

Git 工作流程

作者： 阮一峰

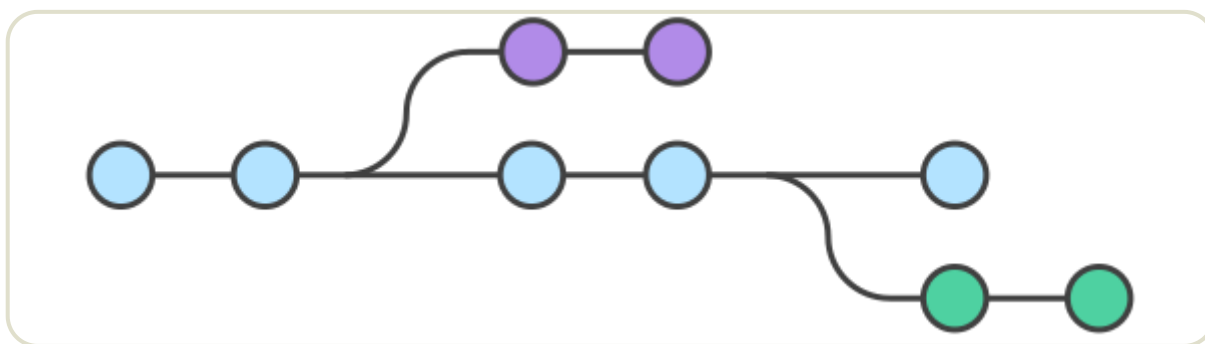
日期： 2015年12月24日



本站由 珠峰培训（专业前端培训）独家赞助

Git 作为一个源码管理系统，不可避免涉及到多人协作。

协作必须有一个规范的工作流程，让大家有效地合作，使得项目井井有条地发展下去。"工作流程"在英语里，叫做"workflow"或者"flow"，原意是水流，比喻项目像水流那样，顺畅、自然地向前流动，不会发生冲击、对撞、甚至漩涡。



本文介绍三种广泛使用的工作流程：

- Git flow
- Github flow
- Gitlab flow

如果你对Git还不是很熟悉，可以先阅读下面的文章。

- [《Git 使用规范流程》](#)
- [《常用 Git 命令清单》](#)
- [《Git 远程操作详解》](#)

