录、缓存区、项目历史」的一种方式。新的提交被存放在当前分支的历史中，导致了项目历史被 fork 了一份。

`git branch` 命令允许你创建、列出、重命名和删除分支。它不允许你切换分支或是将被 fork 的历史放回去。因此，`git branch` 和 `git checkout`、`git merge` 这两个命令通常紧密地结合在一起使用。

用法

```
1. git branch
```

列出仓库中所有分支。

```
1. git branch <branch>
```

创建一个名为 `<branch>` 的分支。不会自动切换到那个分支去。

```
1. git branch -d <branch>
```

删除指定分支。这是一个安全的操作，Git 会阻止你删除包含未合并更改的分支。

```
1. git branch -D <branch>
```

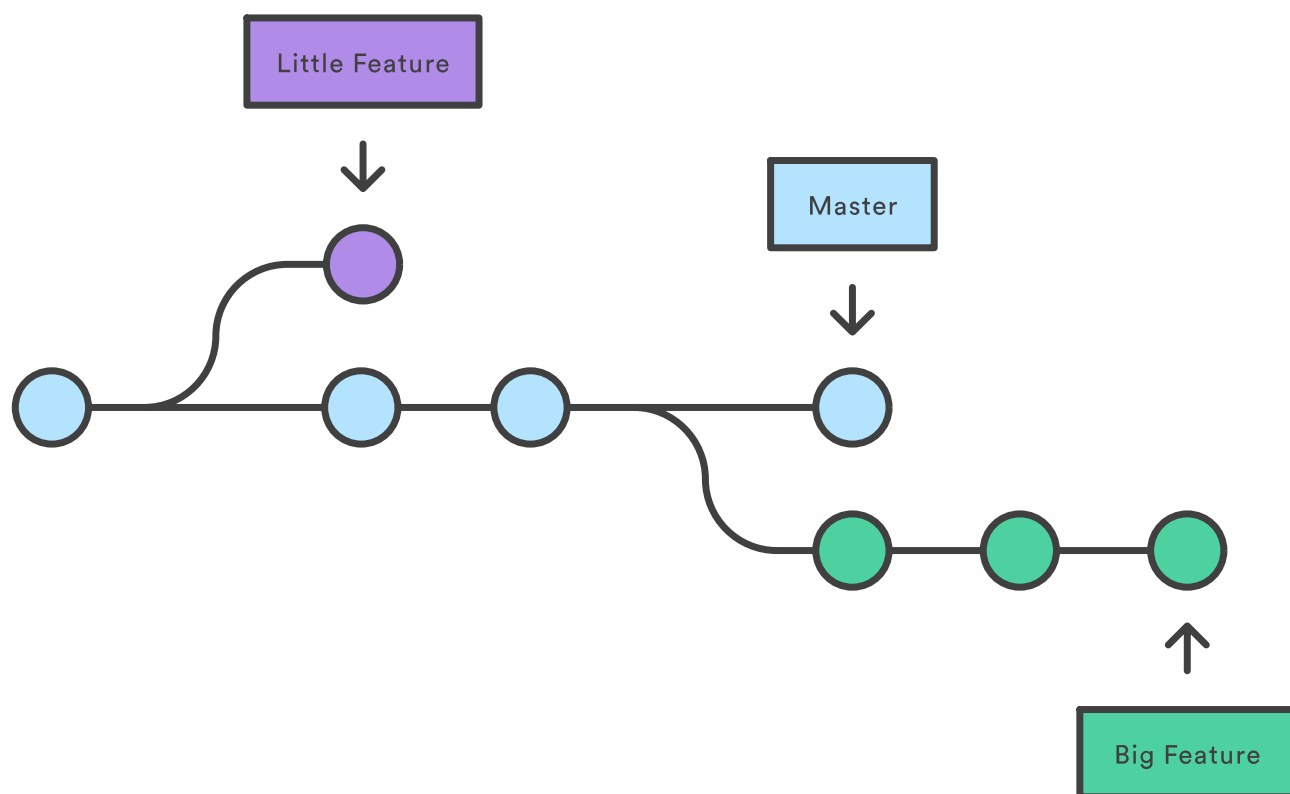
强制删除指定分支，即使包含未合并更改。如果你希望永远删除某条开发线的所有提交，你应该用这个命令。

```
1. git branch -m <branch>
```

将当前分支命名为 `<branch>`。

讨论

在 Git 中，分支是你日常开发流程中的一部分。当你想要添加一个新的功能或是修复一个 bug 时——不管 bug 是大是小——你都应该新建一个分支来封装你的修改。这确保了不稳定的代码永远不会被提交到主代码库中，它同时给了你机会，在并入主分支前清理你 feature 分支的历史。



比如，上图将一个拥有两条独立开发线的仓库可视化，其中一条是一个不起眼的功能，另一条是长期运行的功能。使用分支开发时，不仅可以同时在两条线上工作，还可以保持主要的 `master branch` 不混入奇怪的代码。

分支的顶端

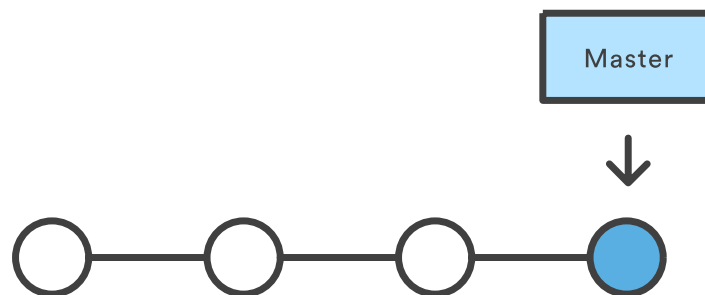
Git 分支背后的实现远比 SVN 的模型要轻量。与其在目录之间复制文件，Git 将分支存为指向提交的引用。换句话说，分支代表了一系列提交的 **顶端**——而不是提交的 **容器**。分支历史通过提交之间的关系来推断。

这使得 Git 的合并模型变成了动态的。SVN 中的合并是基于文件的，而 Git 让你在更抽象的提交层面操作。事实上，你可以看到项目历史中的合并其实是将两个独立的提交历史连接起来。

栗子

创建分支

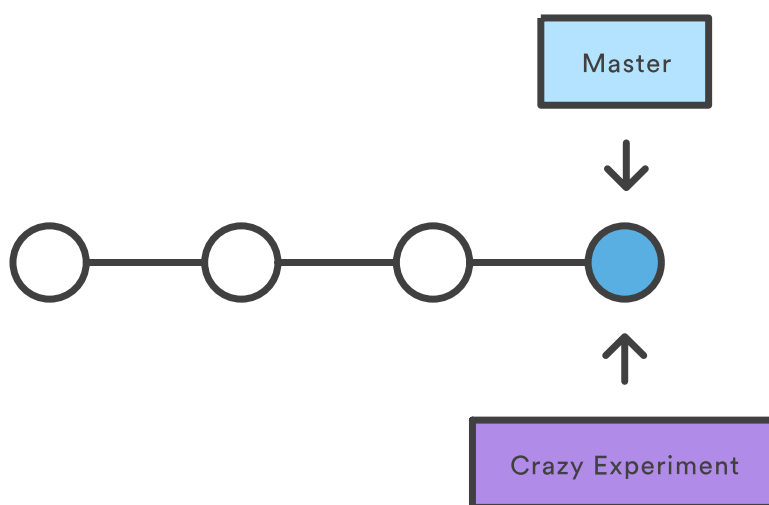
分支只是指向提交的 **指针**，理解这一点很重要。当你创建一个分支是，Git 只需要创建一个新的指针——仓库不会受到任何影响。因此，如果你最开始有这样一个仓库：



接下来你用下面的命令创建了一个分支：

```
1. git branch crazy-experiment
```

仓库历史保持不变。你得到的是一个指向当前提交的新的指针：



注意，这只会 创建 一个新的分支。要开始在上面添加提交，你需要用 `git checkout` 来选中这个分支，然后使用标准的 `git add` 和 `git commit` 命令。

删除分支

一旦你完成了分支上的工作，准备将它并入主代码库，你可以自由地删除这个分支，而不丢失项目历史：

```
1. git branch -d crazy-experiment
```

然后，如果分支还没有合并，下面的命令会产生一个错误信息：

```
1. error: The branch 'crazy-experiment' is not fully merged.
2. If you are sure you want to delete it, run 'git branch -D crazy-experiment'.
```

Git 保护你不会丢失这些提交的引用，或者说丢失访问整条开发线的入口。如果你 真的 想要删除这个分支（比如说这是一个失败的实验），你可以用大写的 `-D` 标记：

```
1. git branch -D crazy-experiment
```

它会删除这个分支，无视它的状态和警告，因此需谨慎使用。

git checkout

`git checkout` 命令允许你切换用 `git branch` 创建的分支。查看一个分支会更新工作目录中的文件，以符合分支中的版本，它还告诉 Git 记录那个分支上的新提交。将它看作一个选中你正在进行的开发的一种方式。

在上一篇中，我们看到了如何用 `git checkout` 来查看旧的提交。「查看分支」和「将工作目录更新到选中的版本/修改」很类似；但是，新的更改 会 保存在项目历史中——这不是一个只读的操作。

用法

```
1. git checkout <existing-branch>
```

查看特定分支，分支应该已经通过 `git branch` 创建。这使得 `<existing-branch>` 成为当前的分支，并更新工作目录的版本。

```
1. git checkout -b <new-branch>
```

创建并查看 `<new-branch>`，`-b` 选项是一个方便的标记，告诉Git在运行 `git checkout <new-branch>` 之前运行 `git branch <new-branch>`。

```
1. git checkout -b <new-branch> <existing-branch>
```

和上一条相同，但将 `<existing-branch>` 作为新分支的基，而不是当前分支。

讨论

`git checkout` 和 `git branch` 是一对好基友。当你想要创建一个新功能时，你用 `git branch` 创建分支，然后用 `git checkout` 查看。你可以在一个仓库中用 `git checkout` 切换分支，同时开发几个功能。

每个功能专门一个分支对于传统 SVN 工作流来说是一个巨大的转变。这使得尝试新的实验超乎想象的简单，不用担心毁坏已有的功能，并且可以同时开发几个不相关的功能。另外，分支可以促进不同的协作工作流。

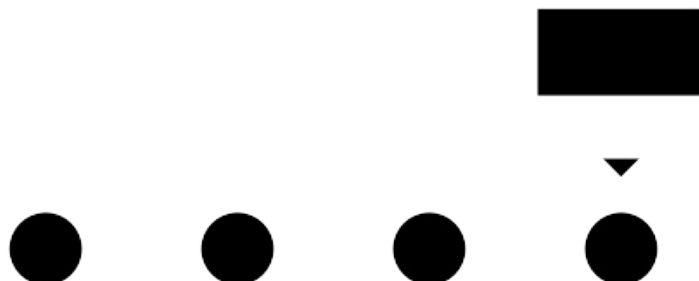
分离的 HEAD

现在我们已经看到了 `git checkout` 最主要的三种用法，我们可以讨论上一篇中提到的「分离 HEAD」状态了。

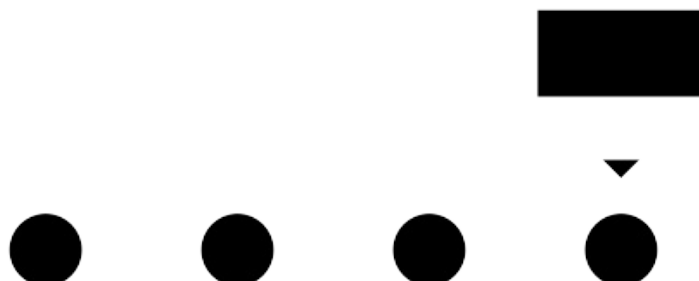
记住，HEAD 是 Git 指向当前快照的引用。`git checkout` 命令内部只是更新 HEAD，指向特定分支或提交。当它指向分支时，Git 不会报错，但当你 check out 提交时，它会进入「分离 HEAD」状态。

有个警告会告诉你所做的更改和项目的其余历史处于「分离」的状态。如果你在分离 `HEAD` 状态开始开发新功能，没有分支可以让你回到之前的状态。当你不可避免地 `checkout` 到了另一个分支（比如你的更改并入了这个分支），你将不再能够引用你的 `feature` 分支：

Attached HEAD



Detatched HEAD



.svg)

重点是，你应该永远在分支上开发——而绝不在分离的 `HEAD` 上。这样确保你一直可以引用到你的新提交。不过，如果你只是想查看旧的提交，那么是否处于分离 `HEAD` 状态并不重要。

例子

下面的例子演示了基本的 `Git` 分支流程。当你想要开发新功能时，你创建一个专门的分支，切换过去：

1. `git branch new-feature`
2. `git checkout new-feature`

接下来，你可以和以往一样提交新的快照：

1. `# 编辑文件`

```
2. git add <file>
3. git commit -m "Started work on a new feature"
4. # 周而复始...
```

这些操作都被记录在 `new-feature` 上，和 `master` 完全独立。你想添加多少提交就可以添加多少，不用关心你其它分支的修改。当你想要回到「主」代码库时，只要 `check out` 到 `master` 分支即可：

```
1. git checkout master
```

这个命令在你开始新的分支之前，告诉你仓库的状态。在这里，你可以选择并入完成的新功能，或者在你项目稳定的版本上继续工作。

git merge

合并是 Git 将被 fork 的历史放回到一起的方式。`git merge` 命令允许你将 `git branch` 创建的多条分支合并成一个。

注意，下面所有命令将更改 并入 当前分支。当前分支会被更新，以响应合并操作，但目标分支完全不受影响。也就是说 `git merge` 经常和 `git checkout` 一起使用，选择当前分支，然后用 `git branch -d` 删除废弃的目标分支。

用法

```
1. git merge <branch>
```

将指定分支并入当前分支。Git 会决定使用哪种合并算法（下文讨论）。

```
1. git merge --no-ff <branch>
```

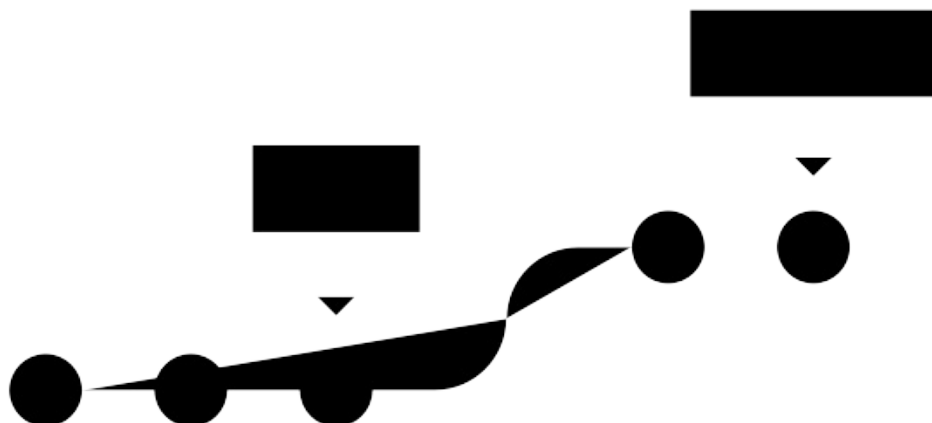
将指定分支并入当前分支，但 总是 生成一个合并提交（即使是快速向前合并）。这可以用来记录仓库中发生的所有合并。

讨论

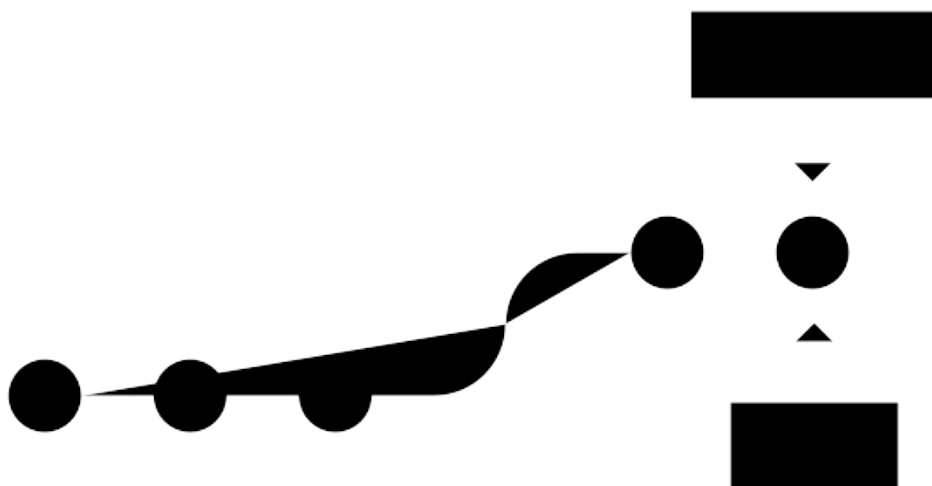
一旦你在单独的分支上完成了功能的开发，重要的是将它放回主代码库。取决于你的仓库结构，Git 有几种不同的算法来完成合并：快速向前合并或者三路合并。

当当前分支顶端到目标分支路径是线性之时，我们可以采取 快速向前合并 。Git 只需要将当前分支顶端（快速向前地）移动到目标分支顶端，即可整合两个分支的历史，而不需要“真正”合并分支。它在效果上合并了历史，因为目标分支上的提交现在在当前分支可以访问到。比如， `some-feature` 到 `master` 分支的快速向前合并会是这样的：

Before Merging



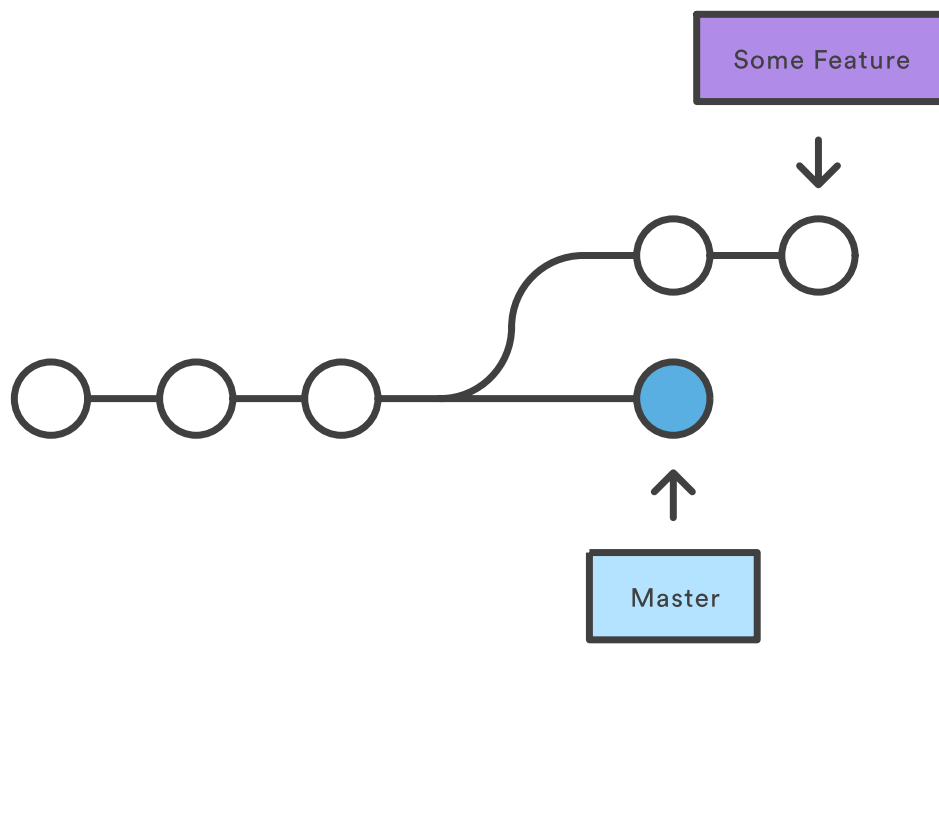
After a Fast-Forward Merge



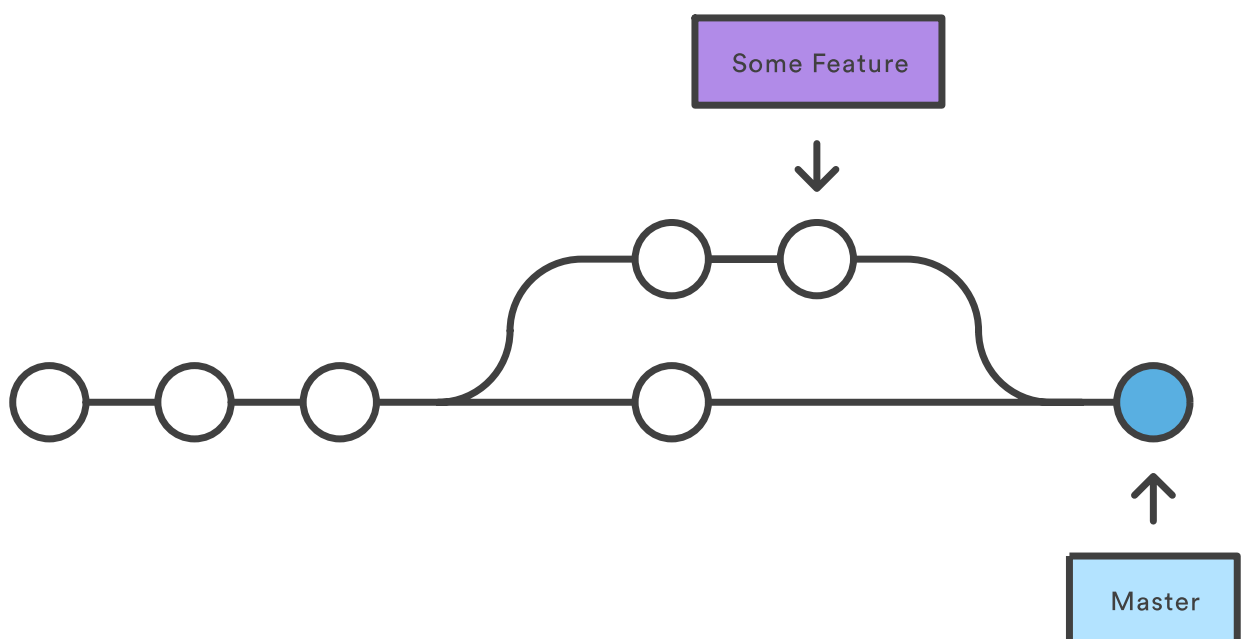
.svg)

但是，如果分支已经分叉了，那么就无法进行快速向前合并。当和目标分支之间的路径不是线性之时，Git 只能执行三路合并。三路合并使用一个专门的提交来合并两个分支的历史。这个术语取自这样一个事实，Git 使用三个提交来生成合并提交：两个分支顶端和它们共同的祖先。

Before Merging



After a 3-way Merge



但你可以选择使用哪一种合并策略时，很多开发者喜欢使用快速向前合并（搭配 rebase 使用）来合并微小的功能或

者修复 bug，使用三路合并来整合长期运行的功能。后者导致的合并提交作为两个分支的连接标志。

解决冲突

如果你尝试合并的两个分支同一个文件的同一个部分，Git 将无法决定使用哪个版本。当这种情况发生时，它会停在合并提交，让你手动解决这些冲突。

Git 的合并流程令人称赞的一点是，它使用我们熟悉的「编辑/缓存/提交」工作流来解决冲突。当你遇到合并冲突时，运行 `git status` 命令来查看哪些文件存在需要解决的冲突。比如，如果两个分支都修改了 `hello.py` 的同一处，你会看到下面的信息：

```
1. # On branch master
2. # Unmerged paths:
3. # (use "git add/rm ..." as appropriate to mark resolution)
4. #
5. # both modified: hello.py
6. #
```

接下来，你可以自己修复这个合并。当你准备结束合并时，你只需对冲突的文件运行 `git add` 告诉 Git 冲突已解决。然后，运行 `git commit` 生成一个合并提交。这和提交一个普通的快照有着完全相同的流程，也就是说，开发者能够轻而易举地管理他们的合并。

