一个成功的 Git 分支模型

作者：[文森特·德里森](https://nvie.com/about/)  
2010 年 1 月 5 日，星期二

**反思记录（2020 年 3 月 5 日）**

这个模型是在 2010 年构想出来的，现在已经是 10 多年前了，也就是 Git 本身出现后不久。在这 10 年中，git-flow（本文中介绍的分支模型）在许多软件团队中变得非常流行，以至于人们开始将其视为某种标准——但不幸的是，它也被视为一种教条或灵丹妙药.

在这 10 年里，Git 本身已经席卷了整个世界，使用 Git 开发的最流行的软件类型正在更多地转向 Web 应用程序——至少在我的过滤器泡沫中是这样。Web 应用程序通常是连续交付的，而不是回滚的，而且您不必支持在野外运行的多个版本的软件。

这不是我 10 年前写博客文章时想到的软件类别。如果您的团队正在持续交付软件，我建议采用更简单的工作流程（例如[GitHub flow](https://guides.github.com/introduction/flow/)），而不是试图将 git-flow 硬塞进您的团队。

但是，如果您正在构建明确版本化的软件，或者如果您需要在野外支持软件的多个版本，那么 git-flow 可能仍然适合您的团队，就像它适合其他人一样过去 10 年。在这种情况下，请继续阅读。

总而言之，永远记住灵丹妙药是不存在的。考虑你自己的背景。不要讨厌。自行决定。

在这篇文章中，我介绍了大约一年前我为我的一些项目（包括工作和私人项目）引入的开发模型，结果证明它非常成功。有一段时间我一直想写这篇文章，但直到现在我才真正找到时间来彻底地写这篇文章。我不会谈论任何项目的细节，只会谈论分支策略和发布管理。



为什么是 git？

有关 Git 与集中式源代码控制系统相比的优缺点的全面讨论，请参阅 [网络](http://git.or.cz/gitwiki/GitSvnComparsion)。那里有很多火焰战争。作为一名开发人员，我更喜欢 Git 胜过当今所有其他工具。Git 确实改变了开发人员对合并和分支的看法。在我来自的经典 CVS/Subversion 世界中，合并/分支一直被认为有点吓人（“当心合并冲突，它们会咬你！”）而且你每隔一段时间才会这样做。

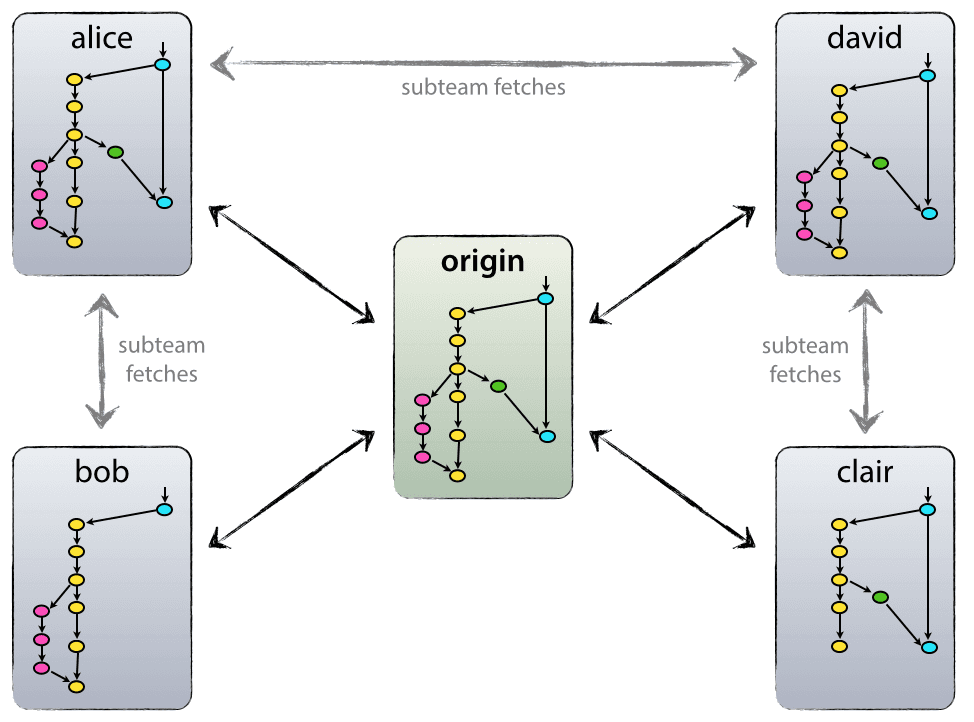
但是使用 Git，这些操作非常便宜和简单，它们被认为是您*日常*工作流程的核心部分之一，真的。例如，在 CVS/Subversion[书籍中](http://svnbook.red-bean.com/)，分支和合并在后面的章节（针对高级用户）中首先讨论，而在 [每本](http://book.git-scm.com/) [Git](http://pragprog.com/titles/tsgit/pragmatic-version-control-using-git) [书籍](http://github.com/progit/progit)中，它已经在第 3 章（基础）中介绍。