一、功能驱动

本文的三种工作流程，有一个共同点：都采用"[功能驱动式开发](#)"（Feature-driven development，简称FDD）。

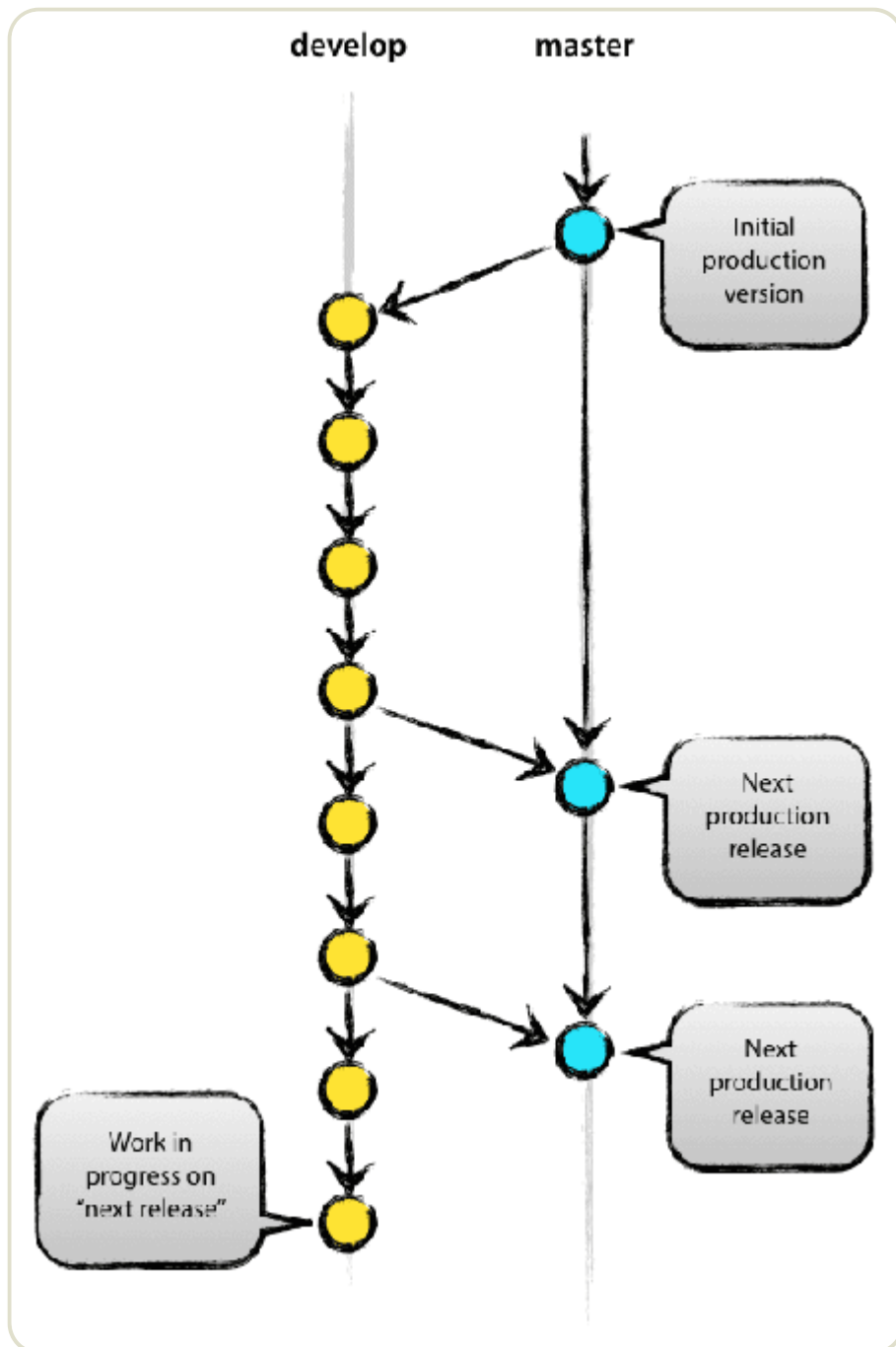
它指的是，需求是开发的起点，先有需求再有功能分支（feature branch）或者补丁分支（hotfix branch）。完成开发后，该分支就合并到主分支，然后被删除。

二、Git flow

最早诞生、并得到广泛采用的一种工作流程，就是[Git flow](#)。

2.1 特点

它最主要的特点有两个。



首先，项目存在两个长期分支。

- 主分支 `master`
- 开发分支 `develop`

前者用于存放对外发布的版本，任何时候在这个分支拿到的，都是稳定的分布版；后者用于日常开发，存放最新的开发版。

其次，项目存在三种短期分支。

- 功能分支 (feature branch)
- 补丁分支 (hotfix branch)

- 预发分支 (release branch)

一旦完成开发，它们就会被合并进 `develop` 或 `master`，然后被删除。

Git flow 的详细介绍，请阅读我翻译的中文版[《Git 分支管理策略》](#)。

2.2 评价

Git flow 的优点是清晰可控，缺点是相对复杂，需要同时维护两个长期分支。大多数工具都将 `master` 当作默认分支，可是开发是在 `develop` 分支进行的，这导致经常要切换分支，非常烦人。

更大问题在于，这个模式是基于"版本发布"的，目标是一段以后时间产出一个新版本。但是，很多网站项目是"持续发布"，代码一有变动，就部署一次。这时，`master` 分支和 `develop` 分支的差别不大，没必要维护两个长期分支。