注意，提交冲突只会出现在三路合并中。在快速向前合并中，我们不可能出现冲突的更改。

例子

快速向前合并

我们第一个例子演示了快速向前合并。下面的代码创建了一个分支，在后面添加了两个提交，然后使用快速向前合并将它并入主分支。

```
1. # 开始新功能
2. git checkout -b new-feature master
3.
4. # 编辑文件
5. git add <file>
6. git commit -m "开始新功能"
7.
8. # 编辑文件
9. git add <file>
10. git commit -m "完成功能"
11.
12. # 合并new-feature分支
13. git checkout master
14. git merge new-feature
15. git branch -d new-feature
```

对于临时存在、用作独立开发环境而不是组织长期运行功能的工具的分支来说，这是一种常见的工作流。

同时注意，运行 `git branch -d` 时 Git 不应该产生错误提示，因为 `new-feature` 现在可以在主分支上访问了。

三路合并

下一个例子很相似，但需要进行三路合并，因为 `master` 在这个功能开发时取得了新进展。这是复杂功能和多个开发者同时工作时常见的情形。

```
1. # 开始新功能
2. git checkout -b new-feature master
3.
4. # 编辑文件
5. git add <file>
6. git commit -m "开始新功能"
7.
8. # 编辑文件
9. git add <file>
10. git commit -m "完成功能"
11.
12. # 在master分支上开发
13. git checkout master
14.
15. # 编辑文件
16. git add <file>
17. git commit -m "在master上添加了一些极其稳定的功能"
18.
19. # 合并new-feature分支
20. git merge new-feature
21. git branch -d new-feature
```

注意，Git 现在无法进行快速向前合并，因为无法将 `master` 直接移动到 `new-feature`。

对大多数 workflow 来说，`new-feature` 会是一个需要一段时间来开发的复杂功能，这也是为什么同时 `master` 会有新的提交出现。如果你的分支上的功能像上面的一样简单，你会更想将它 rebase 到 `master`，使用快速向前合并。它会通过整理项目历史来避免多余的合并提交。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

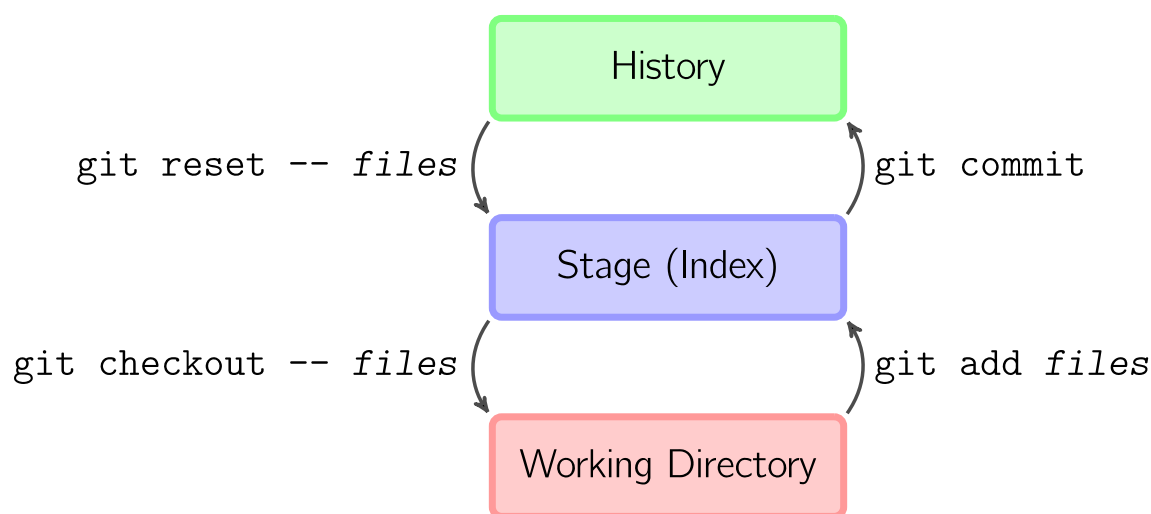
Git 图解

BY 童仲毅 (geeeeeeeeek@github)

这是一篇在[原文](#)基础上演绎的文章。原作者 [Mark Lodato](#)，译者 [wych](#)。原文采用[创用 CC 姓名标示-非商业性-相同方式分享 3.0 美国授权条款](#)授权。

此页图解 git 中的最常用命令。如果你稍微理解 git 的工作原理，这篇文章能够让你理解的更透彻。

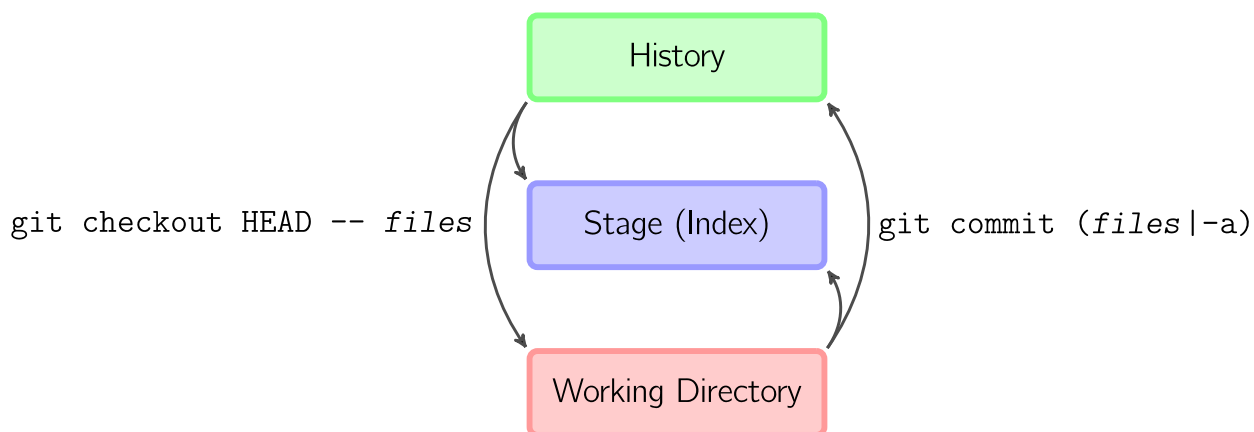
基本用法



上面的四条命令在工作目录、stage 缓存(也叫做索引)和 commit 历史之间复制文件。

- `git add files` 把工作目录中的文件加入 stage 缓存
- `git commit` 把 stage 缓存生成一次 commit，并加入 commit 历史
- `git reset -- files` 撤销最后一次 `git add files`，你也可以用 `git reset` 撤销所有 stage 缓存文件
- `git checkout -- files` 把文件从 stage 缓存复制到工作目录，用来丢弃本地修改

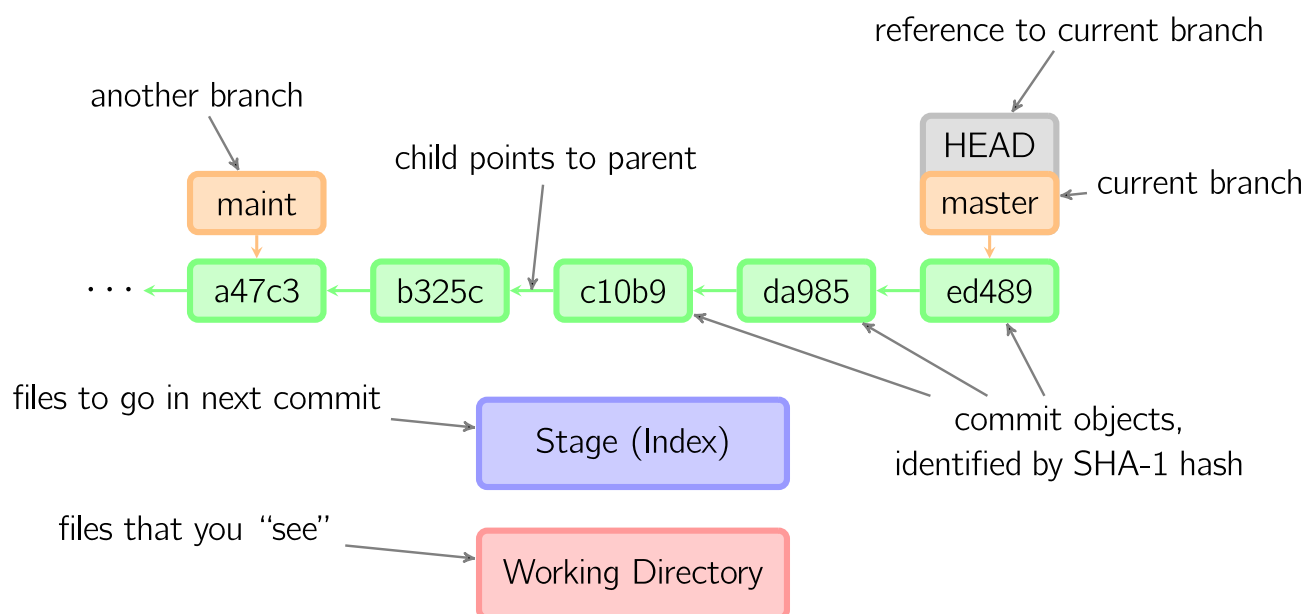
你可以用 `git reset -p`、`git checkout -p` 或 `git add -p` 进入交互模式，也可以跳过 stage 缓存直接从 commit 历史取出文件或者直接提交代码。



- `git commit -a` 相当于运行 `git add` 把所有当前目录下的文件加入 stage 缓存再运行 `git commit`。
- `git commit files` 进行一次包含最后一次提交加上工作目录中文件快照的提交，并且文件被添加到 stage 缓存。
- `git checkout HEAD -- files` 回滚到复制最后一次提交。

约定

后文中以下面的形式使用图片：



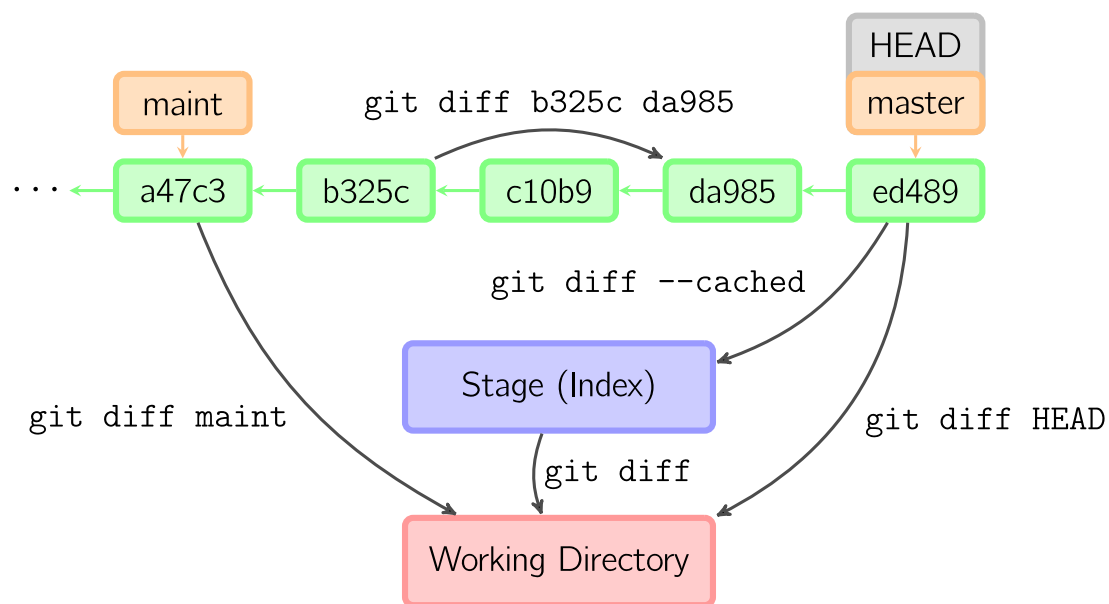
绿色的5位字符表示提交的 ID，分别指向父节点。分支用橙色显示，分别指向特定的提交。当前分支由附在其上的 `_HEAD_` 标识。

这张图片里显示最后 5 次提交，`_ed489_` 是最新提交。`_master_` 分支指向此次提交，另一个 `_maint_` 分支指向祖父提交节点。

命令详解

Diff

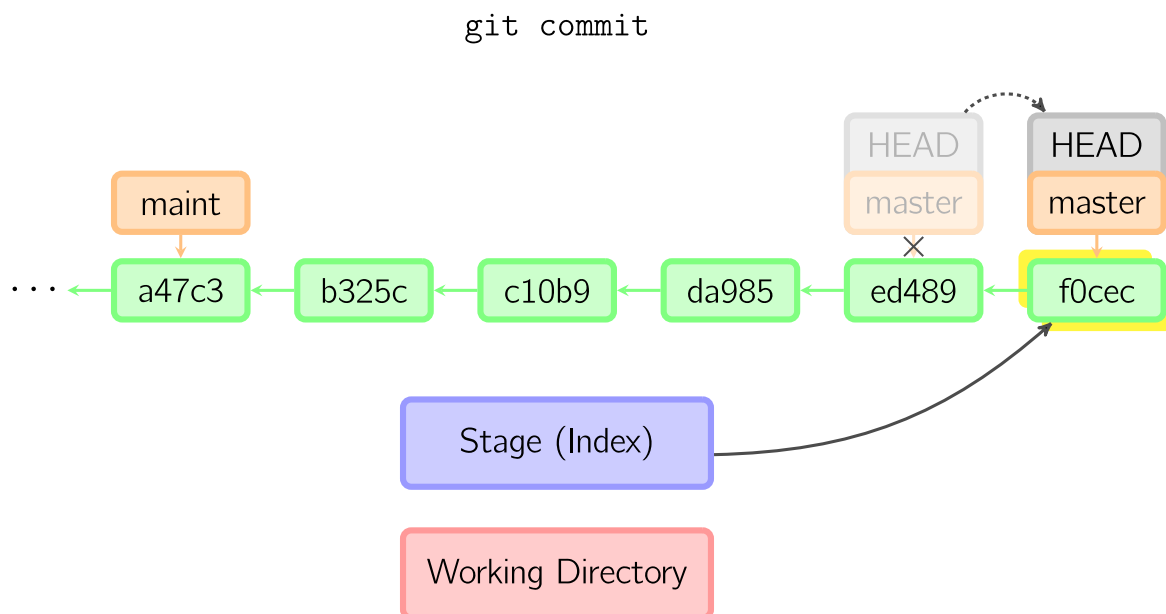
有许多种方法查看两次提交之间的变动，下面是其中一些例子。



Commit

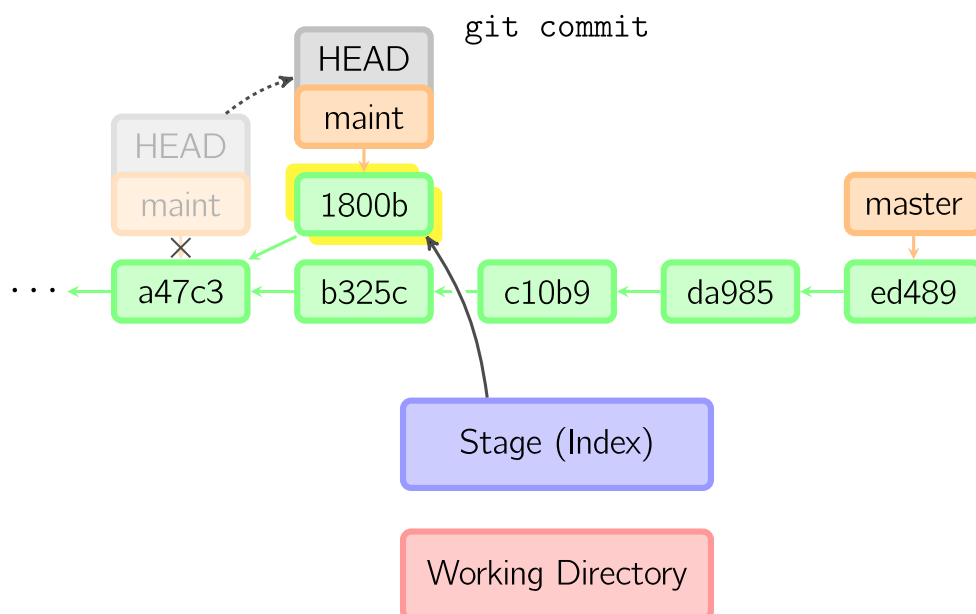
提交时，Git 用 stage 缓存中的文件创建一个新的提交，并把此时的节点设为父节点。然后把当前分支指向新的提交节点。下图中，当前分支是 `_master_`。

在运行命令之前，`_master_` 指向 `_ed489_`，提交后，`_master_` 指向新的节点 `_f0cec_` 并以 `_ed489_` 作为父节点。



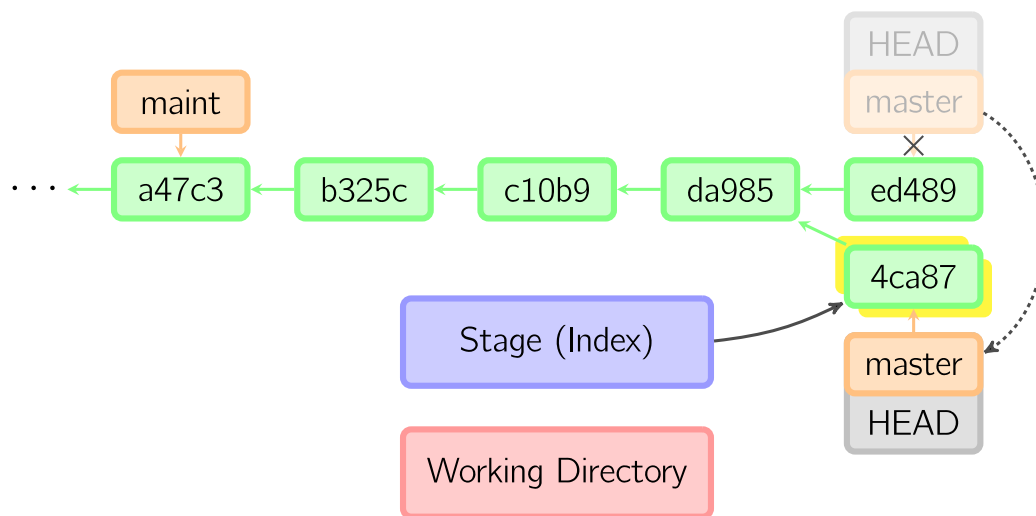
即便当前分支是某次提交的祖父节点，Git 会同样操作。下图中，在 `_master_` 分支的祖父节点 `_maint_` 分支进行一次提交，生成了 `_1800b_`。

这样，`_maint_` 分支就不再是 `_master_` 分支的祖父节点。此时，`merge` 或者 `rebase` 是必须的。



如果想更改一次提交，使用 `git commit --amend`。Git 会使用与当前提交相同的父节点进行一次新提交，旧的提交会被取消。


```
git commit --amend
```



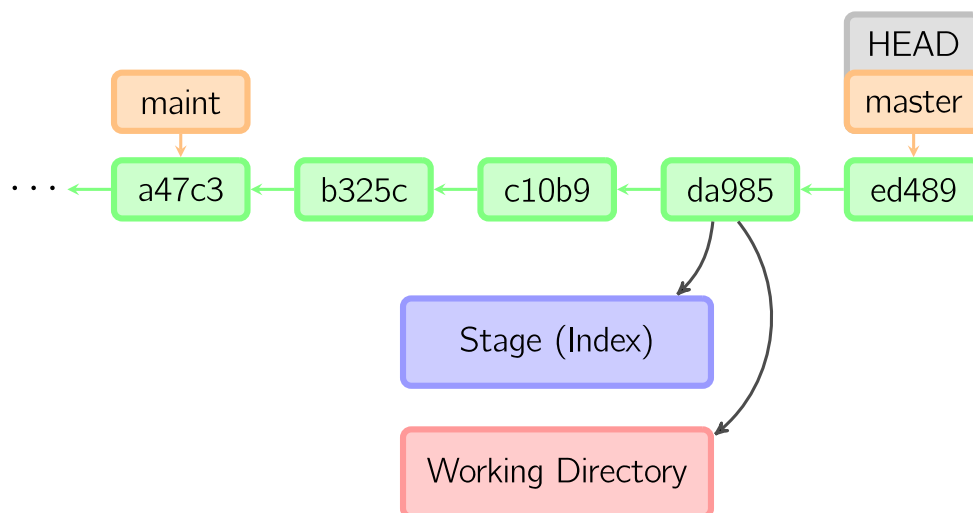
另一个例子是[分离HEAD提交](#)，在后面的章节中介绍。

Checkout

`git checkout` 命令用于从历史提交（或者 stage 缓存）中拷贝文件到工作目录，也可用于切换分支。

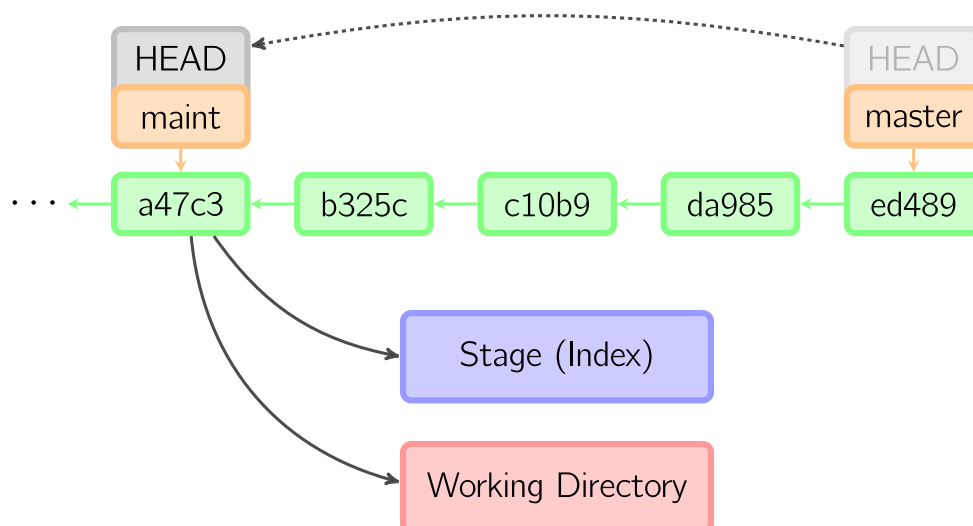
当给定某个文件名（或者打开 `-p` 选项，或者文件名和 `-p` 选项同时打开）时，Git 会从指定的提交中拷贝文件到 stage 缓存和工作目录。比如，`git checkout HEAD~ foo.c` 会将提交节点 `_HEAD~_`（即当前提交节点的父节点）中的 `foo.c` 复制到工作目录并且加到 stage 缓存中。如果命令中没有指定提交节点，则会从 stage 缓存中拷贝内容。注意当前分支不会发生变化。

```
git checkout HEAD~ files
```



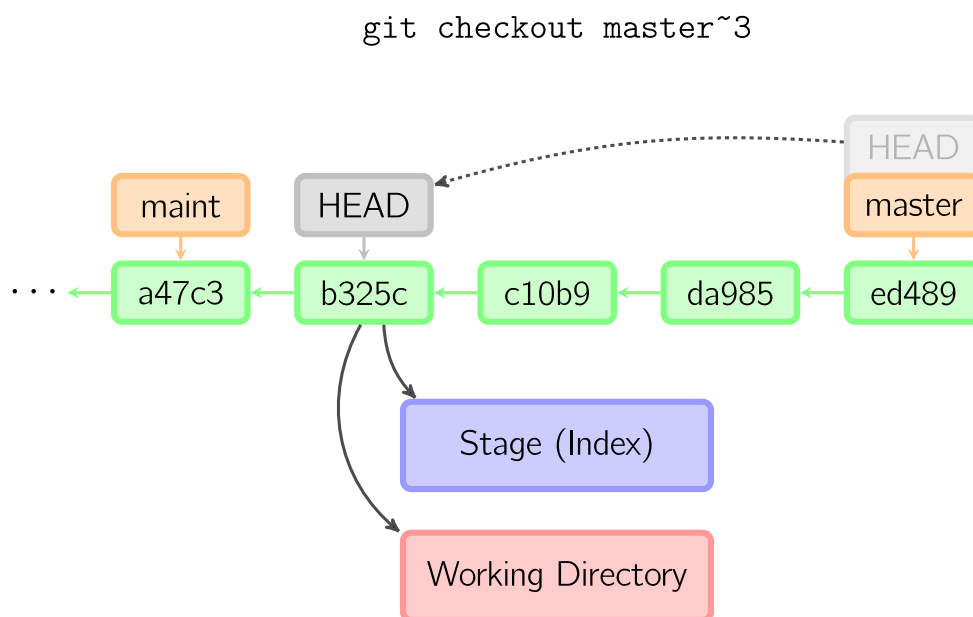
当不指定文件名，而是给出一个（本地）分支时，那么 `_HEAD_` 标识会移动到那个分支（也就是说，我们「切换」到那个分支了），然后 stage 缓存和工作目录中的内容会和 `_HEAD_` 对应的提交节点一致。新提交节点（下图中的 `a47c3`）中的所有文件都会被复制（到 stage 缓存和工作目录中）；只存在于老的提交节点（`ed489`）中的文件会被删除；不属于上述两者的文件会被忽略，不受影响。

```
git checkout maint
```



