07 | 分支管理：Facebook的策略，适合我的团队吗?

2019-09-06 葛俊

研发效率破局之道 进入课程 >

讲述：葛俊
时长 20:27　大小 18.73M

你好，我是葛俊。今天，我来跟你聊聊研发过程中的 Git 代码分支管理和发布策略。

在前面两篇文章中，我们讨论了持续开发、持续集成和持续部署的整个上线流程。这条流水线针对的是分支，因此代码的分支管理便是基础。能否找到适合自己团队的分支管理策略，就是决定代码质量，以及发布顺畅的一个重要因素。

Facebook 有几千名开发人员同时工作在一个大代码仓，每天会有一两千个代码提交入仓，但仍能顺利地进行开发，并发布高质量的产品。平心而论，Facebook 的工程水平的确很高，与他们的分支管理息息相关。

所以在今天这篇文章中，我会先与你详细介绍 Facebook 的分支管理策略，以及背后的原因；然后，与你介绍其他的常见分支管理策略；最后，向你推荐如何选择适合自己的分支策

略。

Facebook 的分支管理和发布策略

Facebook 的分支管理策略，是一种基于主干的开发方式，也叫作 Trunk-based。在这种方式中，用于开发的长期分支只有一个，而用于发布的分支可以有多个。

首先，我们先看看这个长期存在的开发分支。

开发分支

这个长期存在的开发分支，一般被叫作 trunk 或者 master。为方便讨论，我们统一称它为 master。也就是说，所有的开发人员基于 master 分支进行开发，提交也直接 push 到这个分支上。

在主干开发方式下，根据是否允许存在短期的功能分支（Feature Branch），又分为两个子类别：主干开发有功能分支和主干开发无功能分支。Facebook 做得比较纯粹，在主代码仓中，基本上禁止功能分支。

另外，在代码合并回 master 的时候，又有 rebase 和 merge 两种选择。Facebook 选择的是 rebase（关于这样选择的原因，我会在后面与你详细介绍）。所以，Facebook 的整个开发模式非常简单，步骤大概如下。

第一步，获取最新代码。

复制代码

```
git checkout master
git fetch
git rebase origin/master
```

第二步，本地开发，然后执行

复制代码

```
git add
git commit
```

产生本地提交。

第三步，推送到主代码仓的 master 分支。

复制代码

```
git fetch
git rebase origin/master
git push
```

在 rebase 的时候，如果有冲突就先解决冲突，然后使用

复制代码

```
git add
git commit
```

更新自己的提交，最后重复步骤 3，也就是重新尝试推送代码到主代码仓。

看到这里，你可能对这种简单的分支方式有以下两个问题。

问题 1：如果功能比较大，一个代码提交不合适，怎么办？

解决办法：这种情况下，第二步本地开发的时候可以产生多个提交，最后在第三步一次性推送到主仓的 master 分支。

问题 2：如果需要多人协同一个较大的功能，怎么办？

解决办法：这种情况下，Facebook 采用的是使用代码原子性、功能开关、API 版本等方法，让开发人员把功能拆小尽快合并到 master 分支。

比如，一个后端开发者和一个前端开发者合作一个功能，他们的互动涉及 10 个 API 接口，其中两个是在已有接口上做改动，另外 8 个是新增接口。

这两名开发者的合作方式是：

第一，后端开发者把这 10 个接口的编码工作，以至少 10 个单独的提交来完成。强调“至少”，是因为有些接口的编码工作可能比较大，需要不止一个提交来完成。

第二，对已有 API 的改动，如果只涉及增加 API 参数，情况就比较简单，只需要在现有 API 上进行。但如果牵涉到删除或者修改 API 参数，就要给这个 API 添加一个新版本，避免被旧版本阻塞入库。

第三，在实现功能的过程中，如果某个功能暂时还不能暴露给用户，就用功能开关把它关闭。

这就保证了，在不使用功能分支的情况下，这两个开发者可以直接在 master 分支上合作，并能够不被阻塞地尽快提交代码。当然了，这种合作方式，可以扩展到更多的开发者。

