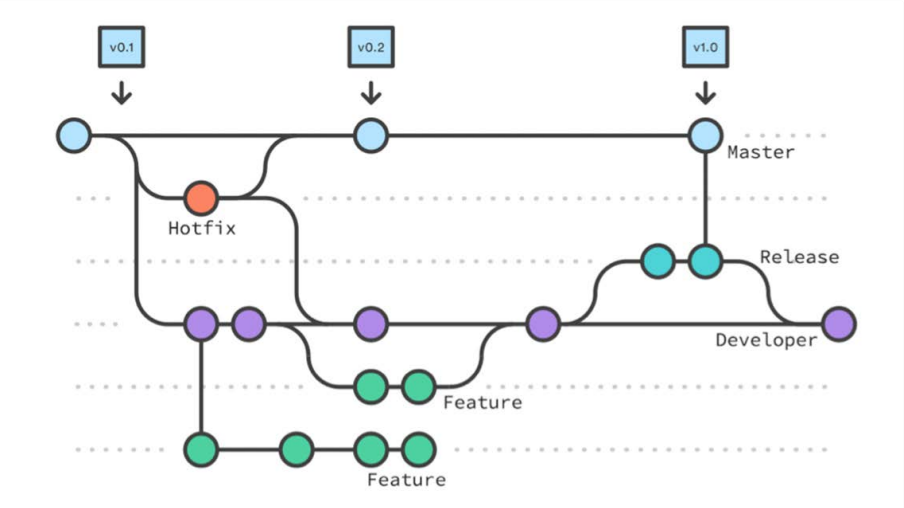


Git协同 workflow，你该怎样选

2018-01-16 陈皓



与传统的代码版本管理工具相比，Git 有很多的优势，因而越来越成为程序员喜欢的版本管理工具。Git 这个代码版本管理工具最大的优势有以下几个。

- Git 是一个分布式的版本管理工具，而且可以是单机版的，所以，你在没有网络的时候同样可以提交（commit）代码。对于我们来说，这意味着在出差途中或是没有网络的环境中依然可以工作写代码。

这是不是听起来有点不对？一方面，以后也不能以将没有网络作为不工作的借口了。另一方面，没有网络意味着没有 Google 和 StackOverflow，光有个本地的 Git 我也一样不能写代码啊……

（哈哈，好吧，这已经超出了 Git 这个技术的范畴了，这里就不讨论了。）

- Git 从一个分支向另一个分支合并代码的时候，会把要合并的分支上的所有提交一个应用到被合并的分支上，合并后也能看到整个代码的变更记录。而其他的版本管理工具则不能。
- Git 切换分支的时候通常很快。不像其他版本管理器，每个分支一份拷贝。
- Git 有很多非常有用的命令，让你可以很方便地工作。

比如我很喜欢的 `git stash` 命令，可以把当前没有完成的事先暂存一下，然后去忙别的事。`git cherry-pick` 命令可以让你有选择地合并提交。`git add -p` 可以让你挑选改动提交，`git grep $regex $ (git rev-list --all)` 可以用来在所有的提交中找代码。因为都是本地操作，所以你会觉得飞快。

除此之外，由 Git 衍生出来的 GitHub/GitLab 可以帮你很好地管理编程工作，比如 `wiki`、`fork`、`pull request`、`issue`……集成了与编程相关的工作，让人觉得这不是一个冷冰冰的工具，而真正和我们的日常工作发生了很好的交互。

GitHub/GitLab 这样工具的出现，让我们的工作可以呈现在一个工作平台上，并以此来规范整个团队的工作，这才正是 Git 这个版本管理工具成功的原因。

今天，我们不讲 Git 是怎么用的，因为互联网上有太多的文章和书了。而且，如果你还不会用 Git 的话，那么你已经严重落后于这个时代了。在这篇文章中，我想讲一下 Git 的协同 workflow，因为我看到很多团队在使用 Git 时，并没有用好。

注意，因为 Git 是一个分布式的代码管理器，所以，是分布式就会出现数据不一致的情况，因此，我们需要一个协同 workflow 来让工作变得高效，并可以有效地让代码具有更好的一致性。