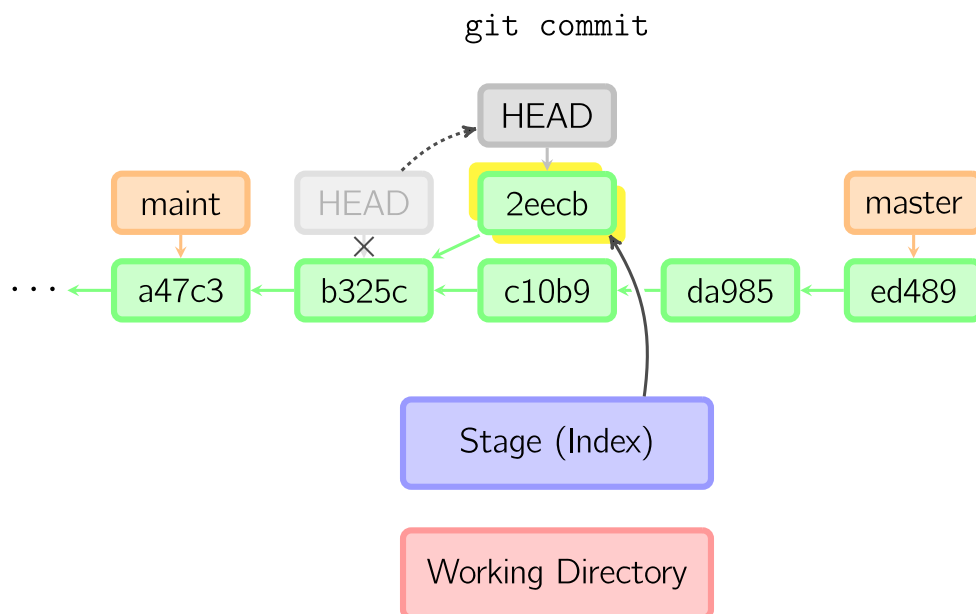
如果既没有指定文件名，也没有指定分支名，而是一个标签、远程分支、SHA-1 值或者是像 `_master~3_` 类似的东西，就得到一个匿名分支，称作 `_detached HEAD_`（被分离的 `_HEAD_` 标识）。这样可以很方便地在历史版本之

间互相切换。比如说你想要编译 1.6.6.1 版本的 Git，你可以运行 `git checkout v1.6.6.1`（这是一个标签，而非分支名），编译，安装，然后切换回另一个分支，比如说 `git checkout master`。然而，当提交操作涉及到「分离的 HEAD」时，其行为会略有不同，详情见在[下面](#)。

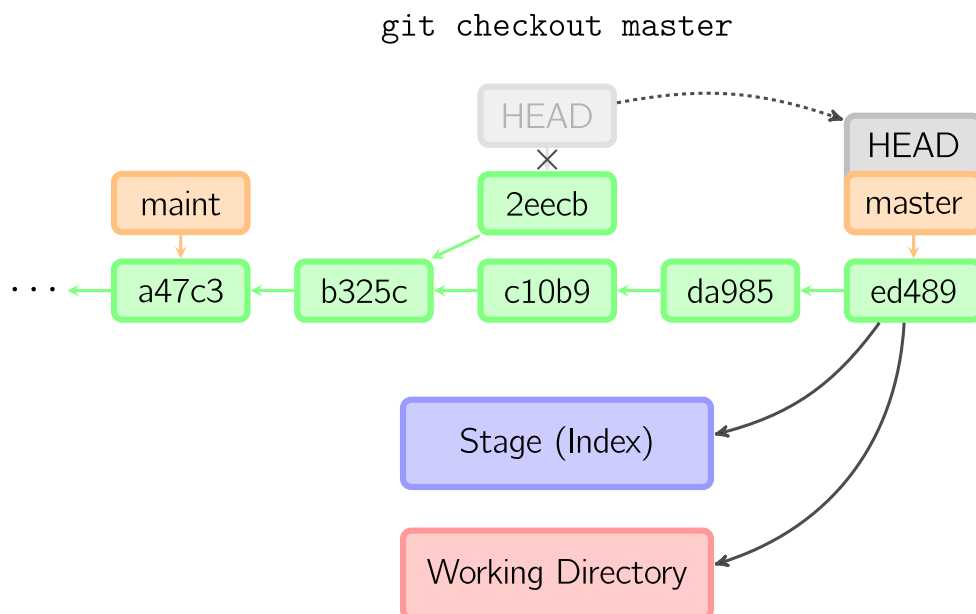


HEAD 标识处于分离状态时的提交操作

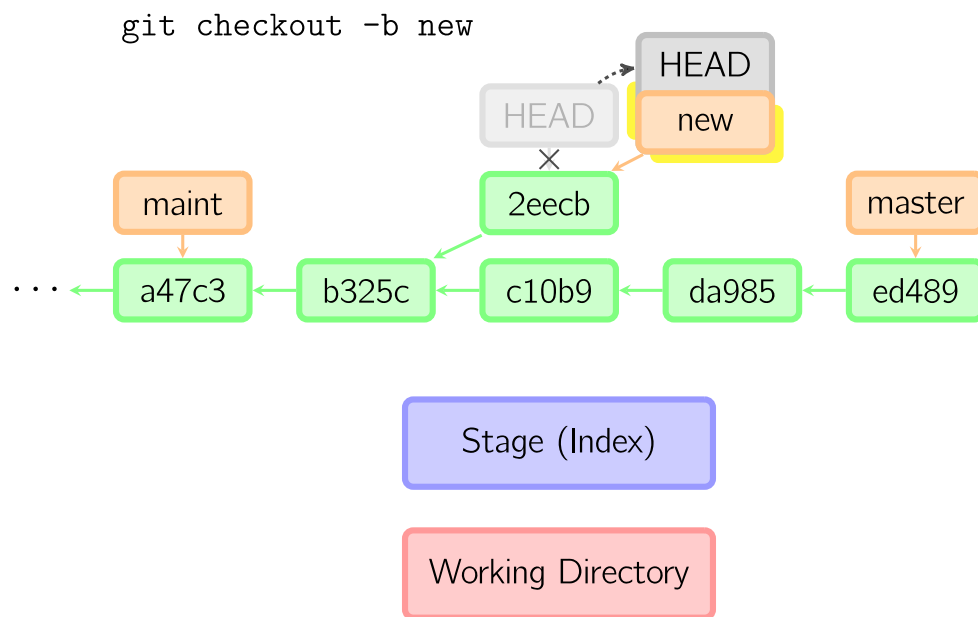
当 `_HEAD_` 处于分离状态（不依附于任一分支）时，提交操作可以正常进行，但是不会更新任何已命名的分支。你可以认为这是在更新一个匿名分支。



一旦此后你切换到别的分支，比如说 `_master_`，那么这个提交节点（可能）再也不会被引用到，然后就会被丢弃掉了。注意这个命令之后就不会有东西引用 `_2eecb_`。



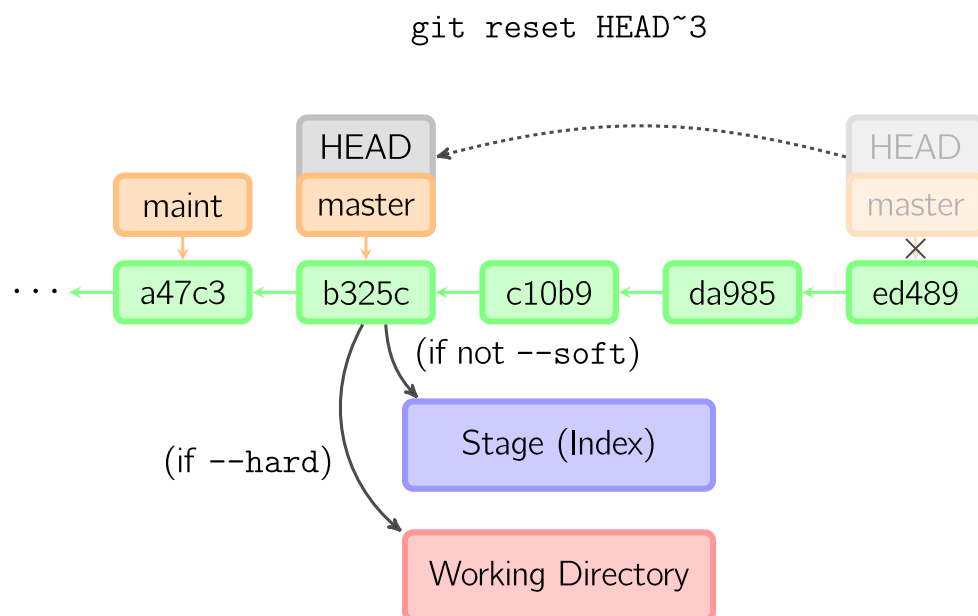
但是，如果你想保存这个状态，可以用命令 `git checkout -b name` 来创建一个新的分支。



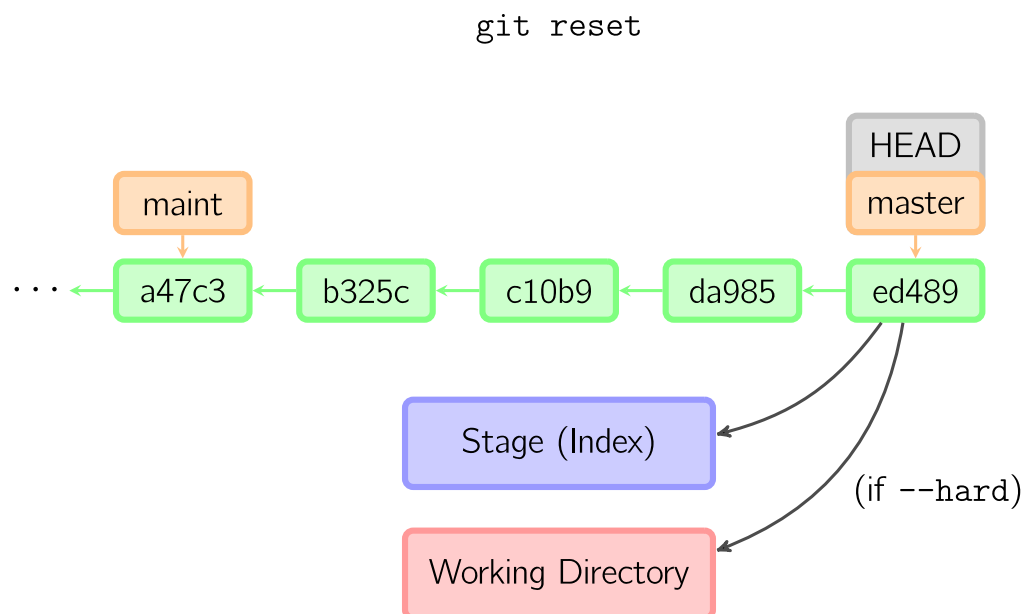
Reset

`git reset` 命令把当前分支指向另一个位置，并且有选择的变动工作目录和索引。也用来在从历史commit历史中复制文件到索引，而不动工作目录。

如果不给选项，那么当前分支指向到那个提交。如果用 `--hard` 选项，那么工作目录也更新，如果用 `--soft` 选项，那么都不变。

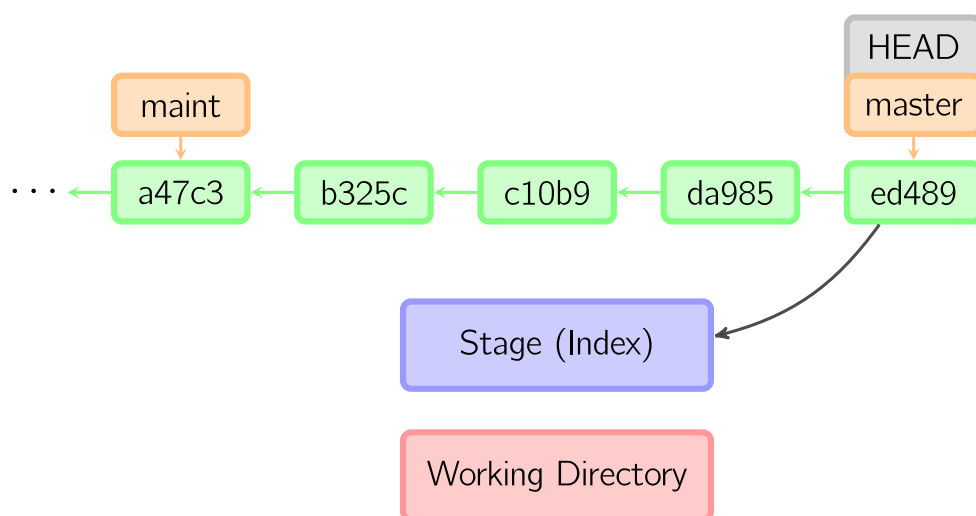


如果没有给出提交点的版本号, 那么默认用 `_HEAD_`。这样, 分支指向不变, 但是索引会回滚到最后一次提交, 如果用 `--hard` 选项, 工作目录也同様。



如果给了文件名(或者 `-p` 选项), 那么工作效果和带文件名的 `checkout` 差不多, 除了索引被更新。

```
git reset -- files
```

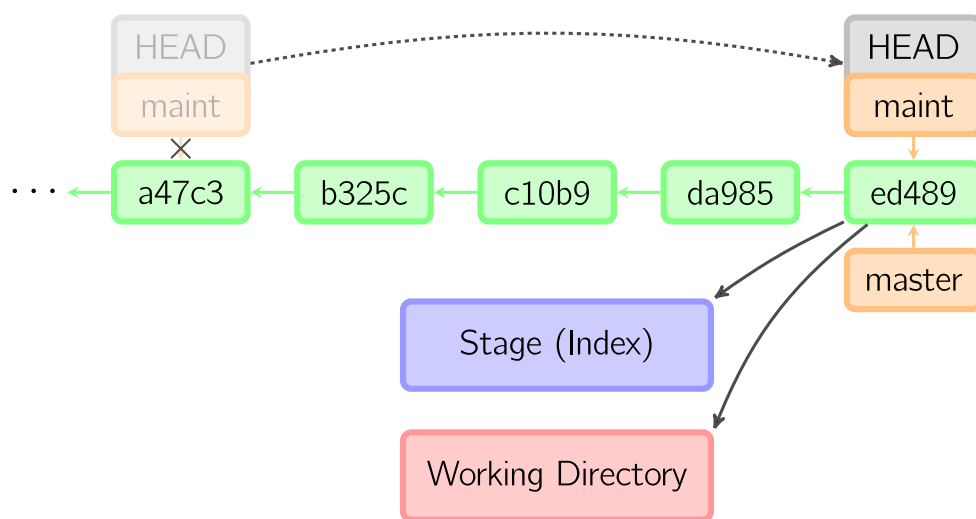


Merge

`git merge` 命令把不同分支合并起来。合并前，索引必须和当前提交相同。如果另一个分支是当前提交的祖父节点，那么合并命令将什么也不做。

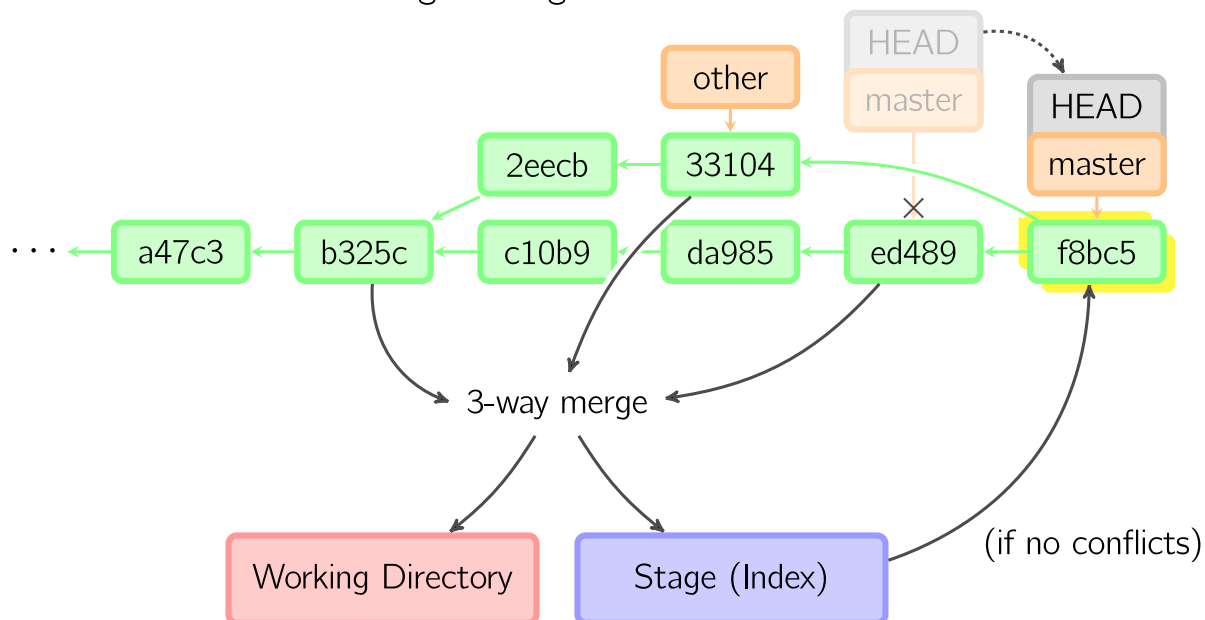
另一种情况是如果当前提交是另一个分支的祖父节点，就导致 `_fast-forward_` 合并。指向只是简单的移动，并生成一个新的提交。

```
git merge master
```



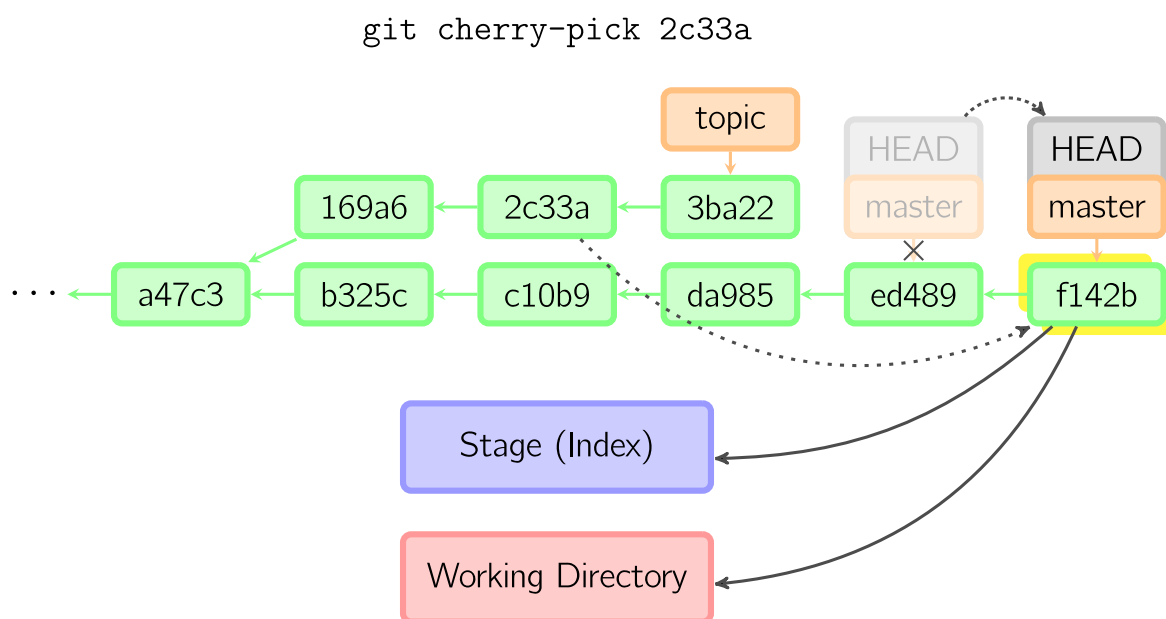
否则就是一次真正的合并。默认把当前提交 (`_ed489_` 如下所示) 和另一个提交 (`_33104_`) 以及他们的共同祖父节点 (`_b325c_`) 进行一次三方合并。结果是先保存当前目录和索引，然后和父节点 `_33104_` 一起做一次新提交。

```
git merge other
```



Cherry Pick

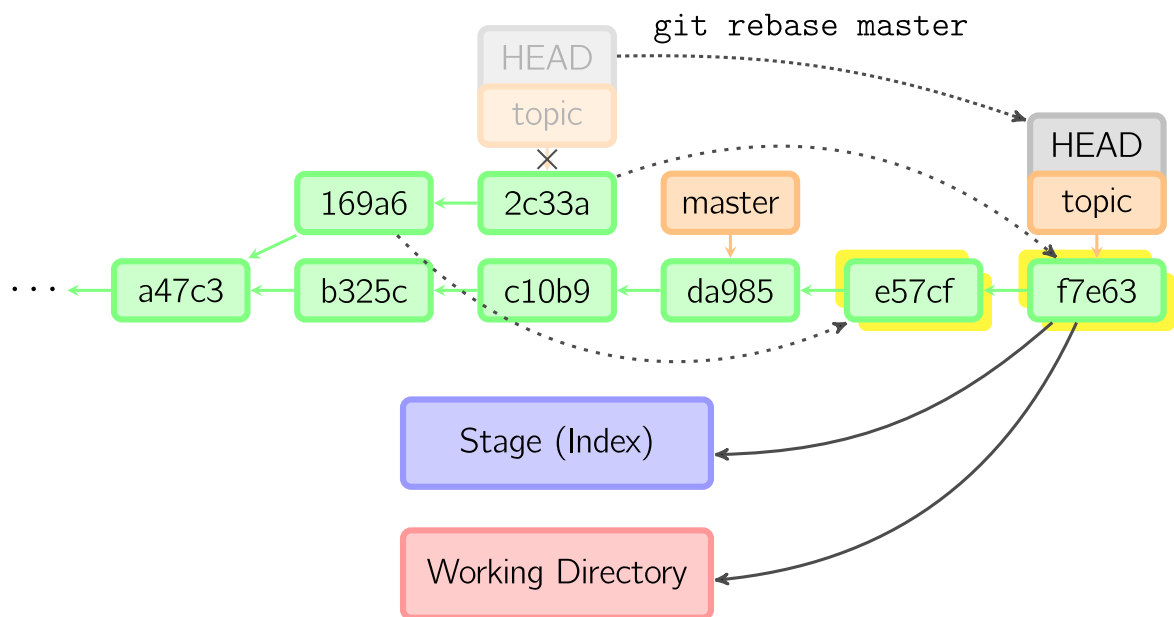
`git cherry-pick` 命令「复制」一个提交节点并在当前分支做一次完全一样的新提交。