以上就是开发分支的情况。接下来，我再与你讲述发布分支和策略。

发布分支

基于主干开发模式中，在需要发布的时候会从 master 拉出一条发布分支，进行测试、稳定。在发布分支发现问题后，先在 master 上修复，然后 cherry-pick 到发布分支上。分支上线之后，如果需要长期存在，比如产品线性质的产品，就保留。如果不需要长期存在，比如 SaaS 产品，就直接删除。Facebook 采用的方式是后者。

具体来说，部署包括 3 种：有每周一次的全量代码部署、每天两次的日部署，以及每天不定次数的热修复部署。日部署和热修复部署类似，我们下面详细介绍周部署和热修复部署。

每次**周部署**代码的时候，流程如下所示。

第一步，从 master 上拉出一个发布分支。

复制代码

```
1 git checkout -b release-date-* origin/master
```

第二步，在发布分支进行各种验证。

第三步，如果验证发现问题，开发者提交代码到 master，然后用 cherry-pick 命令把修复合并到发布分支上：

复制代码

```
git cherry-pick <fix-sha1> # fix-sha1 是修复提交的 commit ID
```

接着继续回到第二步验证。

验证通过就发布当前分支。这个发布分支就成为当前生产线上运行版本对应的分支，我们称之为当前生产分支，同时将上一次发布时使用的生产分支存档或者删除。

在进行**热修复部署**时，从当前生产分支中拉出一个热修复分支，进行验证和修复。具体步骤为：

第一步，拉出一个热修复分支。

复制代码

```
git checkout -b hotfix-date-* release-date-*
```

第二步，开发人员提交热修复到 master，然后 cherry-pick 修复提交到热修复分支上。

复制代码

```
git cherry-pick <fix-sha1>
```

第三步，进行各种验证。

第四步，验证中发现问题，回到第二步重新修复验证。验证通过就发布当前热修复分支，同时将这个热修复分支设置为当前的生产分支，后面如果有新的热修复，就从这个分支拉取。

这里有一张图片，描述了每周五拉取周部署分支，以及从周部署分支上拉取分支进行热修复部署的流程。

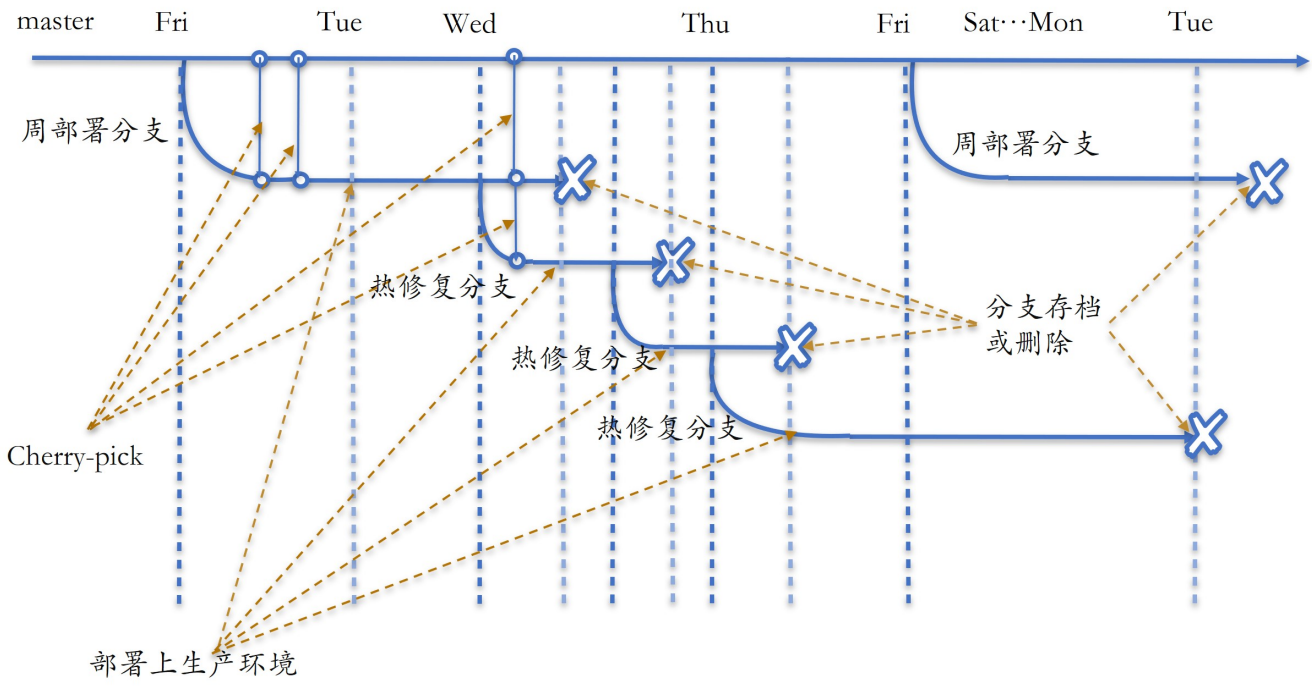

图 1 Facebook 的代码分支管理和部署流程

以上就是 Facebook 的代码分支管理和部署流程。

需要注意的是，这里描述的部署流程是 Facebook 转到持续部署之前采用的。但考虑到非常多的公司还没有达到持续部署的成熟度，所以这种持续交付的方式，对我们更有参考价值。

Facebook 分支管理策略的背后原因

Facebook 采用主干分支模式，最大的好处是可以把持续集成、持续交付做到极致，从而尽量提高 master 分支的代码质量。

解释这一好处之前，我想请你先看看下面这 3 个措施有什么共同效果：

几千名开发者同时工作在同一条主干；

不使用功能分支，直接在 master 上开发；