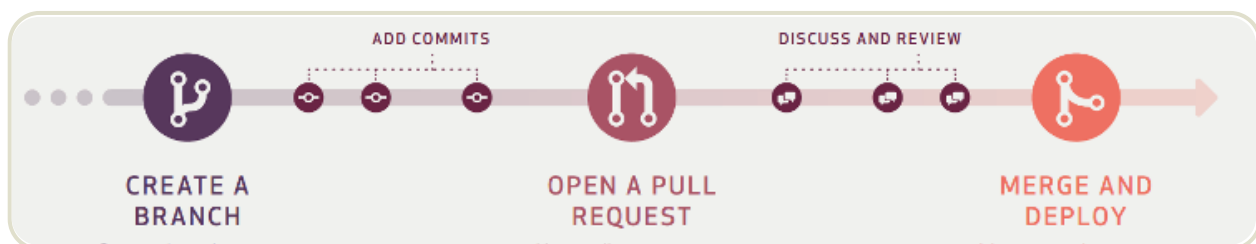
三、Github flow

[Github flow](#) 是 Git flow 的简化版，专门配合"持续发布"。它是 Github.com 使用的工作流程。

3.1 流程

它只有一个长期分支，就是 `master`，因此用起来非常简单。

官方推荐的[流程](#)如下。



第一步：根据需求，从 `master` 拉出新分支，不区分功能分支或补丁分支。

第二步：新分支开发完成后，或者需要讨论的时候，就向 `master` 发起一个 [pull request](#)（简称PR）。

第三步：Pull Request 既是一个通知，让别人注意到你的请求，又是一种对话机制，大家一起评审和讨论你的代码。对话过程中，你还可以不断提交代码。

第四步：你的Pull Request被接受，合并进 `master`，重新部署后，原来你拉出来的那个分支就被删除。（先部署再合并也可。）

3.2 评价

Github flow 的最大优点就是简单，对于"持续发布"的产品，可以说是最合适的流程。

问题在于它的假设：`master` 分支的更新与产品的发布是一致的。也就是说，`master` 分支的最新代码，默认就是当前的线上代码。

可是，有些时候并非如此，代码合并进入 `master` 分支，并不代表它就能立刻发布。比如，苹果商店的APP提交审核以后，等一段时间才能上架。这时，如果还有新的代码提交，`master` 分支就会与刚发布的版本不一致。另一个例子是，有些公司有发布窗口，只有指定时间才能发布，这也会导致线上版本落后于 `master` 分支。

上面这种情况，只有 `master` 一个主分支就不够用了。通常，你不得不在 `master` 分支以外，另外新建一个 `production` 分支跟踪线上版本。

四、Gitlab flow

[Gitlab flow](#) 是 Git flow 与 Github flow 的综合。它吸取了两者的优点，既有适应不同开发环境的弹性，又有单一主分支的简单和便利。它是 Gitlab.com 推荐的做法。