Rebase

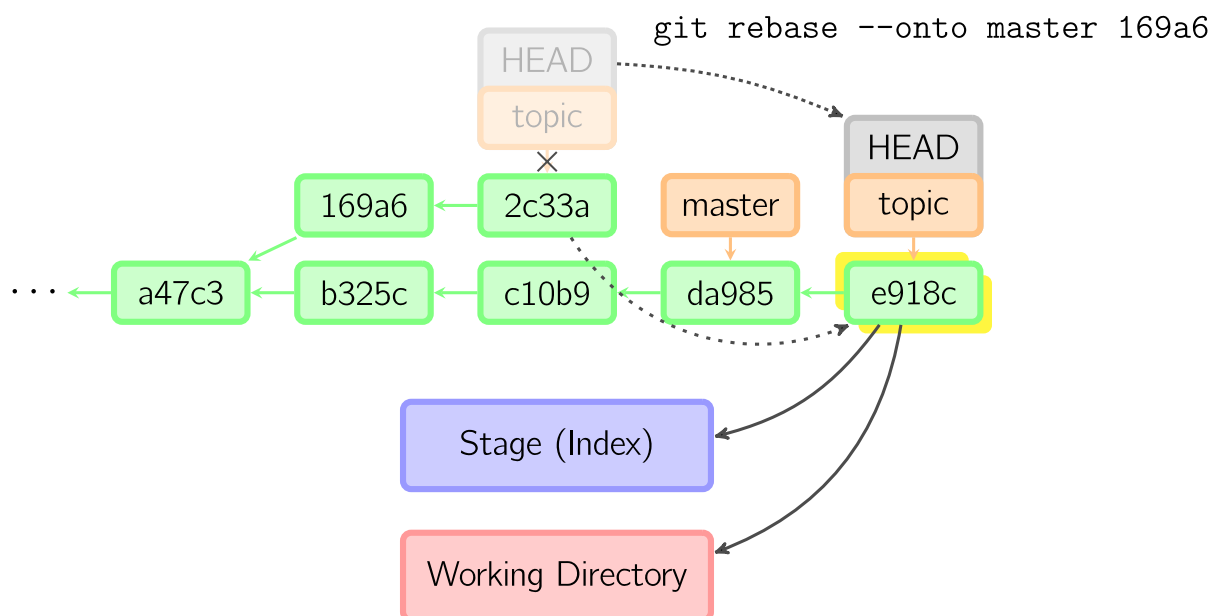
`git rebase` 是合并命令的另一种选择。合并把两个父分支合并进行一次提交，提交历史不是线性的。`rebase` 在当前分支上重演另一个分支的历史，提交历史是线性的。

本质上，这是线性化的自动的 `cherry-pick`。



上面的命令都在 `_topic_` 分支中进行，而不是 `_master_` 分支，在 `_master_` 分支上重演，并且把分支指向新的节点。注意旧提交没有被引用，将被回收。

要限制回滚范围，使用 `--onto` 选项。下面的命令在 `_master_` 分支上重演当前分支从 `_169a6_` 以来的最近几个提交，即 `_2c33a_`。



同样有 `git rebase --interactive` 让你更方便的完成一些复杂操作，比如丢弃、重排、修改、合并提交。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 **Star** :star2: 或 **Fork** :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

代码合并：Merge、Rebase 的选择

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。

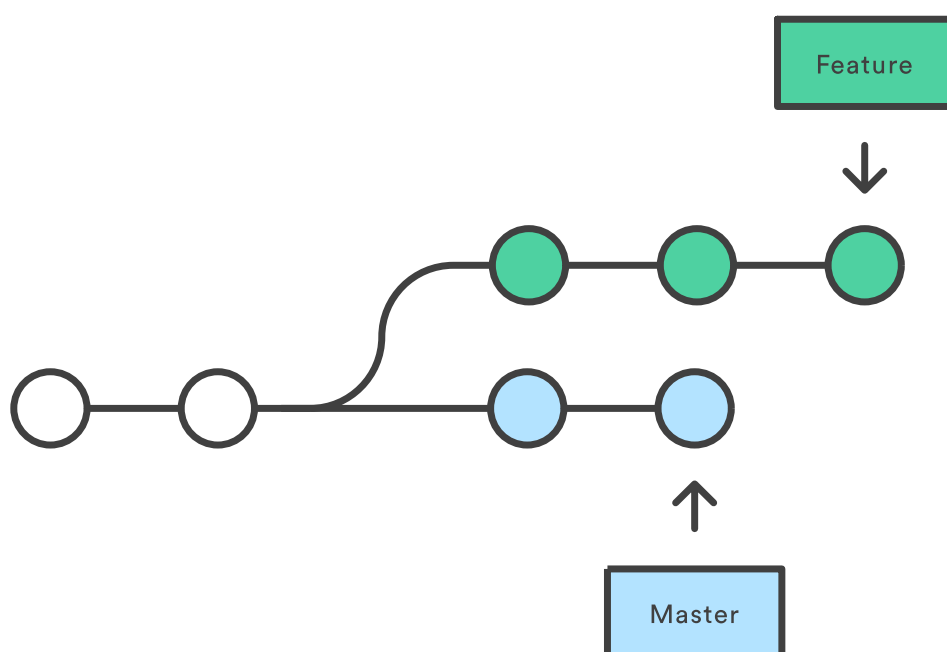
`git rebase` 这个命令经常被人认为是一种 Git 巫术，初学者应该避而远之。但如果使用得当的话，它能给你的团队开发省去太多烦恼。在这篇文章中，我们会比较 `git rebase` 和类似的 `git merge` 命令，找到 Git 工作流程中 rebase 的所有用法。

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：

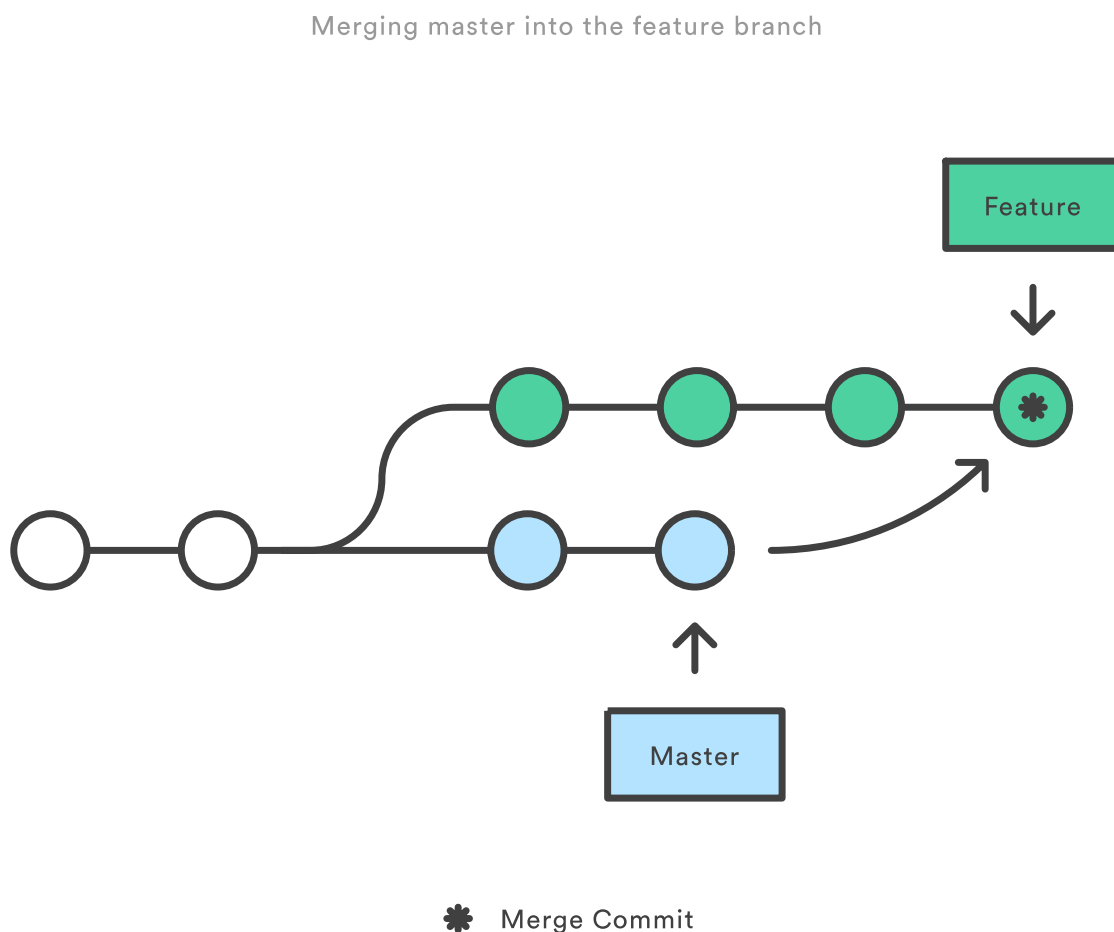
```
1. git checkout feature
```

```
2. git merge master
```

或者，你也可以把它们压缩在一行里。

```
1. 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：



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。

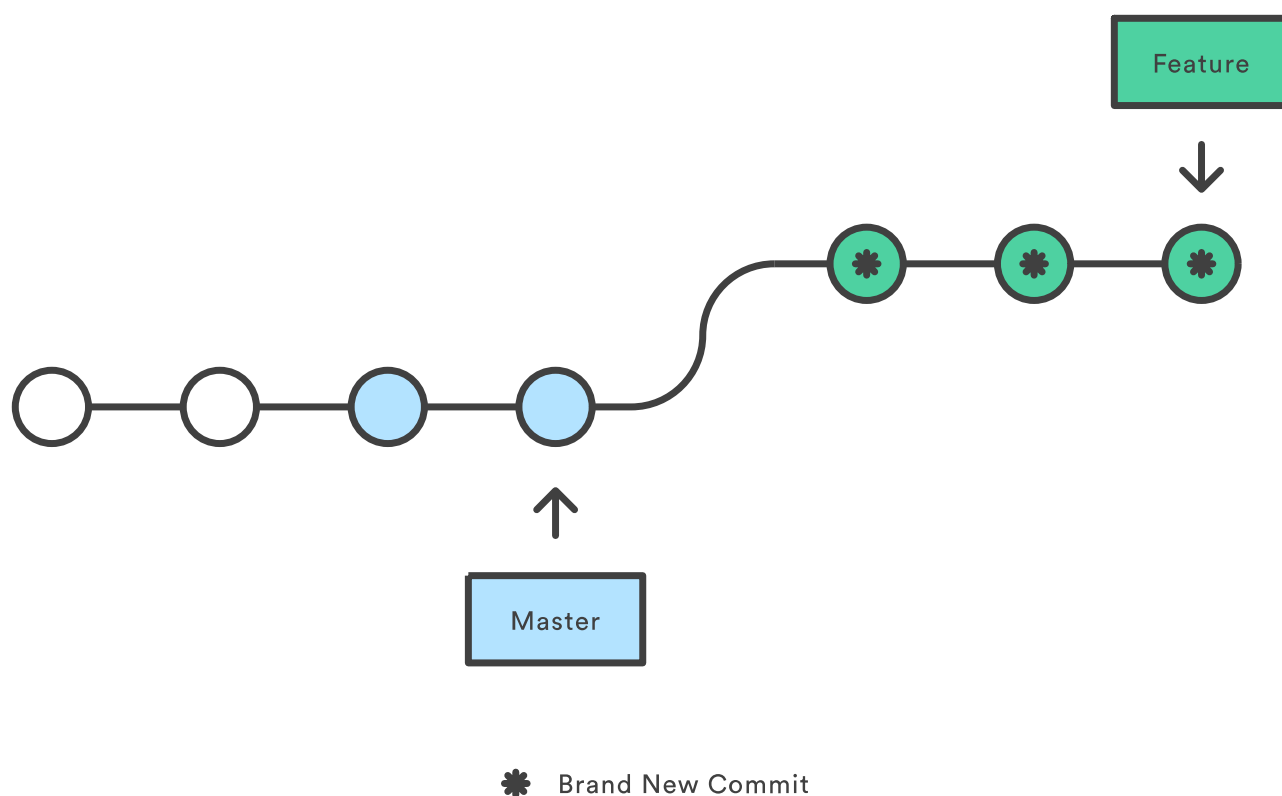
Rebase

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：

```
1. git checkout feature
2. 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 workflow 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：

```
1. git checkout feature
2. 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：

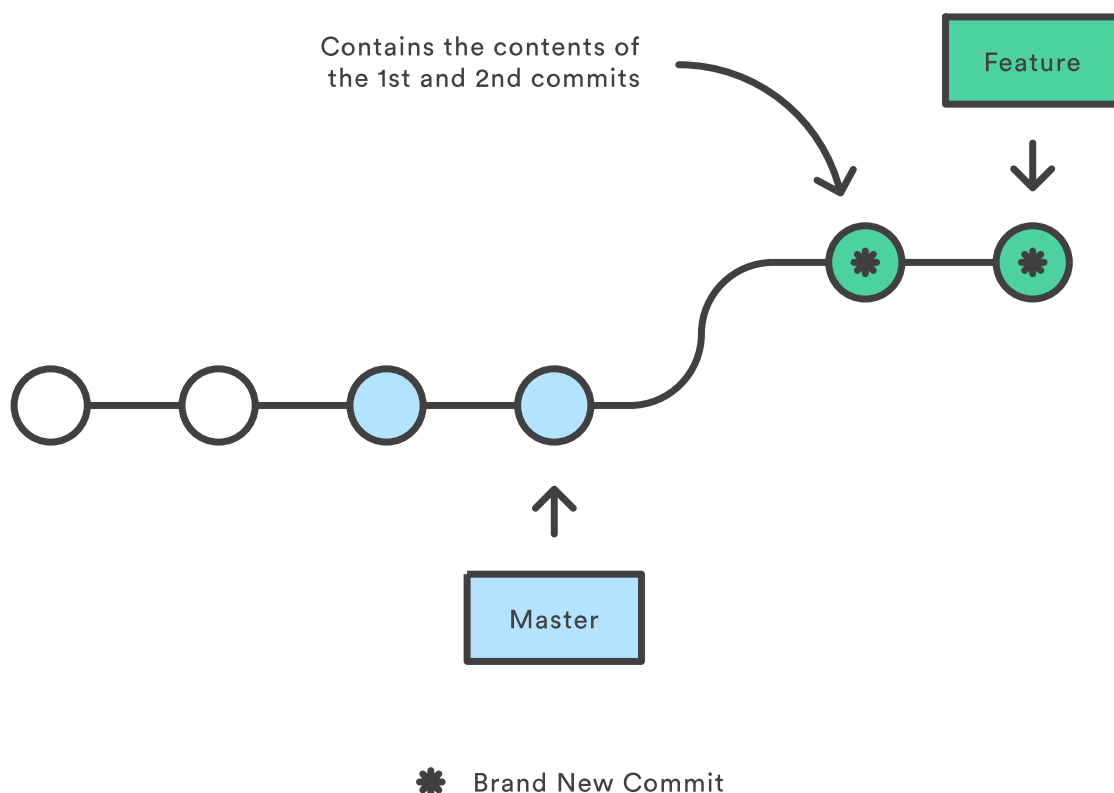
```
1. pick 33d5b7a Message for commit #1
2. pick 9480b3d Message for commit #2
3. pick 5c67e61 Message for commit #3
```

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：

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



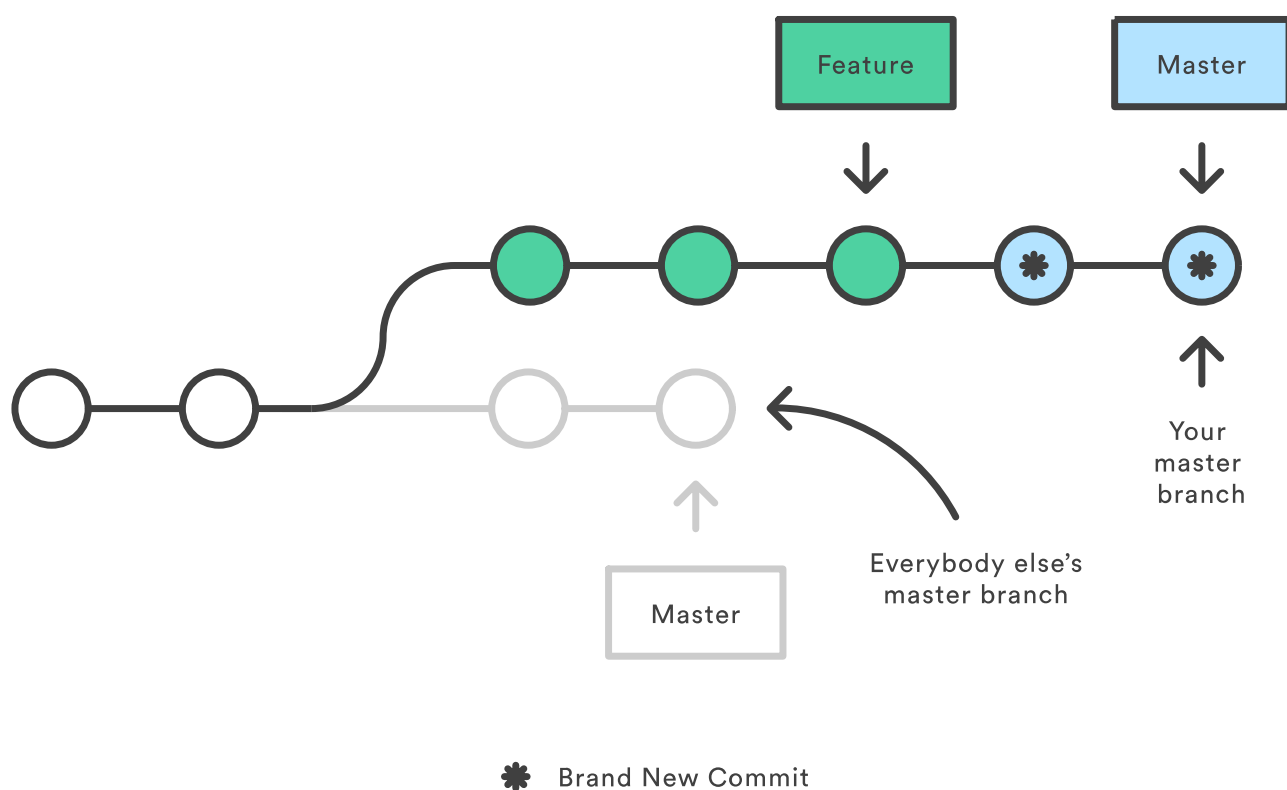
忽略不重要的提交会让你的 feature 分支的历史更清晰易读。这是 `git merge` 做不到的。

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



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：

1. # 小心使用这个命令！
2. `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，其实我并不要推送之前那个 feature 分支的。用我现在的版本替换掉吧。」

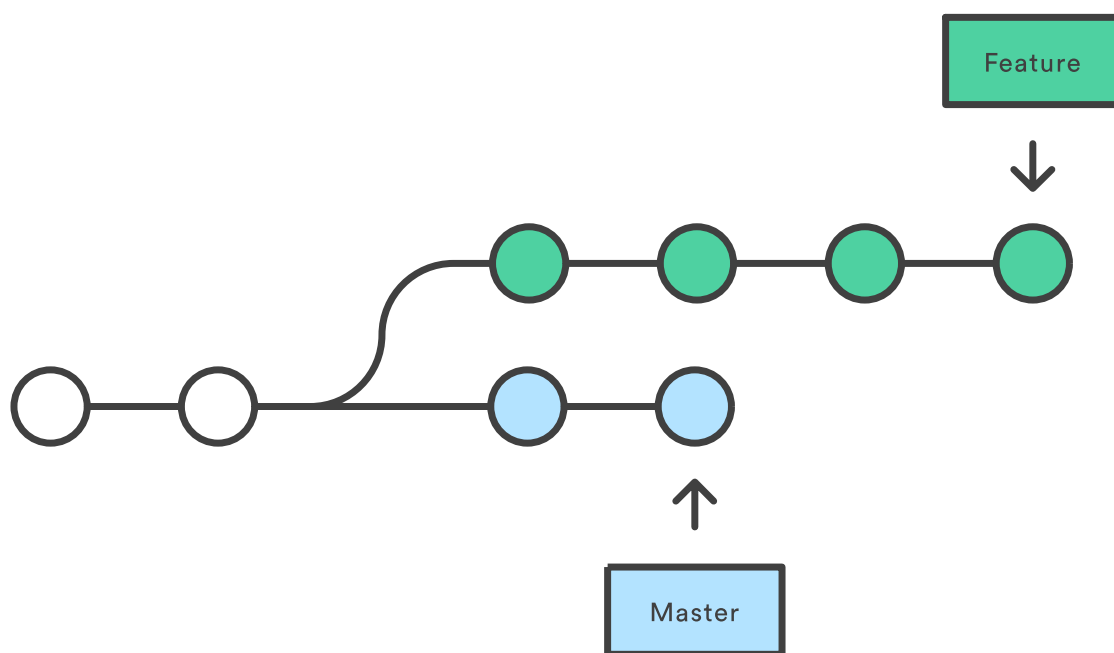
同样，你要注意没有别人正在这个 feature 分支上工作。

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

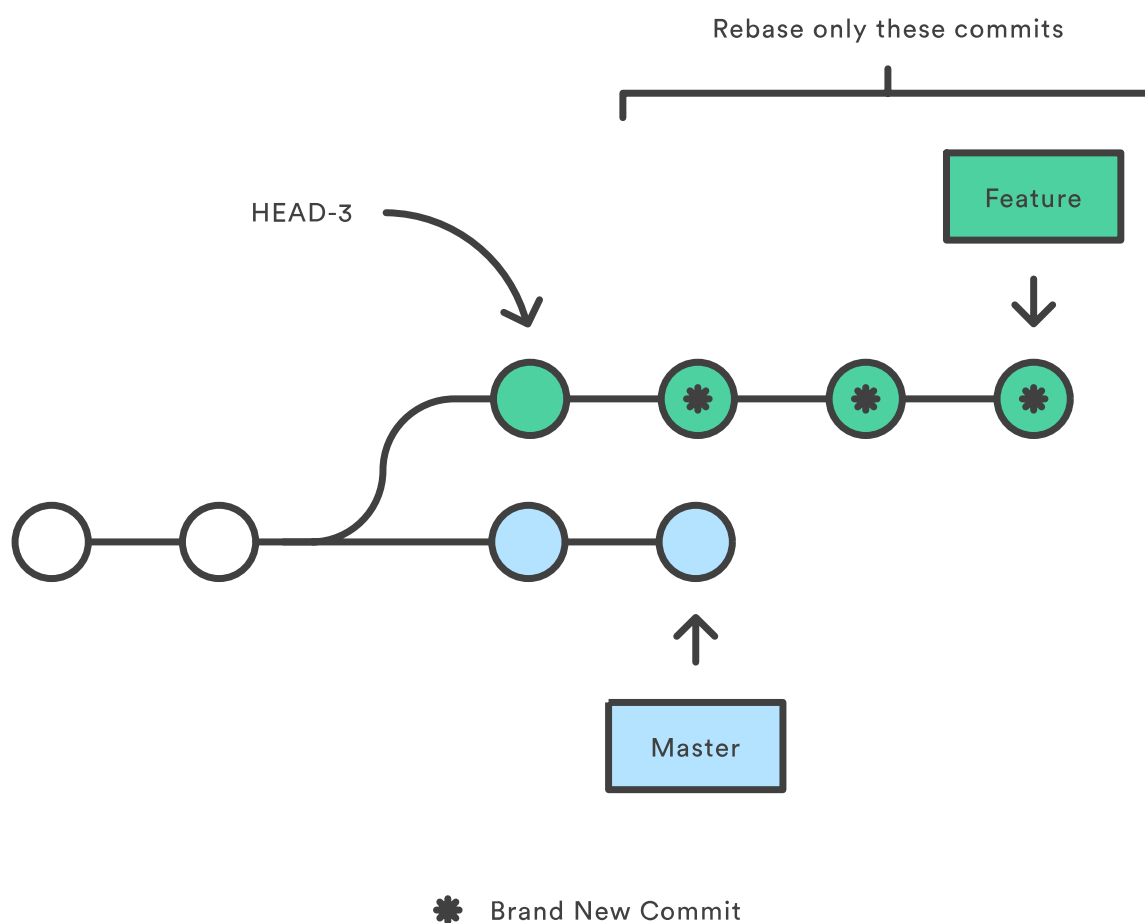
在你工作流中使用 rebase 最好的用法之一就是清理本地正在开发的分支。隔一段时间执行一次交互式 rebase，你可以保证你 feature 分支中的每一个提交都是专注和有意义的。你在写代码时不用担心造成孤立的提交—因为你后面一定能修复。

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：