必须要使用 rebase 才能入库，不能使用 merge。

其实，它们的共同效果就是：必须尽早将代码合入 master 分支，否则就需要花费相当长的时间去解决合并冲突。所以每个开发人员，都会尽量把代码进行原子性拆分，写好一部分就

赶快合并入库。

我曾经有过一个有趣的经历。一天下午，我和旁边的同事在改动同一个 API 接口，实现两个不同的功能。我们关系很好，也都清楚对方在做什么，于是一边开玩笑一边像在比赛一样，看谁先写好代码完成自测入主库。结果是我赢了，他后来花了十分钟很小心地去解决冲突。

Facebook 使用主干分支模式的好处，主要可以总结为以下两点：

> 能够**促进开发人员把代码频繁入主仓进行集成检验**。而这，正是持续集成的精髓。与之相对应的是，很多 20 名开发者的小团队，采用的也是共主干开发方式，但使用了功能分支，让大家在功能分支上进行开发，之后再 merge 回主仓。结果是，大家常常拖到产品上线前才把功能分支合并回主干，导致最后关头出现大量问题。
>
> 能够**确保线性的代码提交历史**，给流程自动化提供最大方便。不要小看“线性”，它对自动化定位问题意义非凡，使得我们可以从当前有问题的提交回溯，找到历史上第一个有问题的提交。更棒的是，我们还可以使用折半查找（也叫作二分查找）的办法，用 O(LogN) 的时间找到那个有问题的提交。

比如，在一个代码仓中，有 C000 ~ C120 的线性提交历史。我们知道一个测试在提交 C100 处都是通过的，但是在 C120 出了问题。我们可以依次 checkout C101、C102，直到 C120，每次 checkout 之后运行测试，总能找到第一个让测试失败的提交。

或者更进一步，我们可以先尝试 C100 和 C120 中间的提交 C110。如果测试在那里通过了，证明问题提交在 C111 和 C120 之间，继续检查 C115；否则就证明问题提交在 C101 和 C110 之间，继续检查 C105。这就大大减少了检查次数。而这，正是软件算法中经典的折半查找。

事实上，Git 本身就提供了一个命令 git bisect 支持折半查找。比如，在刚才的例子中，如果运行测试的命令行是 runtest.sh。那么，我们可以使用下面的命令来自动化这个定位流程：

复制代码

```
> git checkout master # 使用最新提交的代码
> git bisect start
> git bisect bad HEAD # 告知 git bisect，当前 commit 是有问题的提交
> git bisect good C100 # 告知 git bisect，C100 是没有问题的提交
```

```
> git bisect run bash runtest.sh # 开始运行自动化折半查找
...
Cxxx is the first bad commit  # 查找到第一个问题提交
...
bisect run success
> git bisect reset # 结束 git bisect。回到初始的 HEAD
```