4.1 上游优先

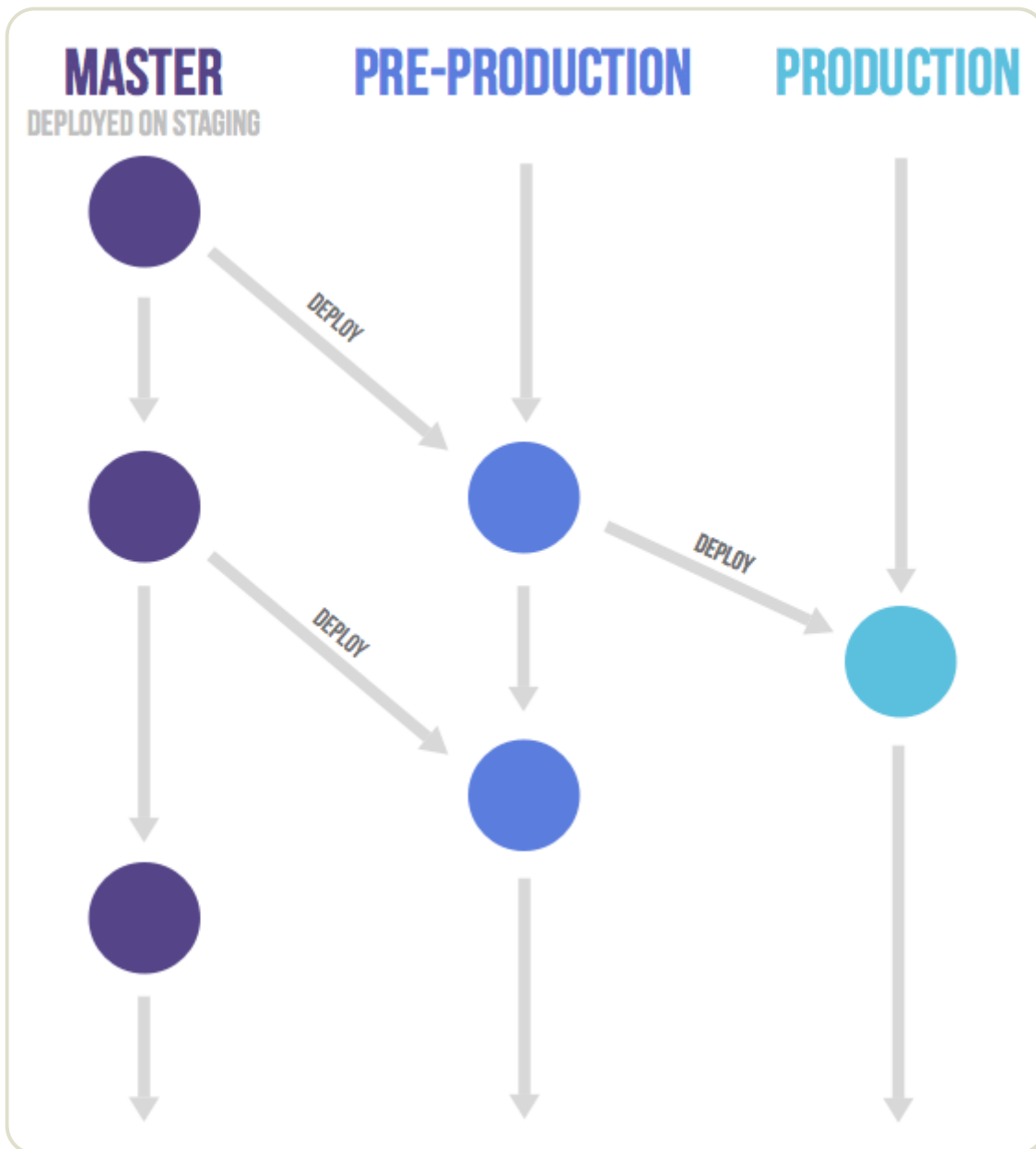
Gitlab flow 的最大原则叫做"上游优先"（upstream first），即只存在一个主分支 `master`，它是所有其他分支的"上游"。只有上游分支采纳的代码变化，才能应用到其他分支。