```
1. git checkout feature
2. 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`：

```
1. git merge-base feature master
```

交互式 rebase 是在你工作流中引入 `git rebase` 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

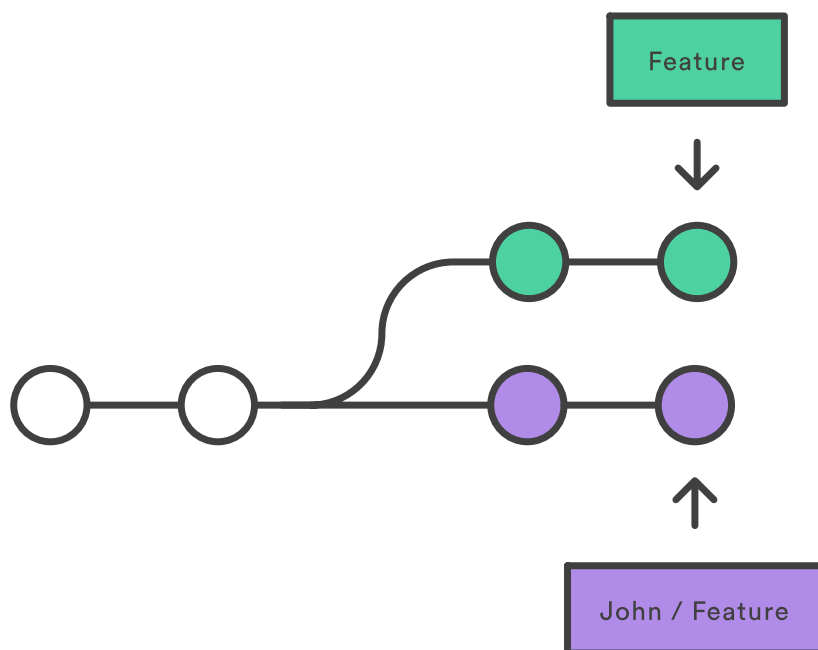
在概览一节，我们看到了 feature 分支如何通过 `git merge` 或 `git rebase` 来并入上游分支。merge 是保留你完整历史的安全选择，rebase 将你的 feature 分支移动到 master 分支后面，创建一个线性的历史。

`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 分支之上协作的时候，你会用到这个用法，将他们的更改并入你的项目。

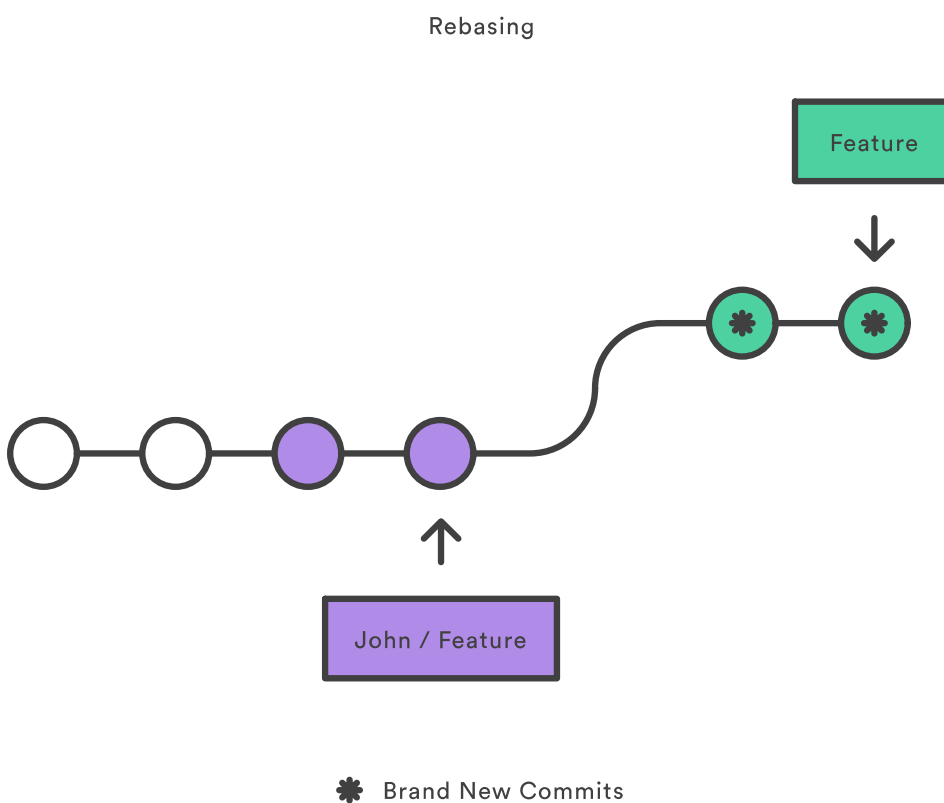
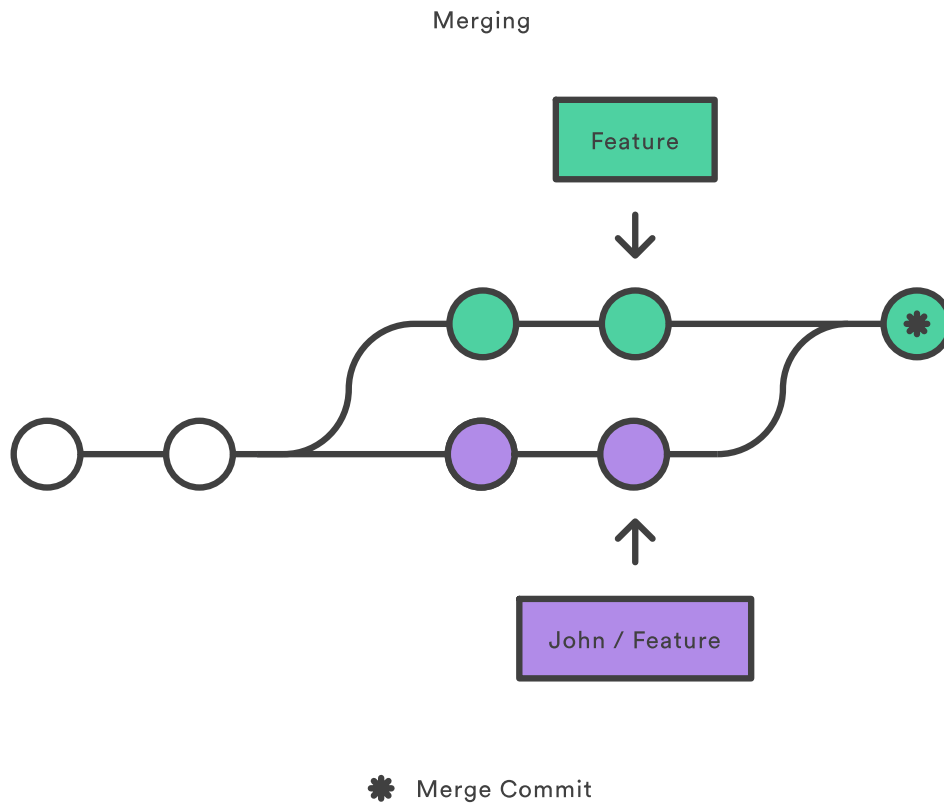
比如说，如果你和另一个开发者 John 往 feature 分支上添加了几个提交，在从 John 的仓库中 fetch 之后，你的仓库可能会像下面这样：

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，要么把你的本地分支 rebase 到 John 的分支后面。

Merging vs. rebasing onto a remote branch



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。

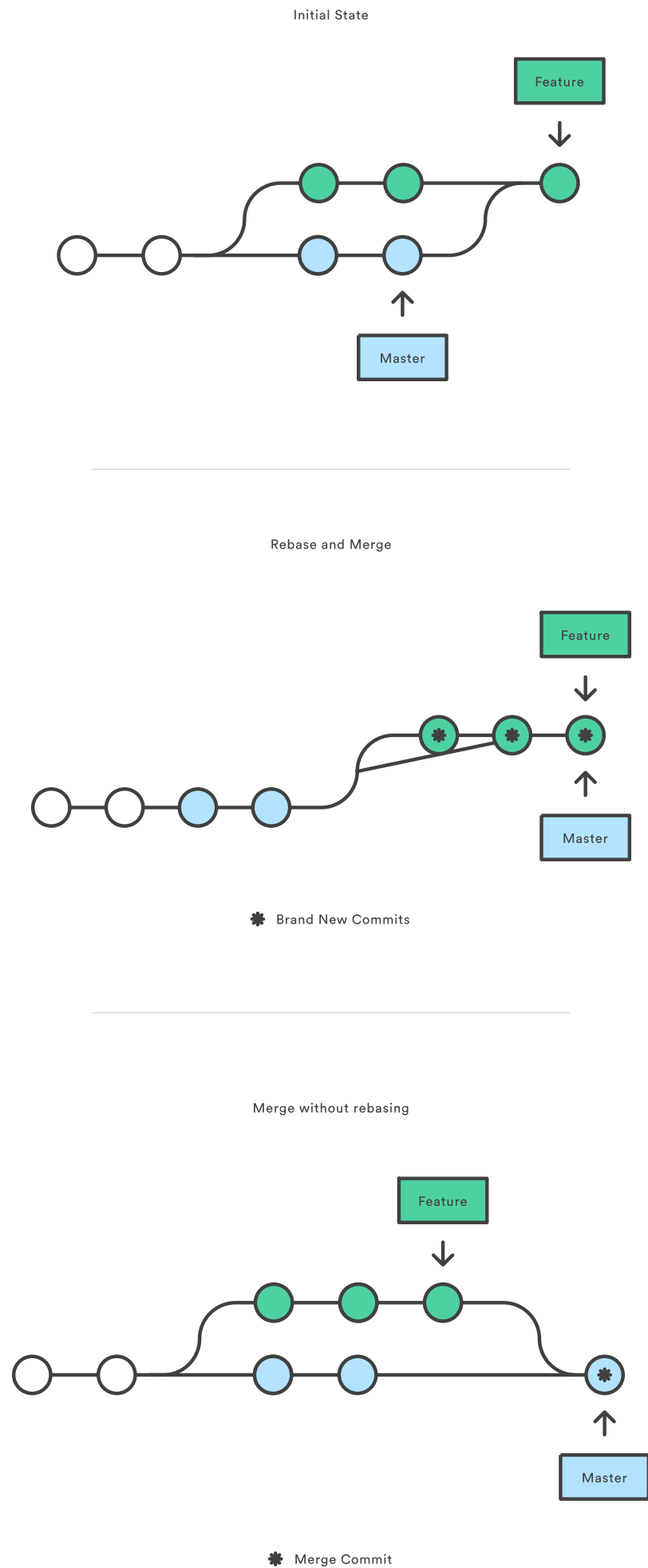
因此，在提交 Pull Request前用交互式的 rebase 进行代码清理通常是一个好的做法。

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



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：

```
1. git checkout feature
2. git checkout -b temporary-branch
3. git rebase -i master
4. # [清理目录]
5. git checkout master
6. 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。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

代码回滚：Reset、Checkout、Revert 的选择

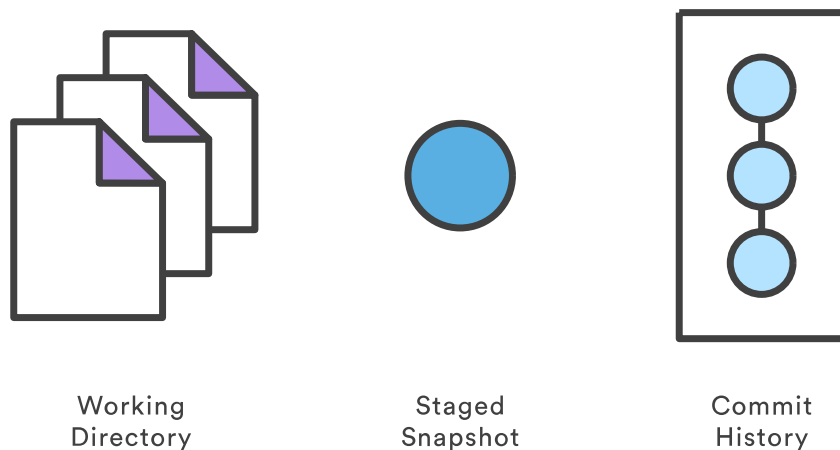
BY 童仲毅 (geeeeeeeeek@github)

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 reset`、`git checkout` 和 `git revert` 是你的 Git 工具箱中最有用的一些命令。它们都用来撤销代码仓库中的某些更改，而前两个命令不仅可以作用于提交，还可以作用于特定文件。

因为它们非常相似，所以我们经常会搞混，不知道什么场景下该用哪个命令。在这篇文章中，我们会比较 `git reset`、`git checkout` 和 `git revert` 最常见的用法。希望你在看完后能游刃有余地使用这些命令来管理你的仓库。

The main components of a Git repository



Git 仓库有三个主要组成——工作目录，缓存区和提交历史。这张图有助于理解每个命令到底产生了哪些影响。当你阅读的时候，牢记这张图。

提交层面的操作

你传给 `git reset` 和 `git checkout` 的参数决定了它们的作用域。如果你没有包含文件路径，这些操作对所有提交生效。我们这一节要探讨的就是提交层面的操作。注意，`git revert` 没有文件层面的操作。

Reset

在提交层面上，`reset` 将一个分支的末端指向另一个提交。这可以用来移除当前分支的一些提交。比如，下面这两条命令让 `hotfix` 分支向后回退了两个提交。

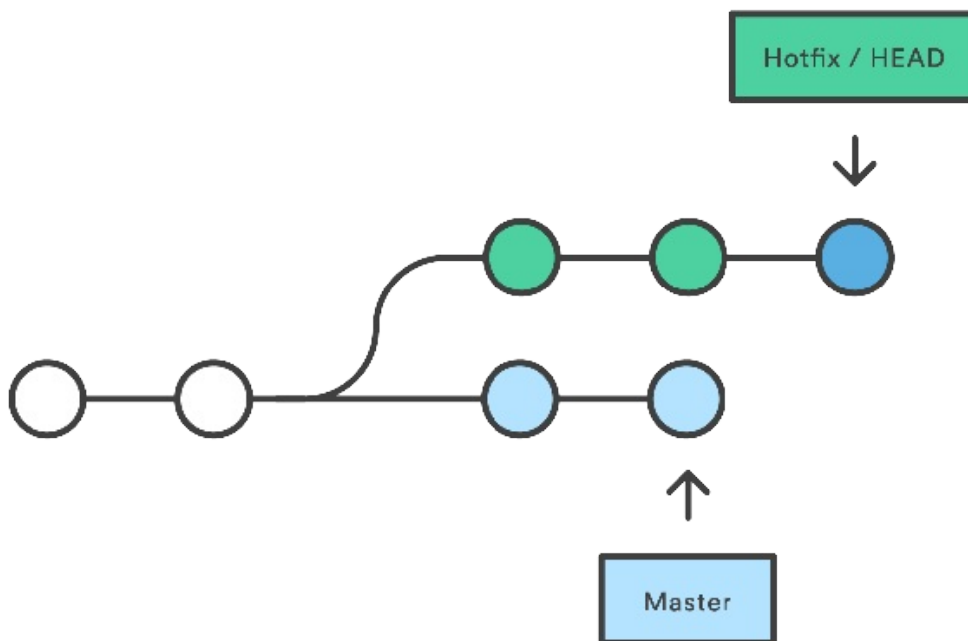
1. `git checkout hotfix`
2. `git reset HEAD~2`

`hotfix` 分支末端的两个提交现在变成了悬挂提交。也就是说，下次 Git 执行垃圾回收的时候，这两个提交会被删

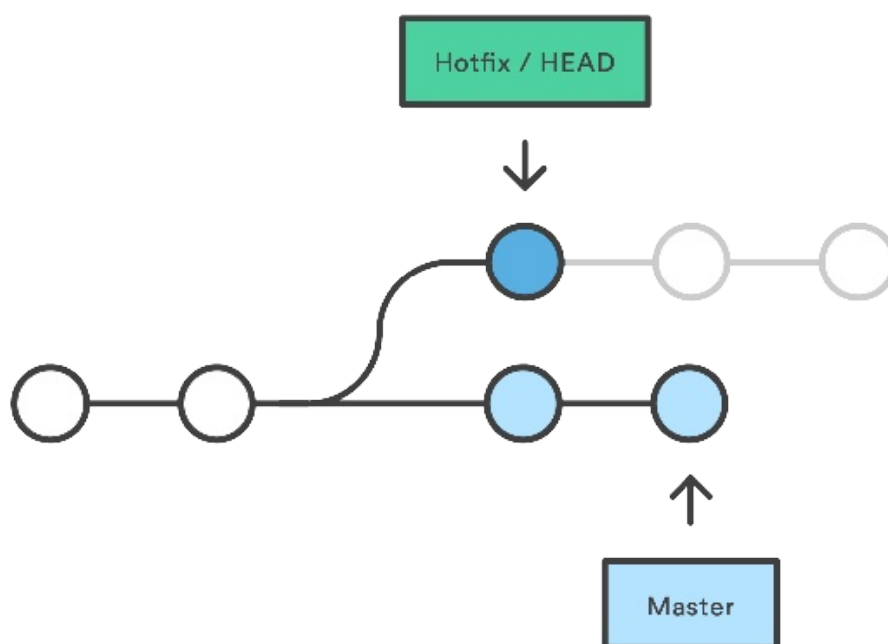
除。换句话说，如果你想扔掉这两个提交，你可以这么做。reset 操作如下图所示：

Resetting the hotfix branch to HEAD-2

Before Resetting



After Resetting



○ - Orphaned Commits

如果你的更改还没有共享给别人，`git reset` 是撤销这些更改的简单方法。当你开发一个功能的时候发现「糟糕，我做了什么？我应该重新来过！」时，`reset` 就像是 `go-to` 命令一样。

除了在当前分支上操作，你还可以通过传入这些标记来修改你的缓存区或工作目录：

- `-soft` - 缓存区和工作目录都不会被改变
- `-mixed` - 默认选项。缓存区和你指定的提交同步，但工作目录不受影响
- `-hard` - 缓存区和工作目录都同步到你指定的提交

把这些标记想成定义 `git reset` 操作的作用域就容易理解多了。



这些标记往往和 `HEAD` 作为参数一起使用。比如，`git reset --mixed HEAD` 将你当前的改动从缓存区中移除，但是这些改动还留在工作目录中。另一方面，如果你想完全舍弃你没有提交的改动，你可以使用 `git reset --hard HEAD`。这是 `git reset` 最常用的两种用法。

当你传入 `HEAD` 以外的其他提交的时候要格外小心，因为 `reset` 操作会重写当前分支的历史。正如 `rebase` 黄金法则所说的，在公共分支上这样做可能会引起严重的后果。

Checkout

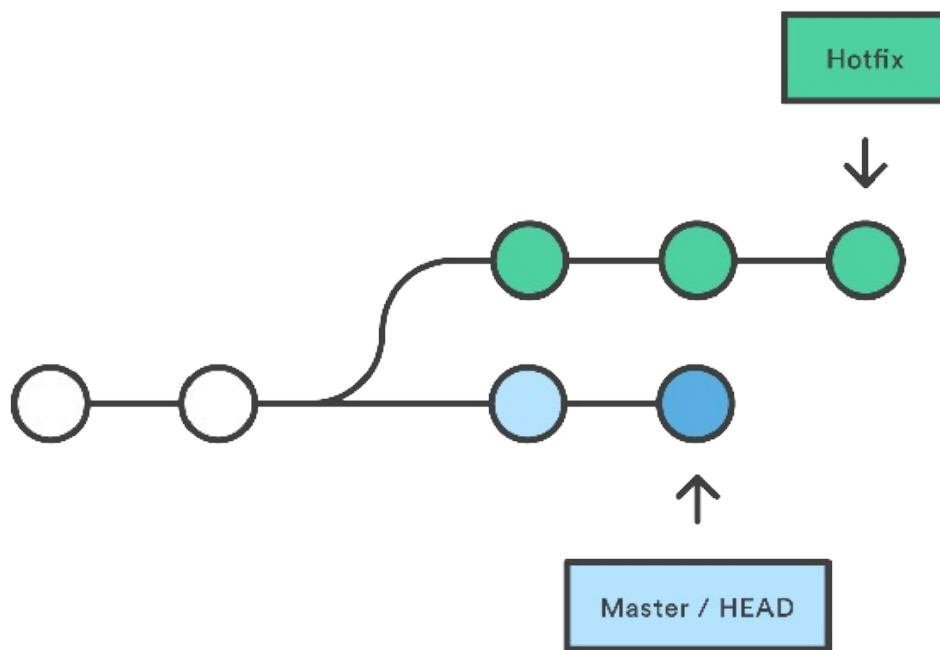
你应该已经非常熟悉提交层面的 `git checkout`。当传入分支名时，可以切换到那个分支。

```
1. git checkout hotfix
```

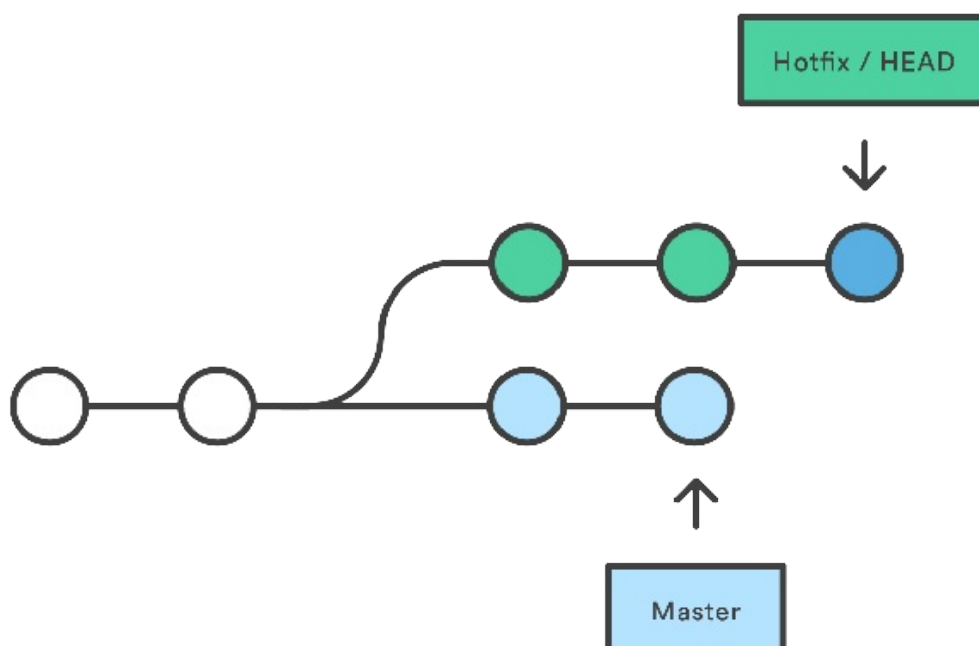
上面这个命令做的不过是将`HEAD`移到一个新的分支，然后更新工作目录。因为这可能会覆盖本地的修改，Git 强制你提交或者缓存工作目录中的所有更改，不然在 `checkout` 的时候这些更改都会丢失。和 `git reset` 不一样的是，`git checkout` 没有移动这些分支。

Moving HEAD from master to hotfix

Before Checking Out



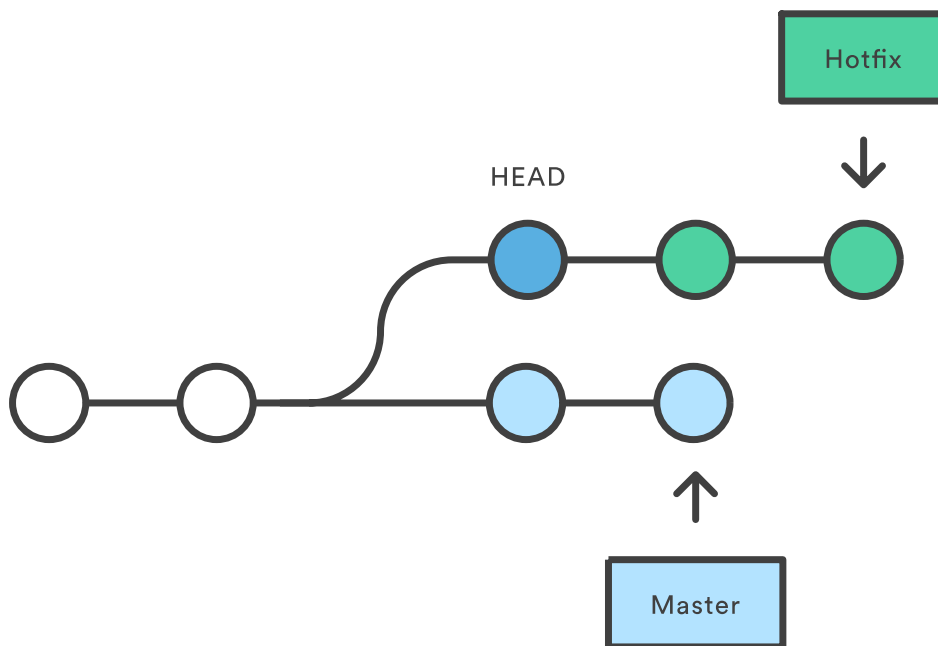
After Checking Out



除了分支之外，你还可以传入提交的引用来 checkout 到任意的提交。这和 checkout 到另一个分支是完全一样的：把 HEAD 移动到特定的提交。比如，下面这个命令会 checkout 到当前提交的祖父提交。