由于其简单性和重复性，分支和合并不再是可怕的事情。版本控制工具应该比其他任何东西都更有助于分支/合并。

工具已经讲完了，让我们进入开发模型。我将在这里介绍的模型本质上只不过是每个团队成员必须遵循的一组程序才能进入托管软件开发过程。

去中心化但中心化

我们使用的存储库设置与此分支模型配合得很好，是具有中央“真实”存储库的设置。请注意，这个 repo 只 *被认为* 是中央仓库（因为 Git 是 DVCS，所以在技术层面上没有中央仓库之类的东西）。我们将此 repo 称为origin，因为所有 Git 用户都熟悉这个名称。



每个开发人员都拉并推到原点。但除了中心化的推拉关系，每个开发者还可以从其他同行那里拉取变更，形成子团队。例如，这对于与两个或多个开发人员一起开发一个大的新功能可能很有用，然后才 origin过早地推动正在进行的工作。上图中，有 Alice 和 Bob、Alice 和 David、Clair 和 David 的子团队。

从技术上讲，这意味着 Alice 定义了一个 Git 远程，名为bob，指向 Bob 的存储库，反之亦然。

主要分支



在核心，开发模型受到现有模型的极大启发。中央仓库拥有两个具有无限生命周期的主要分支：

* master
* develop

每个 Git 用户都应该熟悉at的master分支。origin与分支平行master，存在另一个分支，称为develop.

我们认为origin/master是源代码 HEAD始终反映*生产就绪*状态的主要分支。

我们认为origin/develop是主要分支，其中的源代码 HEAD始终反映具有下一个版本的最新交付开发更改的状态。有人将其称为“集成分支”。这是构建任何自动夜间构建的地方。

当develop分支中的源代码达到稳定点并准备发布时，所有更改都应该以某种方式合并回master ，然后用发布号标记。将进一步讨论如何详细完成此操作。

因此，每次将更改合并回 时，*根据定义*master，这都是一个新的生产版本。我们在这方面往往非常严格，因此理论上，我们可以使用 Git 钩子脚本来自动构建我们的软件并将其推出到我们的生产服务器上，每次提交 .master

支持分支

除了主要分支master和develop，我们的开发模型使用各种支持分支来帮助团队成员之间的并行开发，轻松跟踪功能，准备生产版本并帮助快速修复实时生产问题。与主分支不同，这些分支的生命周期总是有限的，因为它们最终会被删除。

我们可能使用的不同类型的分支是：

* 功能分支
* 发布分支
* 修补程序分支

这些分支中的每一个都有特定的目的，并且必须遵守严格的规则，即哪些分支可以是它们的原始分支，哪些分支必须是它们的合并目标。我们将在一分钟内完成它们。

从技术角度来看，这些分支绝不是“特殊的”。分支类型按我们如何*使用*它们进行分类。它们当然是普通的旧 Git 分支。

**特征分支**



可能从以下分支分支：

develop

必须合并回：

develop

分支命名约定：

除了 master, develop, release-\*, 或hotfix-\*

功能分支（或有时称为主题分支）用于为即将发布或遥远的未来版本开发新功能。当开始开发一个特性时，这个特性将被合并到的目标版本可能是未知的。特性分支的本质是，只要特性处于开发阶段，它就存在，但最终会被合并回develop（以明确将新特性添加到即将发布的版本中）或丢弃（以防实验令人失望）。

功能分支通常仅存在于开发人员存储库中，而不存在于origin.