[Chromium项目](#)就是一个例子，它明确规定，上游分支依次为：

1. Linus Torvalds的分支
2. 子系统（比如netdev）的分支
3. 设备厂商（比如三星）的分支

4.2 持续发布

Gitlab flow 分成两种情况，适应不同的开发流程。

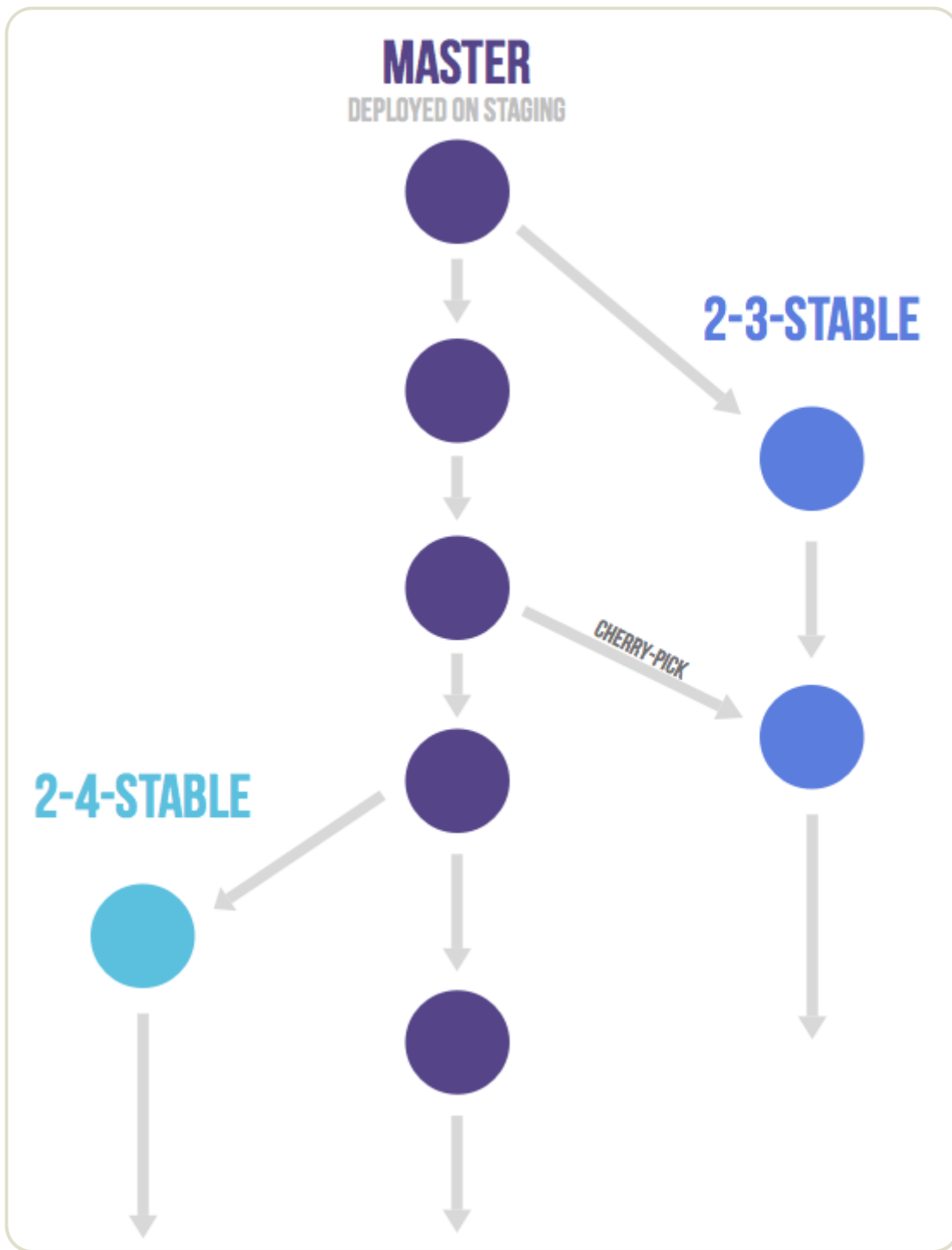


对于"持续发布"的项目，它建议在 `master` 分支以外，再建立不同的环境分支。比如，"开发环境"的分支是 `master`，"预发环境"的分支是 `pre-production`，"生产环境"的分支是 `production`。

开发分支是预发分支的"上游"，预发分支又是生产分支的"上游"。代码的变化，必须由"上游"向"下游"发展。比如，生产环境出现了bug，这时就要新建一个功能分支，先把它合并到 `master`，确认没有问题，再 `cherry-pick` 到 `pre-production`，这一步也没有问题，才进入 `production`。