```
1. git checkout HEAD~2
```

Moving HEAD to an arbitrary commit



这对于快速查看项目旧版本来说非常有用。但如果你当前的 HEAD 没有任何分支引用，那么这会造成 HEAD 分离。这是非常危险的，如果你接着添加新的提交，然后切换到别的分支之后就没办法回到之前添加的这些提交。因此，在为分离的 HEAD 添加新的提交的时候你应该创建一个新的分支。

Revert

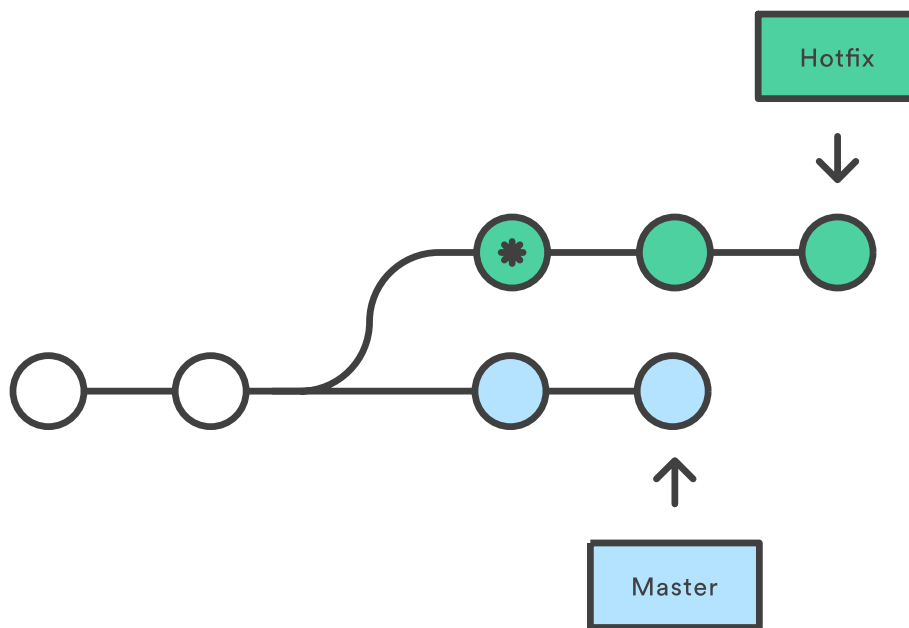
Revert 撤销一个提交的同时会创建一个新的提交。这是一个安全的方法，因为它不会重写提交历史。比如，下面的命令会找出倒数第二个提交，然后创建一个新的提交来撤销这些更改，然后把这个提交加入项目中。

```
1. git checkout hotfix
2. git revert HEAD~2
```

如下图所示：

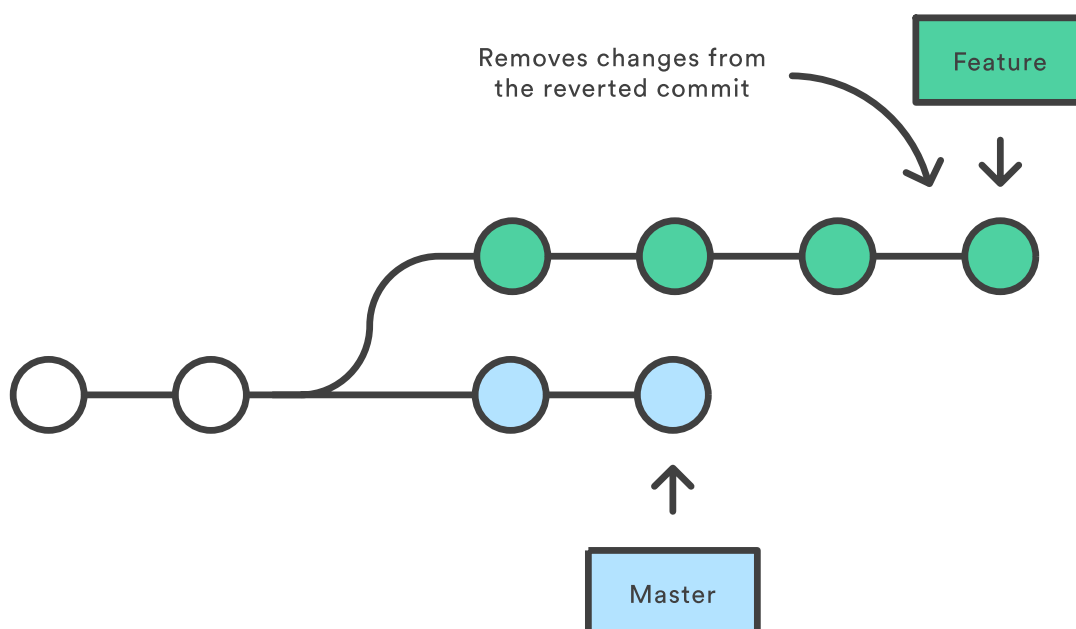
Reverting the 2nd to last commit

Before Reverting



* Commit to be reverted

After Reverting



相比 `git reset`，它不会改变现在的提交历史。因此，`git revert` 可以用在公共分支上，`git reset` 应该用在私有分支上。

你也可以把 `git revert` 当作撤销已经提交的更改，而 `git reset HEAD` 用来撤销没有提交的更改。

就像 `git checkout` 一样，`git revert` 也有可能会重写文件。所以，Git 会在你执行 `revert` 之前要求你提交或者缓存你工作目录中的更改。

文件层面的操作

`git reset` 和 `git checkout` 命令也接受文件路径作为参数。这时它的行为就大为不同了。它不会作用于整份提交，参数将它限制于特定文件。

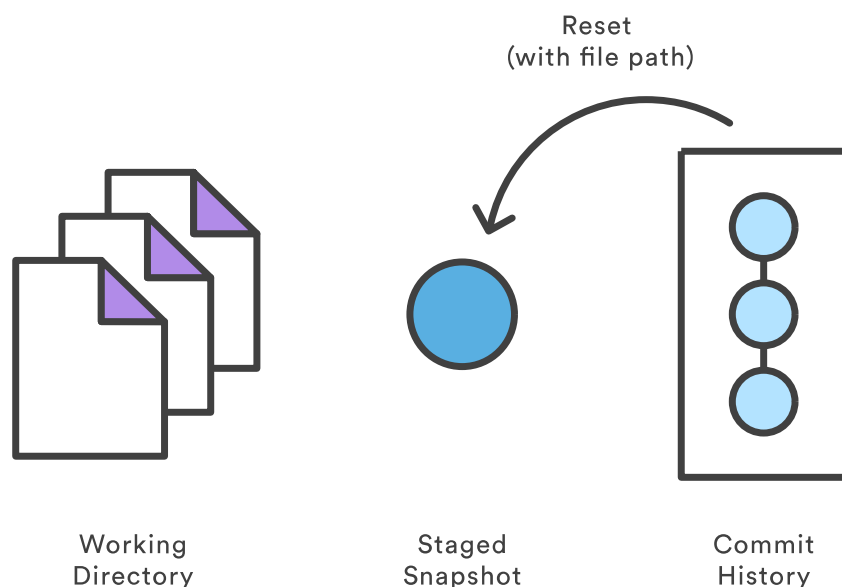
Reset

当检测到文件路径时，`git reset` 将缓存区同步到你指定的那个提交。比如，下面这个命令会将倒数第二个提交中的 `foo.py` 加入到缓存区中，供下一个提交使用。

```
1. git reset HEAD~2 foo.py
```

和提交层面的 `git reset` 一样，通常我们使用HEAD而不是某个特定的提交。运行 `git reset HEAD foo.py` 会将当前的 `foo.py` 从缓存区中移除出去，而不会影响工作目录中对 `foo.py` 的更改。

Moving a file from the commit history into the staged snapshot

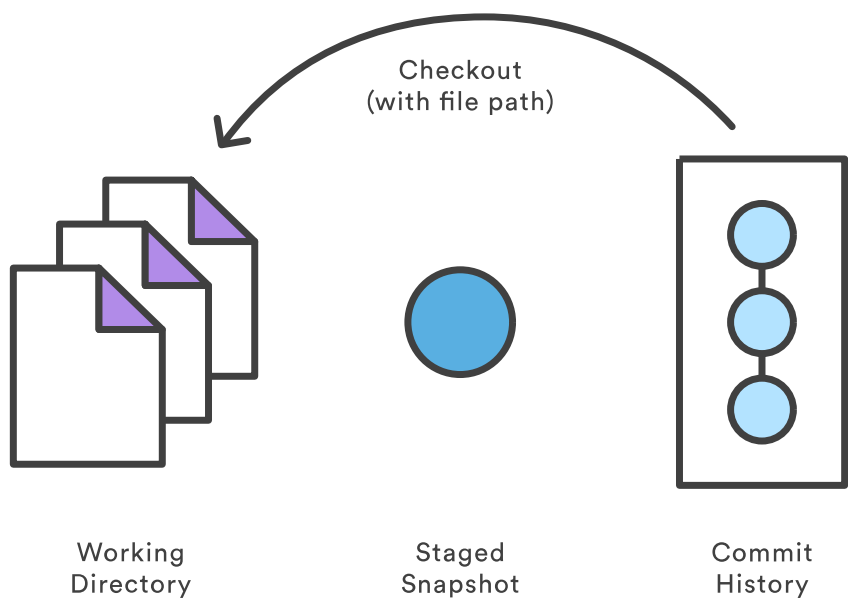


`--soft`、`--mixed` 和 `--hard` 对文件层面的 `git reset` 毫无作用，因为缓存区中的文件一定会变化，而工作目录中的文件一定不变。

Checkout

Checkout 一个文件和带文件路径 `git reset` 非常像，除了它更改的是工作目录而不是缓存区。不像提交层面的 checkout 命令，它不会移动 HEAD引用，也就是你不会切换到别的分支上去。

Moving a file from the commit history into the working directory



比如，下面这个命令将工作目录中的 `foo.py` 同步到了倒数第二个提交中的 `foo.py`。

```
1. git checkout HEAD~2 foo.py
```

和提交层面相同的是，它可以用来检查项目的旧版本，但作用域被限制到了特定文件。

如果你缓存并且提交了 checkout 的文件，它具备将某个文件回撤到之前版本的效果。注意它撤销了这个文件后面所有的更改，而 `git revert` 命令只撤销某个特定提交的更改。

和 `git reset` 一样，这个命令通常和 HEAD 一起使用。比如 `git checkout HEAD foo.py` 等同于舍弃 `foo.py` 没有缓存的更改。这个行为和 `git reset HEAD --hard` 很像，但只影响特定文件。

总结

你已经掌握了 Git 仓库中撤销更改的所有工具。`git reset`、`git checkout` 和 `git revert` 命令比较容易混淆，但当你想起它们对工作目录、缓存区和提交历史的不同影响，就会容易判断现在应该用哪个命令。

下面这个表格总结了这些命令最常用的使用场景。记得经常对照这个表格，因为你使用 Git 时一定会经常用到。

命令	作用域	常用情景
<code>git reset</code>	提交层面	在私有分支上舍弃一些没有提交的更改
<code>git reset</code>	文件层面	将文件从缓存区中移除
<code>git checkout</code>	提交层面	切换分支或查看旧版本
<code>git checkout</code>	文件层面	舍弃工作目录中的更改

git revert	提交层面	在公共分支上回滚更改
git revert	文件层面	（然而并没有）

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 **Star** :star2: 或 **Fork** :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

Git log 高级用法

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。

每一个版本控制系统的出现都是为了让你记录代码的变化。你可以看到项目的历史记录——谁贡献了什么、bug 是什么时候引入的，还可以撤回有问题的更改。但是，首先你得知道如何使用它。这也就是为什么会有 `git log` 这个命令。

到现在为止，你应该已经知道如何用 `git log` 命令来显示最基本的提交信息。但除此之外，你还可以传入各种不同的参数来获得不一样的输出。

`git log` 有两个高级用法：一是自定义提交的输出格式，二是过滤输出哪些提交。这两个用法合二为一，你就可以找到你项目中你需要的任何信息。

格式化 Log 输出

首先，这篇文章会展示几种 `git log` 格式化输出的例子。大多数例子只是通过标记向 `git log` 请求或多或少的信息。

如果你不喜欢默认的 `git log` 格式，你可以用 `git config` 的别名功能来给你想要的格式创建一个快捷方式。

Oneline

`--oneline` 标记把每一个提交压缩到了一行中。它默认只显示提交ID和提交信息的第一行。`git log --oneline` 的输出一般是这样的：

```
1. 0e25143 Merge branch 'feature'
2. ad8621a Fix a bug in the feature
3. 16b36c6 Add a new feature
4. 23ad9ad Add the initial code base
```

它对于获得项目的总体情况很有帮助。

Decorate

很多时候，知道每个提交关联的分支或者标签很有用。`--decorate` 标记让 `git log` 显示指向这个提交的所有引用（比如说分支、标签等）。

这可以和另一个配置项一起使用。比如，执行 `git log --oneline --decorate` 会将提交历史格式化成这样：

```
1. 0e25143 (HEAD, master) Merge branch 'feature'
2. ad8621a (feature) Fix a bug in the feature
3. 16b36c6 Add a new feature
4. 23ad9ad (tag: v0.9) Add the initial code base
```

在这个例子中，你（通过HEAD标记）可以看到最上面那个提交已经被 checkout 了，而且它还是 master 分支的尾端。第二个提交有另一个 feature 分支指向它，以及最后那个提交带有 v0.9 标签。

分支、标签、HEAD 还有提交历史是你 Git 仓库中包含的所有信息。因此，这个命令让你更完整地观察项目结构。

Diff

`git log` 提供了很多选项来显示两个提交之间的差异。其中最常用的两个是 `--stat` 和 `-p`。

`--stat` 选项显示每次提交的文件增删数量（注意：修改一行记作增加一行且删去一行），当你想要查看提交引入的变化时这会非常有用。比如说，下面这个提交在 `hello.py` 文件中增加了 67 行，删去了 38 行。

```
1. commit f2a238924e89ca1d4947662928218a06d39068c3
2. Author: John <john@example.com>
3. Date:   Fri Jun 25 17:30:28 2014 -0500
4.
5.     Add a new feature
6.
7.  hello.py | 105 +++++++++++++++++++++++++++++++++++++-----
8.  1 file changed, 67 insertion(+), 38 deletions(-)
```

文件名后面+和-的数量是这个提交造成的更改中增删的相对比例。它给你一个直观的感觉，关于这次提交有多少改动。如果你想知道每次提交删改的绝对数量，你可以将 `-p` 选项传入 `git log`。这样提交所有的删改都会被输出：

```
1. commit 16b36c697eb2d24302f89aa22d9170dfe609855b
2. Author: Mary <mary@example.com>
3. Date:   Fri Jun 25 17:31:57 2014 -0500
4.
5.     Fix a bug in the feature
6.
7. diff --git a/hello.py b/hello.py
8. index 18ca709..c673b40 100644
9. --- a/hello.py
10. +++ b/hello.py
11. @@ -13,14 +13,14 @@ B
12. -print("Hello, World!")
13. +print("Hello, Git!")
```

对于改动很多的提交来说，这个输出会变得又长又大。一般来说，当你输出所有删改的时候，你是想要查找某一具体的改动，这时你就要用到 `pickaxe` 选项。

Shortlog

`git shortlog` 是一种特殊的 `git log`，它是为创建发布声明设计的。它把每个提交按作者分类，显示提交信息的第一行。这样可以容易地看到谁做了什么。

比如说，两个开发者为项目贡献了 5 个提交，那么 `git shortlog` 输出会是这样的：

```

1. Mary (2):
2.     Fix a bug in the feature
3.     Fix a serious security hole in our framework
4.
5. John (3):
6.     Add the initial code base
7.     Add a new feature
8.     Merge branch 'feature'

```

默认情况下，`git shortlog` 把输出按作者名字排序，但你可以传入 `-n` 选项来按每个作者提交数量排序。

Graph

`--graph` 选项绘制一个 ASCII 图像来展示提交历史的分支结构。它经常和 `--oneline` 和 `--decorate` 两个选项一起使用，这样会更容易查看哪个提交属于哪个分支：

```

1. git log --graph --oneline --decorate
2. For a simple repository with just 2 branches, this will produce the following:
3.
4. * 0e25143 (HEAD, master) Merge branch 'feature'
5. |\
6. | * 16b36c6 Fix a bug in the new feature
7. | * 23ad9ad Start a new feature
8. | * | ad8621a Fix a critical security issue
9. | /
10. * 400e4b7 Fix typos in the documentation
11. * 160e224 Add the initial code base

```

星号表明这个提交所在的分支，所以上图的意思是 `23ad9ad` 和 `16b36c6` 这两个提交在 `topic` 分支上，其余的在 `master` 分支上。

虽然这对简单的项目来说是个很好用的选择，但你可能会更喜欢 `gitk` 或 `SourceTree` 这些更强大的可视化工具来分析大型项目。

自定义格式

对于其他的 `git log` 格式需求，你都可以使用 `--pretty=format:"<string>"` 选项。它允许你使用像 `printf` 一样的占位符来输出提交。

比如，下面命令中的 `%cn`、`%h` 和 `%cd` 这三种占位符会被分别替换为作者名字、缩略标识和提交日期。

```

1. git log --pretty=format:"%cn committed %h on %cd"
2. This results in the following format for each commit:
3.
4. John committed 400e4b7 on Fri Jun 24 12:30:04 2014 -0500
5. John committed 89ab2cf on Thu Jun 23 17:09:42 2014 -0500
6. Mary committed 180e223 on Wed Jun 22 17:21:19 2014 -0500
7. John committed f12ca28 on Wed Jun 22 13:50:31 2014 -0500

```

完整的占位符清单可以在文档中找到。

除了让你只看到关注的信息，这个 `--pretty=format:"<string>"` 选项在你想要在另一个命令中使用日志内容是尤为有用的。

过滤提交历史

格式化提交输出只是 `git log` 其中的一个用途。另一半是理解如何浏览整个提交历史。接下来的文章会介绍如何用 `git log` 选择项目历史中的特定提交。所有的用法都可以和上面讨论过的格式化选项结合起来。

按数量

`git log` 最基础的过滤选项是限制显示的提交数量。当你只对最近几次提交感兴趣时，它可以节省你一页一页查看的时间。

你可以在后面加上 `-<n>` 选项。比如说，下面这个命令会显示最新的 3 次提交：

```
1. git log -3
```

按日期

如果你想要查看某一特定时间段内的提交，你可以使用 `--after` 或 `--before` 标记来按日期筛选。它们都接受好几种日期格式作为参数。比如说，下面的命令会显示 2014 年 7 月 1 日后（含）的提交：

```
1. git log --after="2014-7-1"
```

你也可以传入相对的日期，比如一周前（`"1 week ago"`）或者昨天（`"yesterday"`）：

```
1. git log --after="yesterday"
```

你可以同时提供 `--before` 和 `--after` 来检索两个日期之间的提交。比如，为了显示 2014 年 7 月 1 日到 2014 年 7 月 4 日之间的提交，你可以这么写：

```
1. git log --after="2014-7-1" --before="2014-7-4"
```

注意 `--since`、`--until` 标记和 `--after`、`--before` 标记分别是等价的。

按作者

当你只想看某一特定作者的提交的时候，你可以使用 `--author` 标记。它接受正则表达式，返回所有作者名字满足这个规则的提交。如果你知道那个作者的确切名字你可以直接传入文本字符串：

```
1. git log --author="John"
```

它会显示所有作者叫 John 的提交。作者名不一定是全匹配，只要包含那个子串就会匹配。

你也可以用正则表达式来创建更复杂的检索。比如，下面这个命令检索名叫 Mary 或 John 的作者的提交。

```
1. git log --author="John\|Mary"
```

注意作者的邮箱地址也算作是作者的名字，所以你也可以用这个选项来按邮箱检索。

如果你的工作流区分提交者和作者，`--committer` 也能以相同的方式使用。

按提交信息

按提交信息来过滤提交，你可以使用 `--grep` 标记。它和上面的 `--author` 标记差不多，只不过它搜索的是提交信息而不是作者。

比如说，你的团队规范要求提交信息中包括相关的issue编号，你可以用下面这个命令来显示这个 issue 相关的所有提交：

```
1. git log --grep="JRA-224:"
```

你也可以传入 `-i` 参数来忽略大小写匹配。

按文件

很多时候，你只对某个特定文件的更改感兴趣。为了显示某个特定文件的历史，你只需要传入文件路径。比如说，下面这个命令返回所有和 `foo.py` 和 `bar.py` 文件相关的提交：

```
1. git log -- foo.py bar.py
```

`--` 告诉 `git log` 接下来的参数是文件路径而不是分支名。如果分支名和文件名不可能冲突，你可以省略 `-`。

按内容

我们还可以根据源代码中某一行的增加和删除来搜索提交。这被称为 pickaxe，它接受形如 `-S"<string>"` 的参数。比如说，当你想要知道 `Hello, World!` 字符串是什么时候添加到项目中哪个文件中去的，你可以使用下面这个命令：