说到一致性，就是每个人手里的开发代码，还有测试和生产线上的代码，要有一个比较好的一致性的管理和协同方法。这就是 Git 协同 workflow 所需要解决的问题。

目前来说，你可能以为我想说的是 GitFlow 工作流。恭喜你猜对了。但是，我想说的是，GitFlow 工作流太过复杂，我并不觉得 GitFlow 工作流是一个好的工作流。如果你的团队在用这种 workflow 开发软件，我相信你的感觉一定是糟透了。

所以，我的这篇文章会对比一些比较主流的协同 workflow，然后，再抨击一下 GitFlow 工作流。

中心式协同 workflow

首先，我们先说明一下，Git 是可以像 SVN 这样的中心工作流一样工作的。我相信很多程序员都是在采用这样的工作方式。

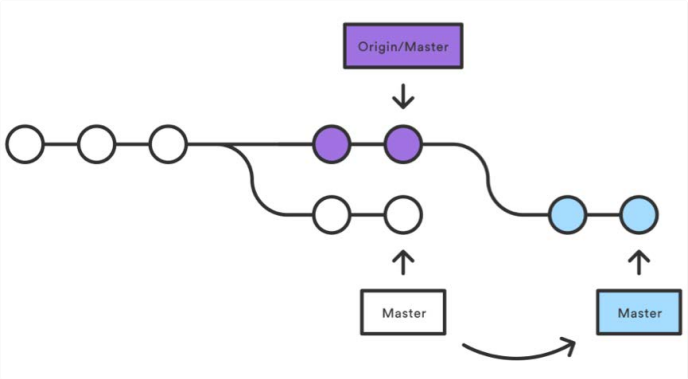
这个过程一般是下面这个样子的。

1. 从服务器上做 `git pull origin master` 把代码同步下来。
2. 改完后，`git commit` 到本地仓库中。
3. 然后 `git push origin master` 到远程仓库中，这样其他同学就可以得到你的代码了。

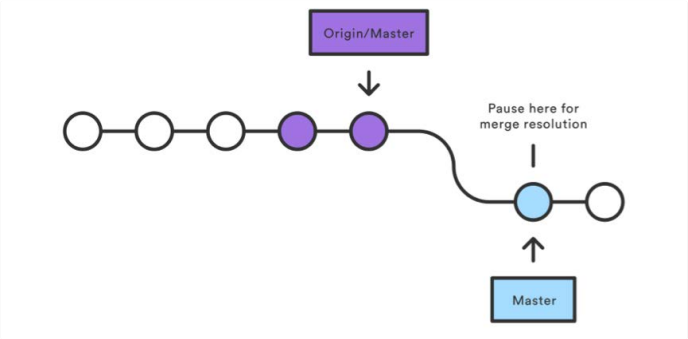
如果在第 3 步发现 `push` 失败，因为别人已经提交了，那么你需要先把服务器上的代码给 `pull` 下来，为了避免有 `merge` 动作，你可以使用 `git pull --rebase`。这样就可以把服务器上的提交直接合并到你的代码中，对此，Git 的操作是这样的。

1. 先把你本地提交的代码放到一边。
2. 然后把服务器上的改动下载下来。
3. 然后在本地把你之前的改动再重新一个一个地做 `commit`，直到全部成功。

如下图所示。Git 会把 Origin/Master 的远程分支下载下来（紫色的），然后把本地的Master分支上的改动一个一个地提交上去（蓝色的）。



如果有冲突，那么你要先解决冲突，然后做 `git rebase --continue`。如下图所示，git在做 `pull --rebase` 时，会一个一个地应用（apply）本地提交的代码，如果有冲突就会停下来，等你解决冲突。