只有紧急情况，才允许跳过上游，直接合并到下游分支。

4.3 版本发布

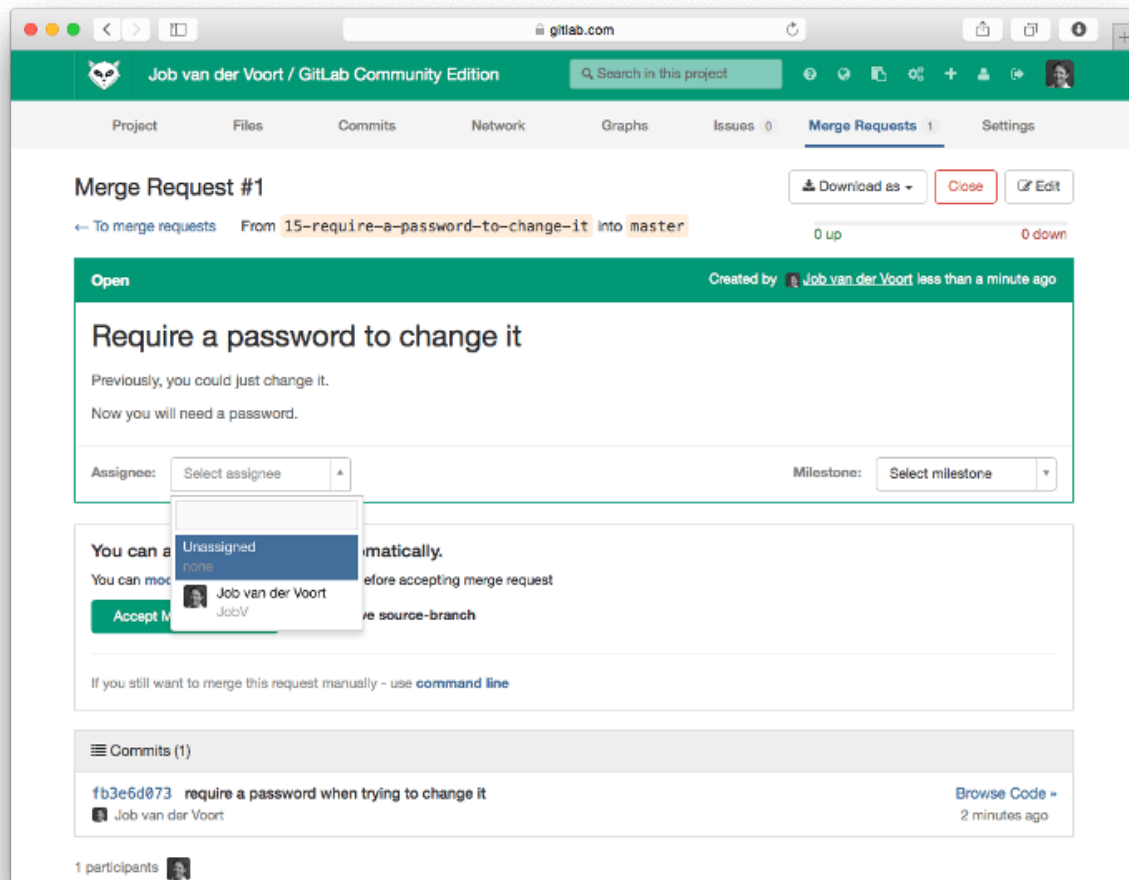


对于"版本发布"的项目，建议的做法是每一个稳定版本，都要从 `master` 分支拉出一个分支，比如 `2-3-stable` 、 `2-4-stable` 等等。