**创建功能分支**

在开始开发新功能时，请从develop分支中分支出来。

$ git checkout -b myfeature develop

切换到新分支“myfeature”

**在开发中合并一个已完成的功能**

完成的功能可以合并到develop分支中，以明确地将它们添加到即将发布的版本中：

$ git checkout develop

切换到分支 'develop'

$ git merge --no-ff myfeature

更新 ea1b82a..05e9557

（更改摘要）

$ git branch -d myfeature

删除分支 myfeature（原为 05e9557）。

$ git push origin 开发

该--no-ff标志使合并始终创建一个新的提交对象，即使可以使用快进执行合并。这避免了丢失有关功能分支的历史存在的信息，并将一起添加该功能的所有提交组合在一起。比较：



在后一种情况下，不可能从 Git 历史记录中看到哪些提交对象一起实现了一项功能——您必须手动读取所有日志消息。在后一种情况下，恢复整个功能（即一组提交）是一个真正令人头疼的问题，而如果使用 --no-ff标志则很容易做到。

是的，它会创建更多（空）提交对象，但收益远大于成本。

**发布分支**

可能从以下分支分支：

develop

必须合并回：

develop和master

分支命名约定：

release-\*

发布分支支持准备新的生产版本。它们允许在最后一分钟点 i 和交叉 t。此外，它们允许修复小错误并为发布准备元数据（版本号、构建日期等）。通过在发布分支上完成所有这些工作，该develop 分支被清除以接收下一个大版本的功能。

分支新发布分支的关键时刻develop是开发（几乎）反映了新发布的期望状态。至少所有针对要构建的版本的功能都必须在 develop此时合并到。针对未来版本的所有功能可能不会——它们必须等到发布分支分支之后。

正是在发布分支的开始，即将发布的版本被分配了一个版本号——而不是更早。直到那一刻，该develop 分支反映了“下一个版本”的更改，但在发布分支启动之前，尚不清楚该“下一个版本”最终会变为 0.3 还是 1.0。该决定是在发布分支开始时做出的，并由项目关于版本号碰撞的规则执行。

**创建发布分支**

发布分支是从develop分支创建的。例如，假设版本 1.1.5 是当前的生产版本，我们即将发布一个大版本。的状态develop已为“下一个版本”做好准备，我们已决定这将成为版本 1.2（而不是 1.1.6 或 2.0）。所以我们分支并给发布分支一个反映新版本号的名称：

$ git checkout -b release-1.2 develop

切换到新分支“release-1.2”

$ ./bump-version.sh 1.2

文件修改成功，版本撞到1.2。

$ git commit -a -m "Bumped version number to 1.2"

[release-1.2 74d9424] Bumped version number to 1.2

1个文件改变，1个插入（+），1个删除（-）

在创建一个新分支并切换到它之后，我们会增加版本号。这 bump-version.sh是一个虚构的 shell 脚本，它更改工作副本中的一些文件以反映新版本。（这当然可以是手动更改——关键是*某些*文件发生了更改。）然后，提交了碰撞的版本号。

这个新分支可能会在那里存在一段时间，直到发布可能确定推出。在此期间，可能会在此分支中应用错误修复（而不是在develop分支上）。严格禁止在此处添加大型新功能。它们必须合并到develop中，因此，等待下一个大版本。

**完成发布分支**

当发布分支的状态准备好成为真正的发布时，需要进行一些动作。首先，将发布分支合并到 （因为*根据定义*master，每次提交master都是一个新发布，请记住）。接下来，必须标记该提交，以便将来参考此历史版本。最后，在发布分支上所做的更改需要合并回，以便将来的版本也包含这些错误修复。masterdevelop

Git 中的前两个步骤：

$ git checkout master

切换到分支'master'

$ git merge --no-ff release-1.2

递归合并。

（更改摘要）

$ git tag -a 1.2

发布现已完成，并标记以供将来参考。

**编辑：**您可能还想使用-sor-u <key>标志以加密方式签署您的标签。

不过，为了保留在发布分支中所做的更改，我们需要将它们合并回develop. 在 Git 中：

$ git checkout develop

切换到分支'develop'