功能分支协同工作流

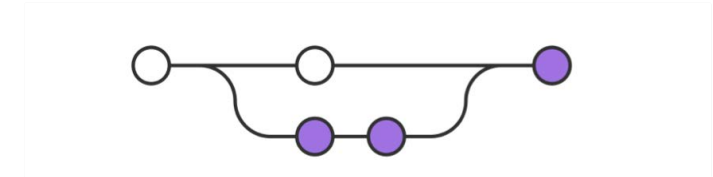
上面的那种方式有一个问题，就是大家都在一个主干上开发程序，对于小团队或是小项目你可以这么干，但是对比较大的项目或是人比较多的团队，这么干就会有很多问题。

最大的问题就是代码可能干扰太严重。尤其是，我们想安安静静地开发一个功能时，我们想把各个功能的代码变动隔离开来，同时各个功能又会有多个开发人员在开发。

这时，我们不想让各个功能的开发人员都在Master分支上共享他们的代码。我们想要的协同方式是这样的：同时开发一个功能的开发人员可以分享各自的代码，但是不会把代码分享给开发其他功能的开发人员，直到整个功能开发完毕后，才会分享给其他的开发人员（也就是进入主干分支）。

因此，我们引入“功能分支”。这个协同工作流的开发过程如下。

1. 首先使用 `git checkout -b new-feature` 创建“new-feature”分支。
2. 然后共同开发这个功能的程序员就在这个分支上工作，进行add、commit等操作。
3. 然后通过 `git push -u origin new-feature` 把分支代码push到服务器上。
4. 其他程序员可以通过`git pull --rebase`来拿到最新的这个分支的代码。
5. 最后通过Pull Request的方式做完Code Review后合并到Master分支上。



就像上面这个图显示的一样，紫色的分支就是功能分支，合并后就会像上面这个样子。

我们可以看到，其实，这种开发也是以服务器为中心的开发，还不是Git分布式开发，它只不过是分支来完成代码改动的隔离。

另外，我想提醒一下，为什么会叫“功能分支”，而不是“项目分支”？因为Git的最佳实践希望大家在开发的过程中，快速提交，快速合并，快速完成。这样可以少很多冲突的事，所以叫功能分支。

传统的项目分支开得太久，时间越长就越合不回去。这种玩法其实就是让我们把一个大项目切分成若干个小项目来执行（最好是一个小功能一个项目）。这样才是互联网式的快速迭代的开发流程。

GitFlow协同工作流

在真实的生产过程中，前面的协同工作流还是不能满足工作的要求。这主要因为我们的生产过程是比较复杂的，软件生产中会有各式各样的问题，并要面对不同的环境。我们要在不停地开发新代码的同时，维护线上的代码，于是，就有了下面这些需求。

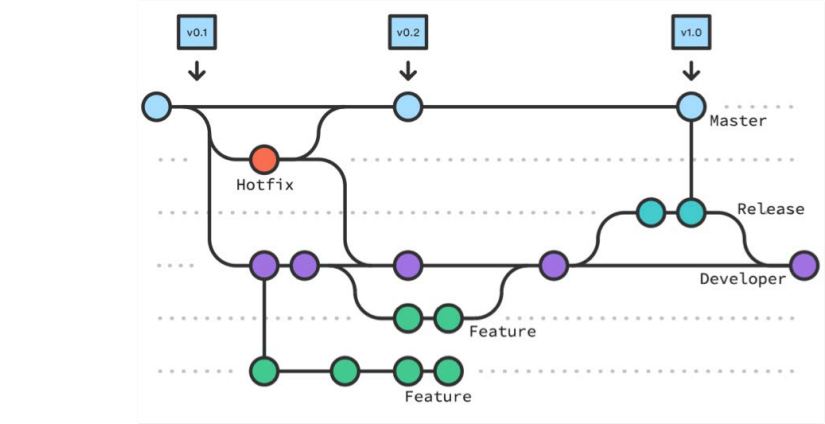
1. 希望有一个分支是非常干净的，上面是可以发布的代码，上面的改动永远都是可以发布到生产环境中的。这个分支上不能有中间开发过程中不可以上生产线的代码提交。
2. 希望当代码达到可以上线的状态时，也就是在alpha/beta release时，在测试和交付的过程中，依然可以开发下一个版本的代码。
3. 最后，对于已经发布的代码，也会有一些Bug-fix的改动，不会将正在开发的代码提交到生产线上去。