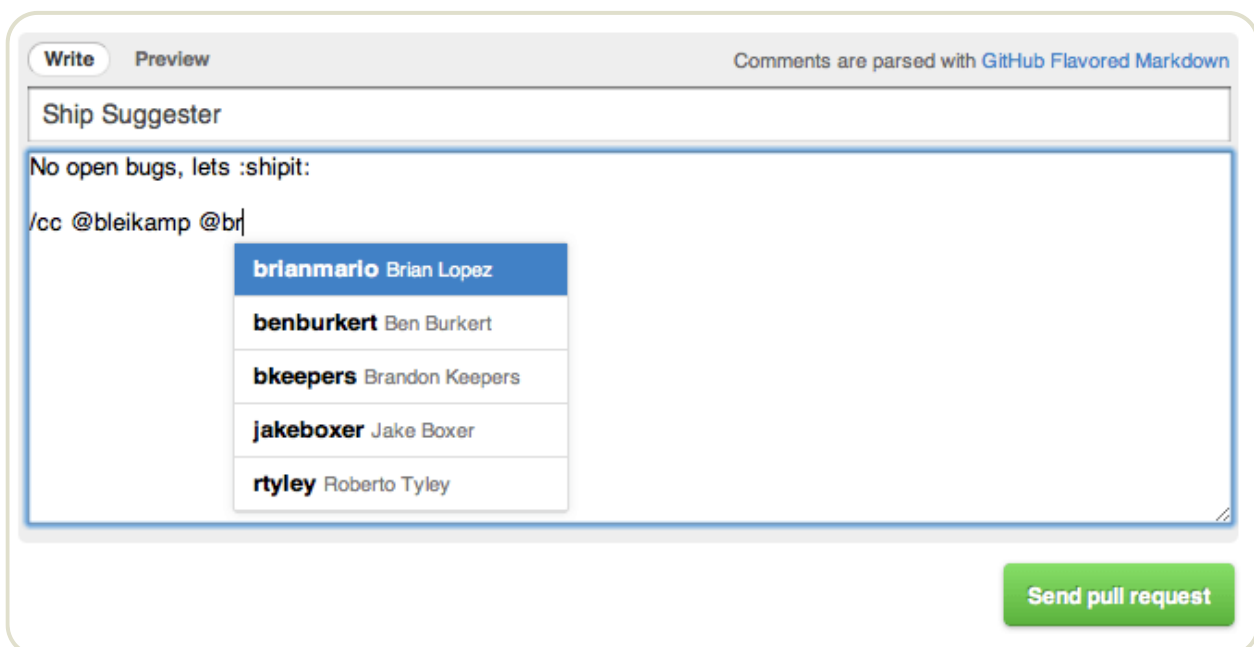
以后，只有修补bug，才允许将代码合并到这些分支，并且此时要更新小版本号。

五、一些小技巧

5.1 Pull Request



功能分支合并进 `master` 分支，必须通过Pull Request（Gitlab里面叫做 Merge Request）。



前面说过，Pull Request本质是一种对话机制，你可以在提交的时候，@ 相关人员或团队，引起他们的注意。

5.2 Protected branch

`master` 分支应该受到保护，不是每个人都可以修改这个分支，以及拥有审批 Pull Request 的权力。

[Github](#) 和 [Gitlab](#) 都提供"保护分支"（Protected branch）这个功能。

5.3 Issue

Issue 用于 Bug追踪和需求管理。建议先新建 Issue，再新建对应的功能分支。功能分支总是为了解决一个或多个 Issue。

功能分支的名称，可以与issue的名字保持一致，并且以issue的编号起首，比如"15-require-a-password-to-change-it"。



开发完成后，在提交说明里面，可以写上"fixes #14"或者"closes #67"。Github规定，只要commit message里面有下面这些[动词](#) + 编号，就会关闭对应的issue。