$ git merge --no-ff release-1.2

递归合并。

（变更概要）

这一步很可能会导致合并冲突（甚至可能，因为我们已经更改了版本号）。如果是这样，修复它并提交。

现在我们真的完成了，发布分支可能会被删除，因为我们不再需要它了：

$ git branch -d release-1.2

已删除分支 release-1.2（原为 ff452fe）。

**修补程序分支**



可能从以下分支分支：

master

必须合并回：

develop和master

分支命名约定：

hotfix-\*

修补程序分支与发布分支非常相似，因为它们也旨在为新的生产版本做准备，尽管是计划外的。它们源于对现场制作版本的不良状态立即采取行动的必要性。当必须立即解决生产版本中的严重错误时，可以从标记生产版本的 master 分支上的相应标签中分支出一个 hotfix 分支。

本质是团队成员（在develop分支上）的工作可以继续，而另一个人正在准备快速的生产修复。

**创建修补程序分支**

修补程序分支是从 master分支创建的。例如，假设版本 1.2 是当前正在运行的生产版本，并且由于严重的错误而导致麻烦。但变化develop仍然不稳定。然后我们可以分支出一个修补程序分支并开始修复问题：

$ git checkout -b hotfix-1.2.1 master

切换到新分支“hotfix-1.2.1”

$ ./bump-version.sh 1.2.1

文件修改成功，版本撞到1.2.1。

$ git commit -a -m "Bumped version number to 1.2.1"

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

1 个文件更改，1 个插入（+），1 个删除（-）

分支后不要忘记增加版本号！

然后，修复错误并在一个或多个单独的提交中提交修复。

$ git commit -m "修复了严重的生产问题"

[hotfix-1.2.1 abbe5d6] 修复了严重的生产问题

5 个文件更改，32 个插入（+），17 个删除（-）

**完成一个修补程序分支**

完成后，需要将 bugfix 合并回master，但也需要将 bugfix 合并回develop，以确保该 bugfix 也包含在下一个版本中。这与发布分支的完成方式完全相似。

首先，更新master并标记版本。

$ git checkout master

切换到分支'master'

$ git merge --no-ff hotfix-1.2.1

递归合并。

（更改摘要）

$ git tag -a 1.2.1

**编辑：**您可能还想使用-sor-u <key>标志以加密方式签署您的标签。

接下来，在 中develop也包含错误修复：

$ git checkout develop

切换到分支'develop'

$ git merge --no-ff hotfix-1.2.1

递归合并。

（变更概要）

此处规则的一个例外是， **当当前存在发布分支时，需要将修补程序更改合并到该发布分支中，而不是**develop. 当发布分支完成时，将错误修复反向合并到发布分支最终会导致错误修复也被合并到develop。（如果develop立即工作需要此错误修复并且不能等待发布分支完成，您也可以安全地将错误修复合并到developnow 中。）

最后，删除临时分支：

$ git branch -d hotfix-1.2.1

已删除分支 hotfix-1.2.1（原为 abbe5d6）。

总结

虽然这个分支模型并没有什么真正令人震惊的新东西，但这篇文章开头的“大局”图已经证明在我们的项目中非常有用。它形成了一个易于理解的优雅心智模型，并允许团队成员对分支和发布过程形成共同的理解。

此处提供了该图的高质量 PDF 版本。来吧，把它挂在墙上，以便随时快速参考。

**更新：**对于任何要求它的人：这是主图图像（Apple Keynote）的 [gitflow-model.src.key](http://github.com/downloads/nvie/gitflow/Git-branching-model-src.key.zip)。

[](https://nvie.com/files/Git-branching-model.pdf)  
[Git 分支模型.pdf](https://nvie.com/files/Git-branching-model.pdf)

此博客上的其他帖子

* [**为什么空列表上的 .every() 为真**](https://nvie.com/posts/why-every-on-an-empty-list-is-true/)
* [**日常使用的 Git 电动工具**](https://nvie.com/posts/git-power-tools/)
* [**解码器简介**](https://nvie.com/posts/introducing-decoders/)
* [**漂亮的代码**](https://nvie.com/posts/beautiful-code/)
* [**美丽的地图**](https://nvie.com/posts/beautiful-map/)

如果你想取得联系，我是Twitter 上的[@nvie](http://twitter.com/nvie)。

Vincent Driessen 是来自荷兰的独立软件工程师和顾问。