很方便吧。而如果历史不是线性的，也就是说如果提交使用了 merge，那么我们就不能方便地定位出第一个问题提交了，更别说是折半查找了。

这种快速定位问题的能力，可以给 CI/CD 带来巨大好处。在持续交付过程中，我们常常没有足够的资源对每一个提交都进行检查。比如前面提过，Facebook 的持续交付流水线就是每隔一段时间，对代码仓最后一个提交运行流水线的检查。如果发现问题，就可以通过上面这种方法自动化地找到问题提交，并自动产生 Bug 工单，分配给提交者。

其他主要分支方式

除了主干开发的分支管理策略，还有 3 种常用方式：

> Git-flow工作流；
>
> Fork-merge 工作流；
>
> 灵活的功能分支组合成发布分支。

我在文中给出了链接供你参考。接下来，我们具体看看这几种方式。

Git-flow 工作流

Git-flow 工作流有两个长期分支：一个是 master，包含可以部署到生产环境的代码；另一个是 develop，是一个用来集成代码的分支，开发新功能、新发布，都从 develop 里拉分支。此外，它还有 3 种短期分支，分别是新功能分支、发布分支、热修复分支，根据需要创建，当完成了自己的任务后就会被删除。

Git-flow 工作流的特点是规定很清晰，对各种开发任务都有明确的规定和步骤。比如：

> 开发新功能时，从 develop 分支拉出一个前缀为 feature- 的新功能分支，在本地开发，并推送到远端中心仓库，完成之后合并入 develop 分支，并删除该功能分支。

发布新版本时，从 develop 分支拉出一个前缀为 release- 的发布分支，部署到测试、类生产等环境进行验证。发现问题后直接在发布分支上修复，测试通过之后，把 release 分支合并到 master 和 develop 分支。在 master 分支上打 tag，并删除该发布分支。

这种工作流，在前几年非常流行。它的好处是流程清晰，但缺点是：

流程复杂，学习成本高。

容易出错，容易出现忘记合并到某个分支的情况。不过可以使用脚本自动化来解决。

不方便进行持续集成。

有太多的代码分支合并，解决冲突成本比较高。

Fork-merge

Fork-merge 是在 GitHub、GitLab 流行之后产生的，具体做法是：每个开发人员在代码仓服务器上有一个“个人”代码仓。这个“个人”代码仓实际上就是主代码仓的一个 clone。开发者对主代码仓贡献代码的步骤如下：

1. 开发者产生一个主代码仓的 fork，成为自己的个人代码仓；
2. 开发者在本地 clone 个人代码仓；
3. 开发者在本地开发，并把代码推送到自己的个人代码仓；
4. 开发者通过 web 界面，向主代码仓作者提出 Pull request；
5. 主代码仓的管理者在自己的开发机器上，取得开发者的提交，验证通过之后再推送到主代码仓。

看起来步骤繁琐，但实际上和主干开发方式很相似，也有一个长期的开发分支，就是主仓的 master 分支。不同之处在于，它提供了一种对主分支更严格、更方便的权限管理方式，即只有主仓管理者有权限推送代码。同时，主仓不需要有功能分支，功能分支可以存在 fork 仓中。所以，主仓干净便于管理。

这种方式对开源项目比较方便，但缺点是步骤繁琐，不太适用于公司内部。

灵活的功能分支组合成发布分支

除了上述方式之外，还有一种非常灵活，但对工具自动化要求很高的分支方式，即基于功能分支灵活产生发布分支的方式。这种方式的典型代表是阿里云效的“分支模式”。

具体方法是大量使用工具对分支的管理进行自动化，开发人员在 web 界面上自助产生针对功能的分支。编码完成后，通过 web 界面对分支组合、验证，并上线，上线之后分支再自动合入主库。

这种方式的好处是：

方便基于功能进行开发。也就是说，开发者可以针对每个功能产生一个分支进行开发。

灵活，也就是能够方便地对功能进行组合，发布到对应环境上测试。出了问题，可以方便地添加或者删除功能。

但这种方式的问题是，对工具的依赖比较高，没有一个系统的工具来自动化的话，基本做不起来。另外，这种方式会大量封装底层的实现，使开发人员不知道底层发生的问题，一旦出现问题就不太容易解决。

哪一种分支管理策略更适合我的团队呢?