你看，面对这些需求，前面的那些协同方式就都不行了。因为我们不仅是要在整个团队中共享代码，我们要的更是管理好不同环境下的代码不互相干扰。说得技术一点儿就是，要管理好代码与环境的一致性。

为了解决这些问题，GitFlow协同工作流就出来了。

GitFlow协同工作流是由Vincent Driessen于2010年在A successful Git branching model 这篇文章介绍给世人的。

这个协同工作流的核心思想如下图所示。



整个代码库中一共有五种分支。

- Master分支。也就是主干分支，用作发布环境，上面的每一次提交都是可以发布的。
- Feature分支。也就是功能分支，用于开发功能，其对应的是开发环境。
- Developer分支。是开发分支，一旦功能开发完成，就向Developer分支合并，合并完成后，删除功能分支。这个分支对应的是集成测试环境。
- Release分支。当Developer分支测试达到可以发布状态时，开出一个Release分支来，然后做发布前的准备工作。这个分支对应的是预发环境。之所以需要这个Release分支，是我们的开发可以继续向前，不会因为要发布而被block住而不能提交。

一旦Release分支上的代码达到可以上线的状态，那么需要把Release分支向Master分支和Developer分支同时合并，以保证代码的一致性。然后再把Release分支删除掉。

- Hotfix分支。是用于处理生产线上代码的Bug-fix，每个线上代码的Bug-fix都需要开一个Hotfix分支，完成后，向Developer分支和Master分支上合并。合并完成后，删除Hotfix分支。

这就是整个GitFlow协同工作流的工作过程。我们可以看到：

1. 我们需要长期维护Master和Developer两个分支。
2. 这其中的方式还是有一定复杂度的，尤其是Release和Hotfix分支需要同时向两个分支作合并。所以，如果没有一个好的工具来支撑的话，这会因为我们可能会忘了做一些操作而导致代码不一致。
3. GitFlow协同虽然工作流比较重。但是它几乎可以应对所有公司的各种开发流程，包括瀑布模型，或是快速迭代模型。

GitHub/GitLab 协同工作流

GitFlow的问题

对于GitFlow来说，虽然可以解决我们的问题，但是也有很多问题。在GitFlow流行了一段时间后，圈内出现了一些不同的声音。参看下面两篇吐槽文章。

- [GitFlow considered harmful](#)
- [Why git flow does not work for us](#)

其中有个问题就是因为分支太多，所以会出现git log混乱的局面。具体来说，主要是git-flow使用git merge --no-ff来合并分支，在git-flow这样多个分支的环境下会让你的分支管理的log变得很难看。如下所示，左边是使用--no-ff参数在多个分支下的问题。



所谓--no-ff参数的意思是——no fast forward的意思。也就是说，合并的方法不要把这个分支的提交以前置合并的方式，而是留下一个merge的提交。这是把双刃剑，我们希望我们的--no-ff能像右边那样，而不是像左边那样。

对此的建议是：只有feature合并到developer分支时，使用--no-ff参数，其他的合并都不使用--no-ff参数来做合并。

另外，还有一个问题就是，在开发得足够快的时候，你会觉得同时维护Master和Developer两个分支是一件很无聊的事，因为这两个分支在大多数情况下都是一样的。包括Release分支，你会觉得创建的这些分支太无聊。

而你的整个开发过程也会因为这么复杂的管理变得非常复杂。尤其当你想回滚某些人的提交时，你就会发现这事似乎有点儿不好干了。而且在工作过程中，你会来来回回地切换工作的分支，有时候一不小心没有切换，就提交到了不正确的分支上，你还要回滚和重新提交，等等。

GitLab一开始是GitFlow的坚定支持者，后来因为这些吐槽，以及Hacker News和Reddit上大量的讨论，GitLab也开始不玩了。他们写了一篇[blog](#)来创造了一个新的Workflow——GitLab Flow，这个GitLab Flow是基于GitHub Flow来做的（参看：[GitLab Flow](#)）。