- close
- closes
- closed
- fix
- fixes
- fixed
- resolve
- resolves
- resolved

这种方式还可以一次关闭多个issue，或者关闭其他代码库的issue，格式是 `username/repository#issue_number` 。

Pull Request被接受以后，issue关闭，原始分支就应该删除。如果以后该issue重新打开，新分支可以复用原来的名字。

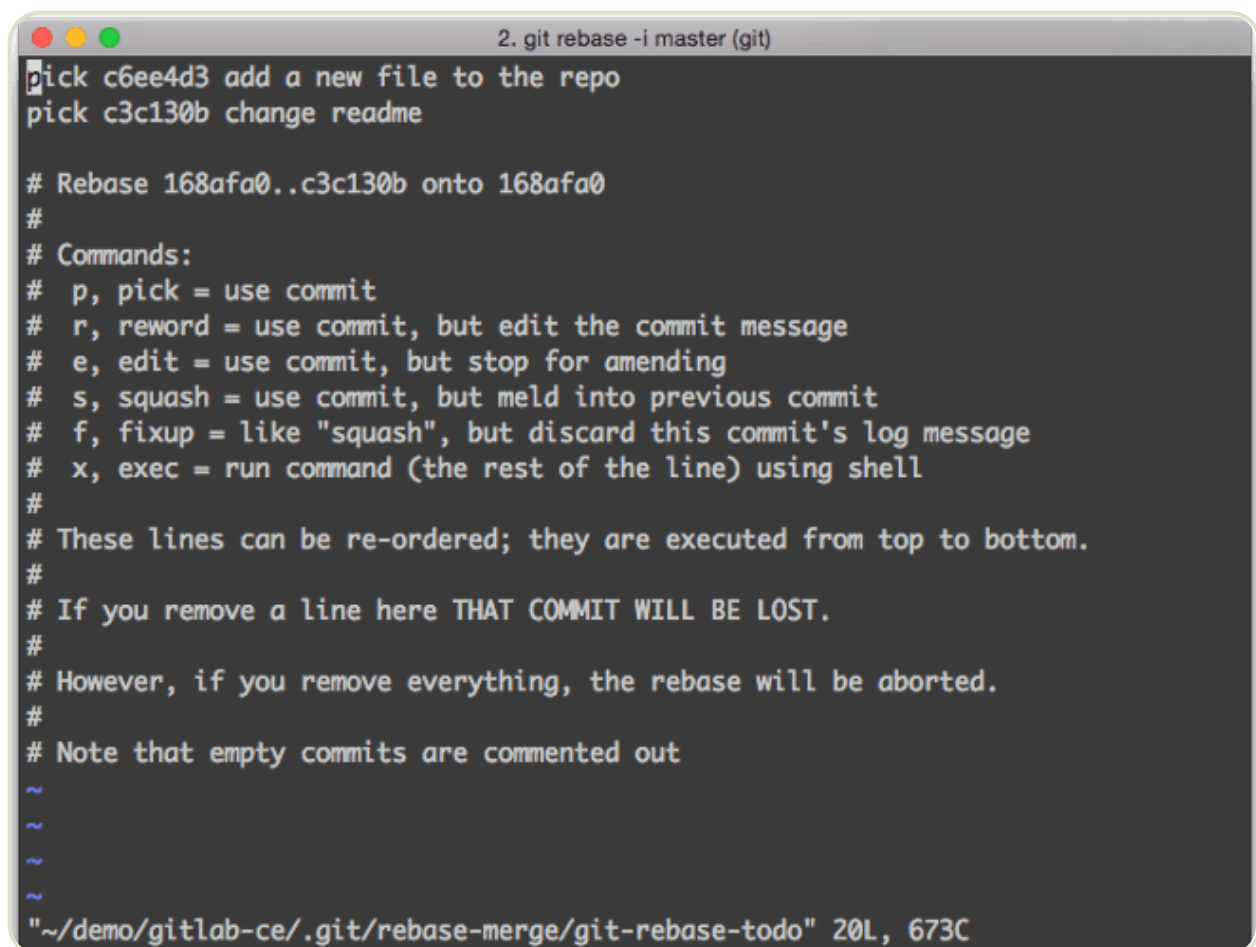
5.4 Merge节点

Git有两种合并：一种是"直进式合并"（fast forward），不生成单独的合并节点；另一种是"非直进式合并"（none fast-forward），会生成单独节点。

前者不利于保持commit信息的清晰，也不利于以后的回滚，建议总是采用后者（即使用 `--no-ff` 参数）。只要发生合并，就要有一个单独的合并节点。

5.5 Squash 多个commit

为了便于他人阅读你的提交，也便于 `cherry-pick` 或撤销代码变化，在发起Pull Request之前，应该把多个commit合并成一个。（前提是，该分支只有你一个人开发，且没有跟 `master` 合并过。）

A screenshot of a terminal window with a dark background and light-colored text. The window title is "2. git rebase -i master (git)". The terminal output shows the interactive rebase process. It starts with "pick c6ee4d3 add a new file to the repo" and "pick c3c130b change readme". Then it shows "# Rebase 168afa0..c3c130b onto 168afa0". Below that is a list of commands: "p, pick = use commit", "r, reword = use commit, but edit the commit message", "e, edit = use commit, but stop for amending", "s, squash = use commit, but meld into previous commit", "f, fixup = like 'squash', but discard this commit's log message", and "x, exec = run command (the rest of the line) using shell". It also includes instructions like "These lines can be re-ordered; they are executed from top to bottom." and "If you remove a line here THAT COMMIT WILL BE LOST.". At the bottom, it shows the file path and line/character count: "~/demo/gitlab-ce/.git/rebase-merge/git-rebase-todo" 20L, 673C.

```
2. git rebase -i master (git)
pick c6ee4d3 add a new file to the repo
pick c3c130b change readme

# Rebase 168afa0..c3c130b onto 168afa0
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~
~
~
~
~/demo/gitlab-ce/.git/rebase-merge/git-rebase-todo" 20L, 673C
```

这可以采用 `rebase` 命令附带的 `squash` 操作，具体方法请参考我写的[《Git 使用规范流程》](http://www.ruanyifeng.com/blog/2015/12/git-workflow.html)。

（完）