要找到适合自己团队的分支管理策略，我们先来对比下上面提到的几种方式的优缺点吧。

方式	优点	缺点
主干分支，无功能分支	简单，可以进行极致的CI	对团队成熟度要求比较高； 需要功能开关，提交原子化
主干分支，有功能分支	简单，小团队容易上手	容易出现集成时间推迟
Git-flow	流程严谨，适合较大规模的团队	流程复杂易出错，合并费时
Fork-merge	主仓干净，权限好管理	操作复杂，不适合团队内部
功能分支组合成发布分支	灵活，分支对应需求	对工具依赖大，抽象太深会让开发者距离底层太远

图 2 几种常用的代码分支管理策略对比

另外，要找到合适的代码分支管理策略，你还可以参考以下 3 个问题，根据答案帮助你进行选择。

问题 1：如果提供功能分支让成员共享，在哪里建立这个分支?

> 如果团队不大，可以允许在主仓创建功能分支，不过注意定时删除不用的分支，避免影响 Git 的性能。如果团队比较大，可以考虑使用 Fork-merge 方式，在上面提到的“个人代码仓”里创建功能分支，从而避免污染主仓。

问题 2：要不要使用 Merge Commit?

> 代码在合并到主干的时候，可以选择 rebase 或者 merge。使用 rebase 的好处是上边提到的方便定位问题。而使用 merge 的好处是，可以清晰地在分支里看到一个功能的所有提交，不像在 rebase 中，一个功能的提交往往是分散的。

问题 3：团队成熟度如何?

> 单分支开发集成早，质量比较好，但对团队成员和流程自动化要求高。所以，如果你的团队比较小，或者比较成熟的话，可以考虑使用单分支，否则可以选择多分支开发模式，但要想办法把集成提前，同时逐步向单主干分支过渡。

总结来说，尽量减少长期分支的数量，代码尽早合并回主仓，方便使用 CI/CD 等方法保证主仓代码提交的质量，是选择分支策略的基本出发点。

小结

首先，我分享了 Facebook 使用的单主干开发分支，以及通过临时发布分支进行部署的分支管理策略和部署方式。然后，我与你介绍了几种常见的分支管理策略，并给出了推荐的选择方法。

在 Facebook 工作时，我们一直使用这种主干分支开发方式。它强迫我们把代码进行原子化，尽量确保每一个提交都尽快合入 master，并保证代码质量。一开始我不是很习惯，但习惯后我发现它的确很棒。

首先，因为你和你的合作开发者都需要尽快把代码拆小、入仓，这就帮助我们提高了功能模块化的能力。其次，因为 master 里面的提交一般都比较健康，并且是比较新的代码，所以很少会被不稳定的因素阻塞。最后，线性提交历史对开发者的日常工作也很有帮助。我们在

开发的时候，常常会碰到一个本来工作得好好的 API，在拉取到最新代码之后出现了问题。这时，我就可以使用这种方法找到第一个造成问题的提交，从而方便定位和解决问题。

一个流程设计、实施得好，对产品来说可以提高质量，对团队来说可以提高效能，对个人来说可以帮助成长。这就是一举三得。

思考题

1. 产品线性质的产品开发，以及 SaaS 产品开发，在选择分支管理策略时有不同的考量。你觉得哪种分支管理方式更适合二者呢？
2. 你知道 trunk-based 里面“trunk”的意思吗？

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎你把这篇文章分享给更多的朋友一起阅读。我们下期再见！

上一篇　06 | 代码入库到产品上线：Facebook如何使用CI/CD满足业务要求？

下一篇　08 | DevOps、SRE的共性：应用全栈思路打通开发和运维

精选留言 (5)

写留言

phobal
2019-09-07

像 Facebook 这种主干（master）开发模式，每个人的代码每天都会合入到 master，实际中经常会有这样的情况：A、B 两个 feature 同时在进行，但上线时间不同，A 先上，那基于这种模式，A 上的话就会带入 B 的代码，难道 A 上的时候再基于上一个稳定版本 tag 再 cheery-pick A 的所有功能代码到测试分支?