```
1. git log -S "Hello, World!"
```

如果你想用正则表达式而不是字符串来搜索，你可以使用 `-G"<regex>"` 标记。

这是一个非常强大的调试工具，它能让你定位到所有影响代码中特定一行的提交。它甚至可以让你看到某一行是什么时候复制或者移动到另一个文件中去的。

按范围

你可以传入范围来筛选提交。这个范围由下面这样的格式指定，其中 `<since>` 和 `<until>` 是提交的引用：

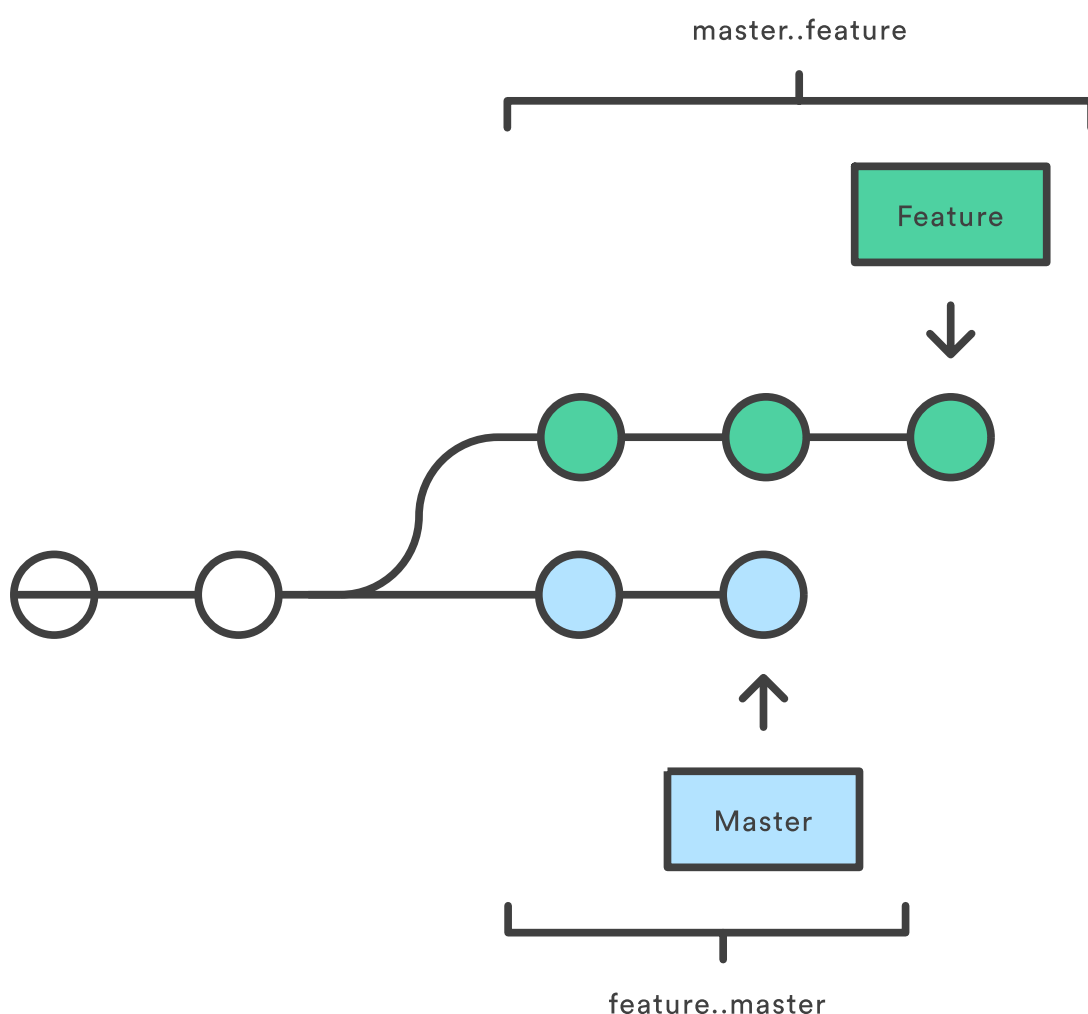
```
1. git log <since>..<until>
```

这个命令在你使用分支引用作为参数时特别有用。这是显示两个分支之间区别最简单的方式。看看下面这个命令：

```
1. git log master..feature
```

其中的 `master..feature` 范围包含了在 `feature` 分支而不在 `master` 分支中所有的提交。换句话说，这个命令可以看出从 `master` 分支 fork 到 `feature` 分支后发生了哪些变化。它可以这样可视化：

Detecting a fork in the history using ranges



注意如果你更改范围的前后顺序（`feature..master`），你会获取到 `master` 分支而非 `feature` 分支上的所有提交。如果 `git log` 输出了全部两个分支的提交，这说明你的提交历史已经分叉了。

过滤合并提交

`git log` 输出时默认包括合并提交。但是，如果你的团队采用强制合并策略（意思是 `merge` 你修改的上游分支而不是将你的分支 `rebase` 到上游分支），你的项目历史中会有很多外来的提交。

你可以通过 `--no-merges` 标记来排除这些提交：

```
1. git log --no-merges
```

另一方面，如果你只对合并提交感兴趣，你可以使用 `--merges` 标记：

```
1. git log --merges
```

它会返回所有包含两个父节点的提交。

总结

你现在应该对使用 `git log` 来格式化输出和选择你要显示的提交的用法比较熟悉了。它允许你查看你项目历史中任何需要的内容。

这些技巧是你 Git 工具箱中重要的部分，不过注意 `git log` 往往和其他 Git 命令连着使用。当你找到了你要的提交，你把它传给 `git checkout`、`git revert` 或是其他控制提交历史的工具。所以，请继续坚持 Git 高级用法的学习。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

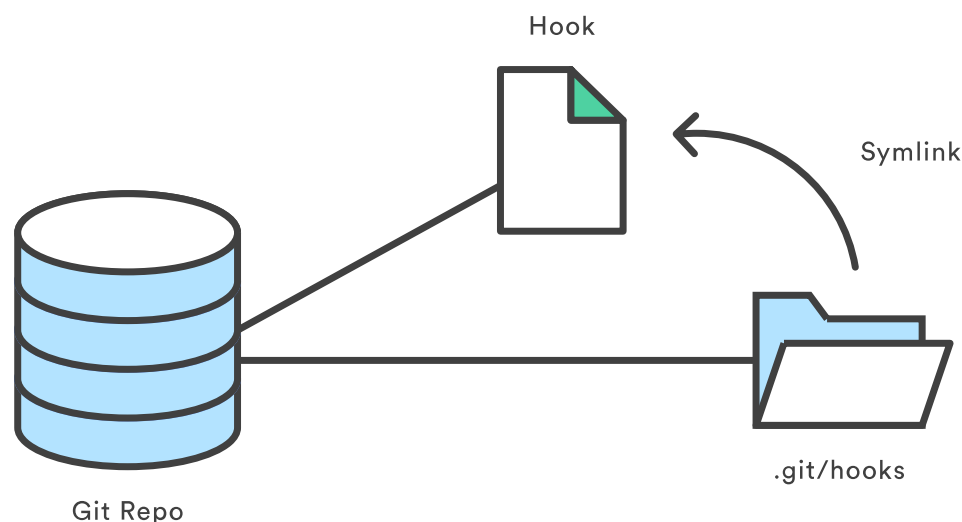
Git 钩子：自定义你的工作流

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 钩子是在 Git 仓库中特定事件发生时自动运行的脚本。它可以让你自定义 Git 内部的行为，在开发周期中的关键点触发自定义的行为。

Maintaining a hook using a symlink to version-controlled script



Git 钩子最常见的使用场景包括推行提交规范，根据仓库状态改变项目环境，和接入持续集成工作流。但是，因为脚本可以完全定制，你可以用 Git 钩子来自动化或者优化你开发工作流中任意部分。

在这篇文章中，我们会先简要介绍 Git 钩子是如何工作的。然后，我们会审视一些本地和远端仓库使用最流行的钩子。

概述

Git 钩子是仓库中特定事件发生时 Git 自动运行的普通脚本。因此，Git 钩子安装和配置也非常容易。

钩子在本地或服务端仓库都可以部署，且只会在仓库中事件发生时被执行。在文章后面我们会具体地研究各种钩子。接下来所讲的配置对本地和服务端钩子都起作用。

安装钩子

钩子存在于每个 Git 仓库的 `.git/hooks` 目录中。当你初始化仓库时，Git 自动生成这个目录和一些示例脚本。当你观察 `.git/hooks` 时，你会看到下面这些文件：

- | | |
|---------------------------------------|--------------------------------|
| 1. <code>applypatch-msg.sample</code> | <code>pre-push.sample</code> |
| 2. <code>commit-msg.sample</code> | <code>pre-rebase.sample</code> |

```

3. post-update.sample      prepare-commit-msg.sample
4. pre-applypatch.sample   update.sample
5. pre-commit.sample

```

这里已经包含了大部分可用的钩子了，但是 `.sample` 拓展名防止它们默认被执行。为了安装一个钩子，你只需要去掉 `.sample` 拓展名。或者你要写一个新的脚本，你只需添加一个文件名和上述匹配的新文件，去掉 `.sample` 拓展名。

比如说，试试安装一个 `prepare-commit-msg` 钩子。去掉脚本的 `.sample` 拓展名，在文件中加上下面这两行：

```

1. #!/bin/sh
2.
3. echo "# Please include a useful commit message!" > $1

```

钩子需要能被执行，所以如果你创建了一个新的脚本文件，你需要修改它的文件权限。比如说，为了确保 `prepare-commit-msg` 可执行，运行下面这个命令：

```
1. chmod +x prepare-commit-msg
```

接下来你每次运行 `git commit` 时，你会看到默认的提交信息都被替换了。我们会在「准备提交信息」一节中细看它是如何工作的。现在我们已经可以定制 Git 的内部功能，你只需要坐和放宽。

内置的样例脚本是非常有用的参考资料，因为每个钩子传入的参数都有非常详细的说明（不同钩子不一样）。

脚本语言

内置的脚本大多是 shell 和 PERL 语言的，但你可以使用任何脚本语言，只要它们最后能编译到可执行文件。每次脚本中的 `#!/bin/sh` 定义了你的文件将被如何解释。比如，使用其他语言时你只需要将 `path` 改为你的解释器的路径。

比如说，你可以在 `prepare-commit-msg` 中写一个可执行的 Python 脚本。下面这个钩子和上一节的 shell 脚本做的事完全一样。

```

1. #!/usr/bin/env python
2.
3. import sys, os
4.
5. commit_msg_filepath = sys.argv[1]
6. with open(commit_msg_filepath, 'w') as f:
7.     f.write("# Please include a useful commit message!")

```

注意第一行改成了 Python 解释器的路径。此外，这里用 `sys.argv[1]` 而不是 `$1` 来获取第一个参数（这个也后面再讲）。

这个特性非常强大，因为你可以用任何你喜欢的语言来编写 Git 钩子。

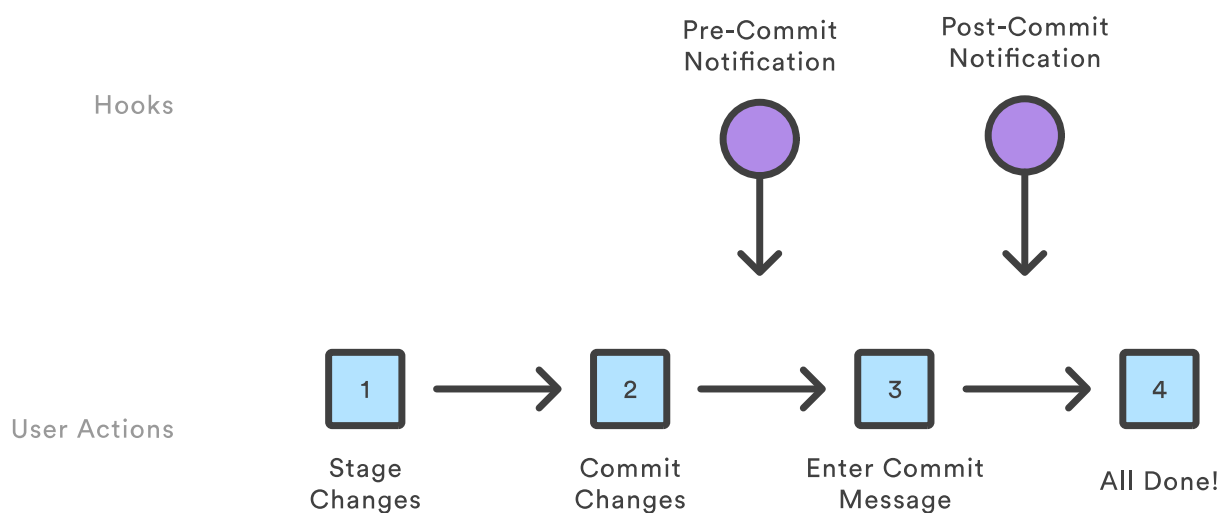
钩子的作用域

对于任何 Git 仓库来说钩子都是本地的，而且它不会随着 `git clone` 一起复制到新的仓库。而且，因为钩子是本地的，任何能接触得到仓库的人都可以修改。

对于开发团队来说，这有很大的影响。首先，你要确保你们成员之间的钩子都是最新的。其次，你也不能强行让其他人用你喜欢的方式提交——你只能鼓励他们这样做。

在开发团队中维护钩子是比较复杂的，因为 `.git/hooks` 目录不随你的项目一起拷贝，也不受版本控制影响。一个简单的解决办法是把你的钩子存在项目的实际目录中（在 `.git` 外）。这样你就可以像其他文件一样进行版本控制。为了安装钩子，你可以在 `.git/hooks` 中创建一个符号链接，或者简单地在更新后把它们复制到 `.git/hooks` 目录下。

Hooks executing during the commit creation process



作为备选方案，Git 同样提供了一个模板目录机制来更简单地自动安装钩子。每次你使用 `git init` 或 `git clone` 时，模板目录文件夹下的所有文件和目录都会被复制到 `.git` 文件夹。

所有的下面讲到的本地钩子都可以被更改或者彻底删除，只要你是项目的参与者。这完全取决于你的团队成员想不想用这个钩子。所以记住，最好把 Git 钩子当成一个方便的开发者工具而不是一个严格强制的开发规范。

也就是说，用服务端钩子来拒绝没有遵守规范的提交是完全可行的。后面我们会再讨论这个问题。

本地钩子

本地钩子只影响它们所在的仓库。当你在读这一节的时候，记住开发者可以修改他们本地的钩子，所以不要用它们来推行强制的提交规范。不过，它们确实可以让开发者更易于接受这些规范。

在这一节中，我们会探讨 6 个最有用的本地钩子：

- pre-commit
- prepare-commit-msg
- commit-msg
- post-commit

- `post-checkout`
- `pre-rebase`

前四个钩子让你介入完整的提交生命周期，后两个允许你执行一些额外的操作，分别为 `git checkout` 和 `git rebase` 的安全检查。

所有带 `pre-` 的钩子允许你修改即将发生的操作，而带 `post-` 的钩子只能用于通知。

我们也会看到处理钩子的参数和用底层 `Git` 命令获取仓库信息的实用技巧。

pre-commit

`pre-commit` 脚本在每次你运行 `git commit` 命令时，`Git` 向你询问提交信息或者生产提交对象时被执行。你可以用这个钩子来检查即将被提交的代码快照。比如说，你可以运行一些自动化测试，保证这个提交不会破坏现有的功能。

`pre-commit` 不需要任何参数，以非0状态退出时将放弃整个提交。让我们看一个简化了的（和更详细的）内置 `pre-commit` 钩子。只要检测到不一致时脚本就放弃这个提交，就像 `git diff-index` 命令定义的那样（只要词尾有空白字符、只有空白字符的行、行首一个 `tab` 后紧接一个空格就被认为错误）。

```
1. #!/bin/sh
2.
3. # 检查这是否是初始提交
4. if git rev-parse --verify HEAD >/dev/null 2>&1
5. then
6.     echo "pre-commit: About to create a new commit..."
7.     against=HEAD
8. else
9.     echo "pre-commit: About to create the first commit..."
10.    against=4b825dc642cb6eb9a060e54bf8d69288fbee4904
11. fi
12.
13. # 使用git diff-index来检查空白字符错误
14. echo "pre-commit: Testing for whitespace errors..."
15. if ! git diff-index --check --cached $against
16. then
17.     echo "pre-commit: Aborting commit due to whitespace errors"
18.     exit 1
19. else
20.     echo "pre-commit: No whitespace errors :)"
21.     exit 0
22. fi
```

使用 `git diff-index` 时我们要指出和哪次提交进行比较。一般来说是 `HEAD`，但 `HEAD` 在创建第一次提交时不存在，所以我们的第一个任务是解决这个极端情形。我们通过 `git rev-parse --verify` 来检查 `HEAD` 是否是一个合法的引用。`>/dev/null 2>&1` 这部分屏蔽了 `git rev-parse` 任何输出。`HEAD` 或者一个新的提交对象被储存在 `against` 变量中供 `git diff-index` 使用。`4b825d...` 这个哈希字符串代表一个空白提交的 ID。

`git diff-index --cached` 命令将提交和缓存区比较。通过传入 `-check` 选项，我们要求它在更改引入空白字符错误时警告我们。如果它这么做了，我们返回状态1来放弃这次提交，否则返回状态 0，提交 workflow 正常进行。

这只是 `pre-commit` 的其中一个例子。它恰好使用了已有的 `Git` 命令来根据提交带来的更改进行测试，但你可以在 `pre-commit` 中做任何你想做的事，比如执行其它脚本、运行第三方测试集、用 `Lint` 检查代码风格。

prepare-commit-msg

`prepare-commit-msg` 钩子在 `pre-commit` 钩子在文本编辑器中生成提交信息之后被调用。这被用来方便地修改自动生成的 `squash` 或 `merge` 提交。

`prepare-commit-msg` 脚本的参数可以是下列三个：

- 包含提交信息的文件名。你可以在原地更改提交信息。
- 提交类型。可以是信息（`-m` 或 `-F` 选项），模板（`-t` 选项），`merge`（如果是个合并提交）或 `squash`（如果这个提交插入了其他提交）。
- 相关提交的 `SHA1` 哈希字符串。只有当 `-c`、`-C` 或 `--amend` 选项出现时才需要。

和 `pre-commit` 一样，以非0状态退出会放弃提交。

我们已经看过一个修改提交信息的简单例子，现在我们来查看一个更有用的脚本。使用 `issue` 跟踪器时，我们通常在单独的分支上处理 `issue`。如果你在分支名中包含了 `issue` 编号，你可以使用 `prepare-commit-msg` 钩子来自动地将它包括在那个分支的每个提交信息中。

```
1. #!/usr/bin/env python
2.
3. import sys, os, re
4. from subprocess import check_output
5.
6. # 收集参数
7. commit_msg_filepath = sys.argv[1]
8. if len(sys.argv) > 2:
9.     commit_type = sys.argv[2]
10. else:
11.     commit_type = ''
12. if len(sys.argv) > 3:
13.     commit_hash = sys.argv[3]
14. else:
15.     commit_hash = ''
16.
17. print "prepare-commit-msg: File: %s\nType: %s\nHash: %s" % (commit_msg_filepath, commit_type, commit_hash)
18.
19. # 检测我们所在的分支
20. branch = check_output(['git', 'symbolic-ref', '--short', 'HEAD']).strip()
21. print "prepare-commit-msg: On branch '%s'" % branch
22.
23. # 用issue编号生成提交信息
24. if branch.startswith('issue-'):
25.     print "prepare-commit-msg: Oh hey, it's an issue branch."
26.     result = re.match('issue-(.*)', branch)
27.     issue_number = result.group(1)
28.
29.     with open(commit_msg_filepath, 'r+') as f:
30.         content = f.read()
```

```

31.         f.seek(0, 0)
32.         f.write("ISSUE-%s %s" % (issue_number, content))

```

首先，上面的 `prepare-commit-msg` 钩子告诉你如何收集传入脚本的所有参数。接下来，它调用了 `git symbolic-ref --short HEAD` 来获取对应 HEAD 的分支名。如果分支名以 `issue-` 开头，它会重写提交信息文件，在第一行加上 issue 编号。比如你的分支名 `issue-224`，下面的提交信息将会生成：

```

1. ISSUE-224
2.
3. # Please enter the commit message for your changes. Lines starting
4. # with '#' will be ignored, and an empty message aborts the commit.
5. # On branch issue-224
6. # Changes to be committed:
7. #   modified:   test.txt

```

有一点要记住的是即使用户用 `-m` 传入提交信息，`prepare-commit-msg` 也会运行。也就是说，上面这个脚本会自动插入 `ISSUE-[#]` 字符串，而用户无法更改。你可以检查第二个参数是否是提交类型来处理这个情况。

但是，如果没有 `-m` 选项，`prepare-commit-msg` 钩子允许用户修改生成后的提交信息。所以脚本的目的是为了方便，而不是推行强制的提交信息规范。如果你要这么做，你需要下一节所讲的 `commit-msg` 钩子。

commit-msg

`commit-msg` 钩子和 `prepare-commit-msg` 钩子很像，但它会在用户输入提交信息之后被调用。这适合用来提醒开发者他们的提交信息不符合你团队的规范。

传入这个钩子唯一的参数是包含提交信息的文件名。如果它不喜欢用户输入的提交信息，它可以在原地修改这个文件（和 `prepare-commit-msg` 一样），或者它会以非 0 状态退出，放弃这个提交。

比如说，下面这个脚本确认用户没有删除 `prepare-commit-msg` 脚本自动生成的 `ISSUE-[#]` 字符串。

```

1. #!/usr/bin/env python
2.
3. import sys, os, re
4. from subprocess import check_output
5.
6. # 收集参数
7. commit_msg_filepath = sys.argv[1]
8.
9. # 检测所在的分支
10. branch = check_output(['git', 'symbolic-ref', '--short', 'HEAD']).strip()
11. print "commit-msg: On branch '%s'" % branch
12.
13. # 检测提交信息，判断是否是一个issue提交
14. if branch.startswith('issue-'):
15.     print "commit-msg: Oh hey, it's an issue branch."
16.     result = re.match('issue-(.*)', branch)
17.     issue_number = result.group(1)
18.     required_message = "ISSUE-%s" % issue_number
19.

```

```

20.     with open(commit_msg_filepath, 'r') as f:
21.         content = f.read()
22.         if not content.startswith(required_message):
23.             print "commit-msg: ERROR! The commit message must start with '%s'" % required_message
24.             sys.exit(1)

```

虽然用户每次创建提交时，这个脚本都会运行。但你还是应该避免做检查提交信息之外的事情。如果你需要通知其他服务一个快照已经被提交了，你应该使用 `post-commit` 这个钩子。

post-commit

`post-commit` 钩子在 `commit-msg` 钩子之后立即被运行。它无法更改 `git commit` 的结果，所以这主要用于通知用途。

这个脚本没有参数，而且退出状态不会影响提交。对于大多数 `post-commit` 脚本来说，你只是想访问你刚刚创建的提交。你可以用 `git rev-parse HEAD` 来获得最近一次提交的SHA1哈希字符串，或者你可以用 `git log -1 HEAD` 获取完整的信息。

比如说，如果你需要每次提交快照时向老板发封邮件（也许对于大多数 workflow 来说这不是个好的想法），你可以加上下面这个 `post-commit` 钩子。

```

1.  #!/usr/bin/env python
2.
3.  import smtplib
4.  from email.mime.text import MIMEText
5.  from subprocess import check_output
6.
7.  # 获得新提交的git log --stat输出
8.  log = check_output(['git', 'log', '-1', '--stat', 'HEAD'])
9.
10. # 创建一个纯文本的邮件内容
11. msg = MIMEText("Look, I'm actually doing some work:\n\n%s" % log)
12.
13. msg['Subject'] = 'Git post-commit hook notification'
14. msg['From'] = 'mary@example.com'
15. msg['To'] = 'boss@example.com'
16.
17. # 发送信息
18. SMTP_SERVER = 'smtp.example.com'
19. SMTP_PORT = 587
20.
21. session = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
22. session.ehlo()
23. session.starttls()
24. session.ehlo()
25. session.login(msg['From'], 'secretPassword')
26.
27. session.sendmail(msg['From'], msg['To'], msg.as_string())
28. session.quit()

```

你虽然可以用 `post-commit` 来触发本地的持续集成系统，但大多数时候你想用的是 `post-receive` 这个钩子。它运行在服务端而不是用户的本地机器，它同样在任何开发者推送代码时运行。那里更适合你进行持续集成。

post-checkout

`post-checkout` 钩子和 `post-commit` 钩子很像，但它在你用 `git checkout` 查看引用的时候被调用。这是用来清理你的工作目录中可能会令人困惑的生成文件。

这个钩子接受三个参数，它的返回状态不影响 `git checkout` 命令。

- HEAD 前一次提交的引用
- 新的 HEAD 的引用
- 1 或 0，分别代表是分支 checkout 还是文件 checkout。

Python 程序员经常遇到的问题是切换分支后那些之前生成的 `.pyc` 文件。解释器有时使用 `.pyc` 而不是 `.py` 文件。为了避免歧义，你可以在每次用 `post-checkout` 切换到新的分支的时候，删除所有 `.pyc` 文件。

```
1. #!/usr/bin/env python
2.
3. import sys, os, re
4. from subprocess import check_output
5.
6. # 收集参数
7. previous_head = sys.argv[1]
8. new_head = sys.argv[2]
9. is_branch_checkout = sys.argv[3]
10.
11. if is_branch_checkout == "0":
12.     print "post-checkout: This is a file checkout. Nothing to do."
13.     sys.exit(0)
14.
15. print "post-checkout: Deleting all '.pyc' files in working directory"
16. for root, dirs, files in os.walk('.'):
17.     for filename in files:
18.         ext = os.path.splitext(filename)[1]
19.         if ext == '.pyc':
20.             os.unlink(os.path.join(root, filename))
```

钩子脚本当前的工作目录总是位于仓库的根目录下，所以 `os.walk('.')` 调用遍历了仓库中所有文件。接下来，我们检查它的拓展名，如果是 `.pyc` 就删除它。

通过 `post-checkout` 钩子，你还可以根据你切换的分支来来更改工作目录。比如说，你可以在代码库外面使用一个插件分支来储存你所有的插件。如果这些插件需要很多二进制文件而其他分支不需要，你可以选择只在插件分支上 build。

pre-rebase

`pre-rebase` 钩子在 `git rebase` 发生更改之前运行，确保不会有什么糟糕的事情发生。

这个钩子有两个参数：fork 之前的上游分支，将要 rebase 的下游分支。如果 rebase 当前分支则第二个参数为

空。以非 0 状态退出会放弃这次 rebase。

比如说，如果你想彻底禁用 rebase 操作，你可以使用下面的 `pre-rebase` 脚本：

```
1. #!/bin/sh
2.
3. # 禁用所有rebase
4. echo "pre-rebase: Rebasing is dangerous. Don't do it."
5. exit 1
```

每次运行 `git rebase`，你都会看到下面的信息：

```
1. pre-rebase: Rebasing is dangerous. Don't do it.
2. The pre-rebase hook refused to rebase.
```

内置的 `pre-rebase.sample` 脚本是一个更复杂的例子。它在何时阻止 rebase 这方面更加智能。它会检查你当前的分支是否已经合并到了下一个分支中去（也就是主分支）。如果是的话，rebase 可能会遇到问题，脚本会放弃这次 rebase。

服务端钩子

服务端钩子和本地钩子几乎一样，只不过它们存在于服务端的仓库中（比如说中心仓库，或者开发者的公共仓库）。当和官方仓库连接时，其中一些可以用来拒绝一些不符合规范的提交。

这节中我们要讨论下面三个服务端钩子：

- pre-receive
- update
- post-receive

这些钩子都允许你对 `git push` 的不同阶段做出响应。

服务端钩子的输出会传送到客户端的控制台中，所以给开发者发送信息是很容易的。但你要记住这些脚本在结束完之前都不会返回控制台的控制权，所以你要小心那些长时间运行的操作。

pre-receive

`pre-receive` 钩子在有人用 `git push` 向仓库推送代码时被执行。它只存在于远端仓库中，而不是原来的仓库中。

这个钩子在任意引用被更新前被执行，所以这是强制推行开发规范的好地方。如果你不喜欢推送的那个人（多大仇 = ），提交信息的格式，或者提交的更改，你都可以拒绝这次提交。虽然你不能阻止开发者写出糟糕的代码，但你可以用 `pre-receive` 防止这些代码流入官方的代码库。

这个脚本没有参数，但每一个推送上来的引用都会以下面的格式传入脚本的单独一行：

```
1. <old-value> <new-value> <ref-name>
```

你可以看到这个钩子做了非常简单的事，就是读取推送上来的引用并且把它们打印出来。

```
1. #!/usr/bin/env python
2.
3. import sys
4. import fileinput
5.
6. # 读取用户试图更新的所有引用
7. for line in fileinput.input():
8.     print "pre-receive: Trying to push ref: %s" % line
9.
10. # 放弃推送
11. # sys.exit(1)
```

这和其它钩子相比略微有些不同，因为信息是通过标准输入而不是命令行传入的。在远端仓库的 `.git/hooks` 中加上这个脚本，推送到 `master` 分支，你会看到下面这些信息打印出来：

```
1. b6b36c697eb2d24302f89aa22d9170dfe609855b 85baa88c22b52ddd24d71f05db31f4e46d579095 refs/heads/master
```

你可以用 SHA1 哈希字符串，或者底层的 Git 命令，来检查将要引入的更改。一些常见的使用包括：

- 拒绝将上游分支 `rebase` 的更改
- 防止错综复杂的合并（非快速向前，会造成项目历史非线性）
- 检查用户是否有正确的权限来做这些更改（大多用于中心化的 Git 工作流中）
- 如果多个引用被推送，在 `pre-receive` 中返回非 0 状态，拒绝所有提交。如果你想一个个接受或拒绝分支，你需要使用 `update` 钩子

update

`update` 钩子在 `pre-receive` 之后被调用，用法也差不多。它也是在实际更新前被调用的，但它可以分别被每个推送上来的引用分别调用。也就是说如果用户尝试推送到4个分支，`update` 会被执行 4 次。和 `pre-receive` 不一样，这个钩子不需要读取标准输入。事实上，它接受三个参数：

- 更新的引用名称
- 引用中存放的旧的对象名称
- 引用中存放的新的对象名称

这些信息和 `pre-receive` 相同，但因为每次引用都会分别触发更新，你可以拒绝一些引用而接受另一些。

```
1. #!/usr/bin/env python
2.
3. import sys
4.
5. branch = sys.argv[1]
6. old_commit = sys.argv[2]
7. new_commit = sys.argv[3]
8.
9. print "Moving '%s' from %s to %s" % (branch, old_commit, new_commit)
10.
```

```
11. # 只放弃当前分支的推送
12. # sys.exit(1)
```

上面这个钩子简单地输出了分支和新旧提交的哈希字符串。当你向远程仓库推送超过一个分支时，你可以看到每个分支都有输出。

post-receive

`post-receive` 钩子在成功推送后被调用，适合用于发送通知。对很多工作流来说，这是一个比 `post-commit` 更好的发送通知的地方，因为这些更改在公共的服务器而不是用户的本地机器上。给其他开发者发送邮件或者触发一个持续集成系统都是 `post-receive` 常用的操作。

这个脚本没有参数，但和 `pre-receive` 一样通过标准输入读取。

总结

在这篇文章中，我们学习了如果用 Git 钩子来修改内部行为，当仓库中特定的事件发生时接受消息。钩子是存在于 `git/hooks` 仓库中的普通脚本，因此也非常容易安装和定制。

我们还看了一些常用的本地和服务端的钩子。这使得我们能够介入到整个开发生命周期中去。我们现在知道了如何在创建提交或推送的每个阶段执行自定义的操作。有了这些简单的脚本知识，你就可以对 Git 仓库为所欲为了：]

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 `Star` :star2: 或 `Fork` :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。

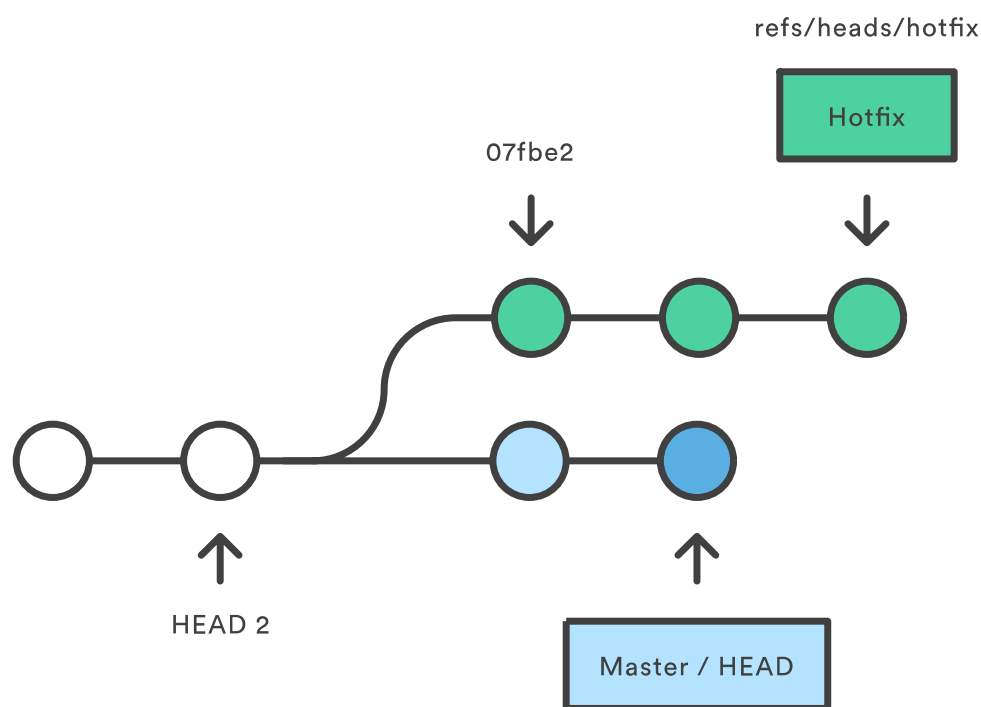
Git提交引用和引用日志

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。

提交是 Git 的精髓所在，你无时不刻不在创建和缓存提交、查看以前的提交，或者用各种Git命令在仓库间转移你的提交。大多数的命令都对同一个提交操作，而有些会接受提交的引用作为参数。比如，你可以给 `git checkout` 传入一个引用来查看以前的提交，或者传入一个分支名来切换到对应的分支。

Many ways of referring to a commit



知道提交的各种引用方式之后，Git 的命令就会变得更加强大。在这章中，我们研究提交的各种引用方式，来一窥 `git checkout`、`git branch`、`git push` 等命令的工作原理。

我们还会学到如何使用 Git 的引用日志查看似乎已被删除的提交。

哈希字符串

引用一个提交最直接的方式是通过 SHA-1 的哈希字符串，这是每个提交唯一的 ID。你可以在 `git log` 的输出中找到提交的哈希字符串。

```
1. commit 0c708fdec272bc4446c6cabea4f0022c2b616eba
2. Author: Mary Johnson <mary@example.com>
3. Date:   Wed Jul 9 16:37:42 2014 -0500
4.
5.     一些提交信息
```

在 Git 命令中传递时，你只需要提供足以确定那个提交的哈希子串即可。比如，你可以这样用 `git show` 的命令显示上面的提交：

```
1. git show 0c708f
```

有时，我们需要把分支、标签或者其他间接的引用转变成对应提交的哈希。`git rev-parse` 命令正是你需要的。下面这个命令返回 master 分支提交的哈希字符串：

```
1. git rev-parse master
```

当你写的自定义脚本中需要将提交引用作为参数时，这个命令非常有用。你可以让 `git rev-parse` 帮你处理转换，而不用手动做这件事。

引用

ref 是提交的间接引用。你可以把它当做哈希字符串的别名，但对用户更友好。这就是 Git 内部表示分支和标签的机制。

引用以一段普通的文本存在于 `.git/refs` 目录中，就是我们平时说的那个 `.git`。你去 `.git/refs` 文件夹查看仓库中的引用。你可以看到下面这样的结构，但具体的文件取决于你的仓库中有什么分支和标签，以及你的远程仓库。

```
1. .git/refs/  
2.     heads/  
3.         master  
4.         some-feature  
5.     remotes/  
6.         origin/  
7.             master  
8.     tags/  
9.         v0.9
```

`heads` 目录定义了你本地仓库中的所有分支。每一个文件名和你的分支名一一对应，文件中包含一个提交的哈希字符串。这个就是分支顶端的所在位置。为了验证这一点，试试在 Git 根目录运行下面这两个命令：

```
1. # 输出`refs/heads/master`文件内容  
2. cat .git/refs/heads/master  
3.  
4. # 查看`master`分支尾端的提交  
5. git log -1 master
```

`cat` 命令返回的哈希字符串和 `git log` 命令显示的哈希字符串应该是一致的。

如果要改变 master 分支的位置，Git 只需要更改 `refs/heads/master` 的文件内容。同样地，创建新的分支也只需要将当前提交的哈希字符串写入到新的文件中。这也是为什么 Git 分支比 SVN 轻量那么多的其中一个原因。

`tags` 目录也是以相同的方式存储，只不过其中存的是标签而不是分支。`remotes` 目录将你之前用 `git remote` 命令创建的所有远程仓库以子目录的形式一一列出。在每个文件夹中，你可以找到所有 `fetch` 到本地仓库的远程分支。

指定引用

当你向 `Git` 命令传入引用的时候，你既可以指定引用完整的名称，也可以使用缩写，然后让 `Git` 来寻找匹配。你应该已经对引用的缩写很熟悉了，每次你通过名称引用分支的时候都会这么做。

```
1. git show some-feature
```

这里的 `some-feature` 参数其实是分支名的缩写。`Git` 在使用前将它解析成 `refs/heads/some-feature`。你也可以在命令行中指定引用的全称，就像这样：

```
1. git show refs/heads/some-feature
```

这避免了引用可能产生的所有歧义。这是非常必要的，比如你同时有一个标签和分支都叫 `some-feature`。然而，如果使用正常的命名规范，你不应该有这样的歧义。

我们会在 `refspec` 一节见到更多引用名称。

打包引用目录

对于大型仓库，`Git` 会周期性地执行垃圾回收来移除不需要的对象，将所有引用文件压缩成单个文件来获得更好的性能。你可以使用这个命令强制垃圾回收来执行压缩：

```
1. git gc
```

这个命令把 `refs` 文件夹中所有单独的分支和标签移动到了 `.git` 根目录下的 `packed-refs` 文件中。如果你打开这个文件，你会发现提交的哈希字串和引用之间的映射关系：

```
1. 00f54250cf4e549fdfcafe2cf9a2c90bc3800285 refs/heads/feature
2. 0e25143693cfe9d5c2e83944bbaf6d3c4505eb17 refs/heads/master
3. bb883e4c91c870b5fed88fd36696e752fb6cf8e6 refs/tags/v0.9
```

另一方面，正常的 `Git` 功能不会受到任何影响。但如果你好奇你的 `.git/refs` 文件夹为什么是空的，这一节告诉你了答案。

特殊的引用

除了 `refs` 文件夹外，`.git` 根目录还有一些特殊的引用。如下所示：

- `HEAD` - 当前所在的提交或分支。
- `FETCH_HEAD` - 远程仓库中 `fetch` 到的最新一次提交。
- `ORIG_HEAD` - `HEAD` 的备份引用，避免损坏。

- MERGE_HEAD - 你通过 `git merge` 并入当前分支的引用（们）。
- CHERRY_PICK_HEAD - 你 `cherry pick` 使用的引用。

这些引用由 Git 在需要时创建和更新。比如说，`git pull` 命令首先运行 `git fetch`，而 `FETCH_HEAD` 引用随之改变。然后，运行 `git merge FETCH_HEAD` 来将 fetch 到的分支最终并入仓库。当然，你也可以使用其他任何引用，因为我相信你已经对 `HEAD` 很熟悉了。

这些文件包含的内容取决于它们的类型和你的仓库状态。`HEAD` 引用可以包含符号链接（指向另一个引用而不是哈希字符串），或是提交的哈希字符串。比如说，看看当你在 `master` 分支上时 `HEAD` 的内容：

```
1. git checkout master
2. cat .git/HEAD
```

这个命令会输出 `ref: refs/heads/master`，也就是说 `HEAD` 指向 `refs/heads/master` 这个引用。这也正是 Git 如何知道现在所在的是 `master` 分支。如果你要切换分支，`HEAD` 的内容将会被更新到新的分支。但如果你要切换到一个提交而不是分支，`HEAD` 会包含一个提交的哈希而不是符号引用。这就是 Git 如何知道现在 `HEAD` 处于分离状态。

在大多数情况下，`HEAD` 是你唯一用得到的引用。其它引用一般只在写底层脚本，接触到 Git 内部的工作机制时才会用到。

refspec

refspec 将本地分支和远程分支对应起来。我们可以通过它用本地的 Git 命令管理远程分支，设置一些高级的 `git push` 和 `git fetch` 行为。

refspec 的定义是这样的：`[+]<src>[:<dst>]`。`<src>` 参数是本地的源分支，`<dst>` 是远程的目标分支。可选的 `+` 号强制远程仓库采用非快速向前的更新策略。

refspec 可以和 `git push` 一起使用，用来指定远程的分支的名称。比如，下面这个命令将 `master` 分支推送到远程 `origin`，就像一般的 `git push` 一样，但它使用 `qa-master` 作为远程仓库中的分支名。对于 QA 团队来说，这个方法非常有用。

```
1. git push origin master:refs/heads/qa-master
```

你也可以用 refspec 来删除远程分支。feature 分支的工作流经常会遇到这种情况，将 feature 分支推送到远程仓库中（比如说为了备份）。你删除本地的 feature 分支之后，远程的 feature 分支依然存在，虽然我们现在已经不再需要它。你可以 push 一个 `<src>` 参数为空的 refspec 来删除它们，就像这样：

```
1. git push origin:some-feature
```

这非常方便，因为你不需要登录到你的远程仓库然后手动删除这些远程分支。注意，在 Git v1.7.0 之后你可以用 `--delete` 标记代替上面这个方法。下面这个命令和上面的命令作用相同：

```
1. git push origin --delete some-feature
```

在 Git 配置文件中增加几行，你就可以更改 `git fetch` 的行为。默认地，`git fetch` 会 fetch 远程仓库中所有分支。原因就是 `.git/config` 文件的这段配置：

```
1. [remote "origin"]
2.     url = https://git@github.com:mary/example-repo.git
3.     fetch = +refs/heads/*:refs/remotes/origin/*
```

fetch 这一行告诉 `git fetch` 从 origin 仓库中下载所有分支。但是，一些工作流不需要所有分支。比如，很多持续集成工作流只关心 master 分支。为了做到这一点，我们需要将 fetch 这行改成下面这样：

```
1. [remote "origin"]
2.     url = https://git@github.com:mary/example-repo.git
3.     fetch = +refs/heads/master:refs/remotes/origin/master
```

你还可以类似地修改 `git push` 的配置。比如，如果你总是将 master 分支推送到 origin 仓库的 qa-master 分支（就像我们之前做的一样），你要把配置文件改成这样：

```
1. [remote "origin"]
2.     url = https://git@github.com:mary/example-repo.git
3.     fetch = +refs/heads/master:refs/remotes/origin/master
4.     push = refs/heads/master:refs/heads/qa-master
```

refspec 给了你完全的掌控权，可以定制 Git 命令如何在仓库之间转移分支。你可以重命名或是删除你的本地分支，fetch 或是 push 不同的分支名，修改 `git push` 和 `git fetch` 的设置，只对你想要的分支进行操作。

相对引用

你还可以通过提交之间的相对关系来引用。`~` 符号让你访问父节点的提交。比如说，下面这个命令显示 `HEAD` 祖父节点的提交：

```
1. git show HEAD~2
```

但是，面对合并提交（merge commit）的时候，事情就会变得有些复杂。因为合并提交有多个父节点，所以你可以找到多条回溯的路径。对于 3 路合并，第一个父节点是你执行合并时的分支，第二个父节点是你传给 `git merge` 命令的分支。

`~` 符号总是选择合并提交的第一个父节点。如果你想选择其他父节点，你需要用 `^` 符号来指定。比如说，`HEAD` 是一个合并提交，下面这个命令返回 `HEAD` 的第二个父节点：

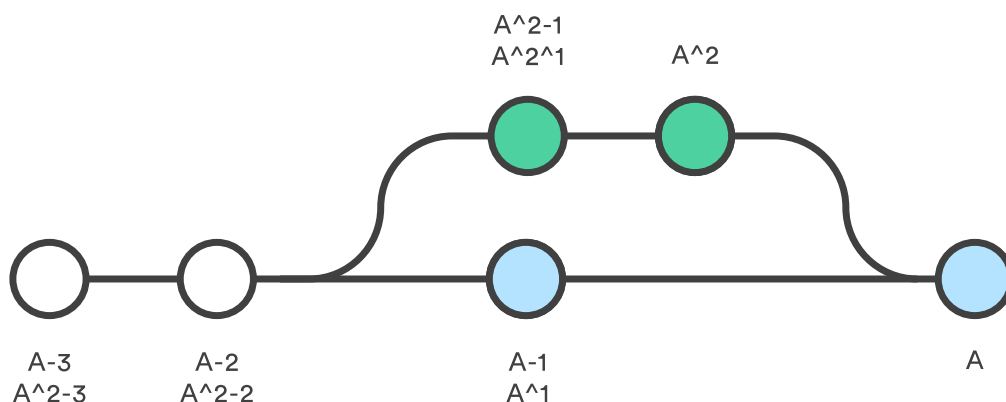
```
1. git show HEAD^2
```

你可以使用不止一个 `^` 来查看超过一层的节点。比如，下面的命令显示的是 `HEAD` 的祖父节点，也就是 `HEAD` 第二个父节点的父节点。

```
1. git show HEAD^2^1
```


为了阐明 `~` 和 `^` 是如何工作的，下面这张图告诉你如何使用相对引用，来指向任意的提交。有的提交可以通过多种方式引用。

Accessing commits using relative refs



相对引用在命令中的用法和普通的引用相同。比如，下面所有命令中使用的都是相对引用：

```
1. # 只列出合并提交的第二个父节点的父节点
2. git log HEAD^2
3.
4. # 移除当前分支最新的 3 个提交
5. git reset HEAD~3
6.
7. # 交互式rebase当前分支最新的 3 个提交
8. git rebase -i HEAD~3
```

引用日志

引用日志是 Git 的安全网。它记录了你在仓库中做的所有更改，不管你有没有提交。你也可以认为这是你本地更改的完整历史记录。运行 `git reflog` 命令查看引用日志。它应该会打印出像下面这样的信息：

```
1. 400e4b7 HEAD@{0}: checkout: moving from master to HEAD~2
2. 0e25143 HEAD@{1}: commit (amend): 将一些很赞的新特性引入`master`
3. 00f5425 HEAD@{2}: commit (merge): 合并'feature'分支
4. ad8621a HEAD@{3}: commit: 结束feature分支开发
```

说人话就是：

- 你刚刚切换到 `HEAD~2`
- 你刚刚修改了一个提交信息
- 你刚刚把 `feature` 分支合并到了 `master` 分支
- 你刚刚提交了一份缓存

`HEAD{<n>}` 语法允许你引用保存在日志中的提交。这和上一节的 `HEAD~<n>` 引用差不多，不过 `<n>` 指的是引用日志中的对象，而不是提交历史。

你可以用办法回到之前可能已经丢失的状态。比如，你刚刚用 `git reset` 方法粉碎了新的 feature 分支。你的引用日志看上去可能会是这样的：

```
1. ad8621a HEAD@{0}: reset: moving to HEAD~3
2. 298eb9f HEAD@{1}: commit: 一些提交信息
3. bbe9012 HEAD@{2}: commit: 继续开发
4. 9cb79fa HEAD@{3}: commit: 开始新特性开发
```

`git reset` 前的三个提交现在都成了悬挂的了，也就是说除了引用日志之外没有办法再引用到它们。现在，假设你意识到了你不应该丢掉你全部的工作。你只需要切换到 `HEAD@{1}` 这个提交就能回到你运行 `git reset` 之前仓库的状态。

```
1. git checkout HEAD@{1}
```

这会让你处于 `HEAD` 分离的状态。你可以从这里开始，创建新的分支，继续你的工作。

总结

你现在对 Git 提交的引用应该已经相当熟悉了。我们知道了分支和标签是如何存在于 `.git` 的子文件夹 `refs` 中，如何读取打包的引用文件，如何使用 `refspec` 来进行更高级的 `push` 和 `fetch` 操作，如何使用 `~` 和 `^` 符号来遍历分支结构。

我们还了解了引用日志，来引用到其他方式已经不存在的提交。这是一种很好的恢复误删提交的方法。

它的意义在于：在任何开发场景下，你都能找到你需要的特定提交。你很容易就可以把这些技巧用在你一有的 Git 知识中，因为很多常用的命令都接受引用作为参数，包括 `git log` 、 `git show` 、 `git checkout` 、 `git reset` 、 `git revert` 、 `git rebase` 等等。

这篇文章是「[git-recipes](#)」的一部分，点击 [目录](#) 查看所有章节。

如果你觉得文章对你有帮助，欢迎点击右上角的 [Star](#) :star2: 或 [Fork](#) :fork_and_knife:。

如果你发现了错误，或是想要加入协作，请参阅 [Wiki](#) [协作说明](#)。