展开

作者回复: 如果B的功能还没有准备好给用户使用，可以使用功能开关隐藏。

1

XUN
2019-09-07

老师能不能讲讲移动端的开发管理，看了几篇都是后端的集成和发布流程

作者回复: 前边描述的各种原则，在移动端后端都是一致的，比如说持续开发，持续集成，持续交付等。

就拿在iOS的Facebook APP 的开发做例子。首先他们也是采用单主干的开发分支模式。要求代码提交的原子性，以及master分支上代码的线性历史。在持续集成方面。他们也是使用的Phabricator作为历程和质量控制中心，进行各种各样的代码入库前检查。在持续交付方面。他们也是采用了和后端类似的方式，每隔一定时间进行一次全量的构建和验证。

当然，也会有一些差别。我在后面的文章中会找机会详细描述。

感谢你的提问!

囧囧冰淇淋
2019-09-07

看到今天，感觉研发和代码越来越有趣了XD

展开

作者回复: 对大家有用，让大家觉研发有趣，感觉写文章越来越有趣了XD

Johnson
2019-09-06

听起来trunk-base这种方式基本和集中式的svn和perforce是一个思路，那疑问就是为啥不直接用perforce,收费？学习成本？需要额外的运维投入？
trunk-base针对pull request的方式还有一个疑问，怎么做pre checkin的code review来控制入库代码的质量？
我能想到的好像还是集中式的那种checkin必须使用固定的工具，这个工具会去自动获取...
展开

> 作者回复: > 听起来trunk-base这种方式基本和集中式的svn和perforce是一个思路，那疑问就是为啥不直接用perforce,收费？学习成本？需要额外的运维投入？
>
> 非常好的问题。我觉得至少有以下几个好处：
> 1. 分布式的好处，使得开发人员在本地操作速度快。
> 2. 分布式的另一个好处。虽然我在文章中没有写，使用的频率也不是很高，但是开发人员的个人repo是可以互相推送代码进行合作的，不经过统一的中央repo。
> 3. Git开源。事实上，2015年左右，Facebook每天有太多的commits，所以Git的性能逐渐不能满足。于是Facebook尝试修改Git的代码。不过因为Git社区觉得Git代码仓不应该超级大，所以没有合作成功。最终Facebook从Git转向了Mercurial（hg）。ht是另一个类似Git的分布式代码仓管理系统。hg也是开源，Facebook通过提高hg解决的commits太多的问题。这就是使用开源的好处。这里强调一下，虽然Facebook从Git转到了hg，但是今天文章中讨论的分支策略对hg也是有效的。
> 4. Git免费
> 5. 在进行周部署，日部署、热修复部署中各种分支处理，Git的比svn，perforce强大。这一点因为我对Perfoce不是特别熟悉，不是100%确定。
>
> > trunk-base针对pull request的方式还有一个疑问，怎么做pre checkin的code review来控制入库代码的质量？
> 我能想到的好像还是集中式的那种checkin必须使用固定的工具，这个工具会去自动获取这个提交的code review，如果没有就不允许checkin.
>
> 这个的确是一种方式。比如使用Phabricator进行Code Review的时候，Phabricator对外提供接口。在开发者尝试push代码到Git Server的时候，Git Server可以会去Phabricator检查用户的commit有没有通过Code Review。
>
> 另外一个方式是让代码审查工具，比如Phabricator，Gerrit，在Code Review通过之后，代替开发者Push commit。

2

李双

2019-09-06

各种分支管理策略，学习！问下，基于base开发的这种方式，是不是根据时间线，截止某一时间点（或者某个版本号之前的），代码验证过了，就可以上线了，而不是根据业务功能的先后紧急！

> 作者回复: 对的。这种方式是发布周期与功能解耦。版本火车一列一列发出去。功能开发者自己决定把commit搭乘哪一列。办不上的话，没办法，等下一列。