GitHub Flow

所谓GitHub Flow，其实也叫Forking flow，也就是GitHub上的那个开发方式。

1. 每个开发人员都把“官方库”的代码fork到自己的代码仓库中。
2. 然后，开发人员在自己的代码仓库中做开发，想干啥干啥。
3. 因此，开发人员的代码库中，需要配两个远程仓库，一个是自己的库，一个是官方库（用户的库用于提交代码改动，官方库用于同步代码）。
4. 然后在本地建“功能分支”，在这个分支上做代码开发。
5. 这个功能分支被push到开发人员自己的代码仓库中。
6. 然后，向“官方库”发起pull request，并做Code Review。
7. 一旦通过，就向官方库进行合并。

这就是GitHub的工作流程。

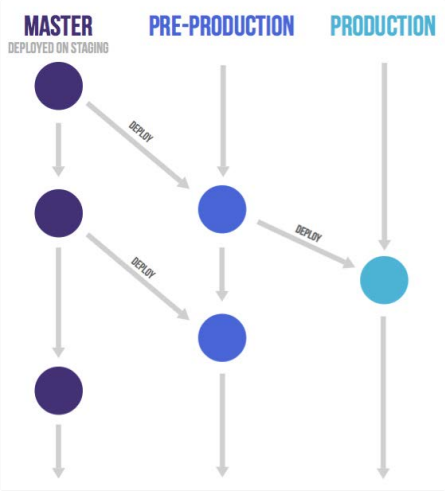
如果你有“官方库”的权限，那么就可以直接在“官方库”中建功能分支开发，然后提交pull request。通过Code Review后，合并进Master分支，而Master一旦有代码被合并就可以马上release。

这是一种非常Geek的玩法。这需要一个自动化的CI/CD工具做辅助。是的，CI/CD应该是开发中的标配了。

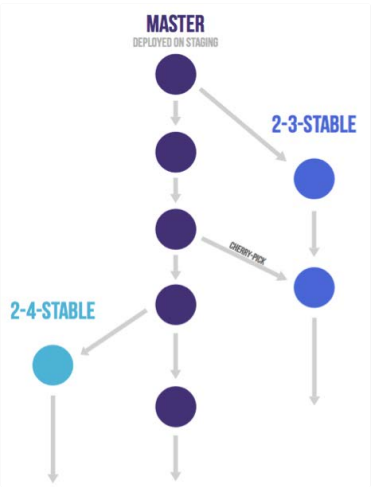
GitLab Flow

然而，GitHub Flow这种玩法依然会有好多问题，因为其虽然变得很简单，但是没有把我们的代码和我们的运行环境给联系在一起。所以，GitLab提出了几个优化点。

其中一个引入环境分支，如下图所示，其包含了预发布（Pre-Production）和生产（Production）分支。



而有些时候，我们还会有不同版本的发布，所以，还需要有各种release的分支。如下图所示。Master分支是一个roadmap分支，然后，一旦稳定了就建稳定版的分支，如2.3.stable分支和2.4.stable分支，其中可以cherry-pick master分支上的一些改动过去。



这样也就解决了两个问题：

- 环境和代码分支对应的问题；
- 版本和代码分支对应的问题。

老实说，对于互联网公司来说，环境和代码分支对应这个事，只要有个比较好的CI/CD生产线，这种环境分支应该也是没有必要的。而对于版本和代码分支的问题，我觉得这应该是有意义的，但是，最好不要维护太多的版本，版本应该是短暂的，等新的版本发布时，老的版本就应该删除掉了。

协同工作流的本质

对于上面这些各式各样的工作流的比较和思考，虽然，我个人非常喜欢GitHub Flow，在必要的时候使用上GitLab中的版本或环境分支。不过，我们现实生活中，还是有一些开发工作不是以功能为主要，而是以项目为主的。也就是说，项目的改动量可能比较大，时间和周期可能也比较长。

我在想，是否有一种工作流，可以面对我们现实工作中的各种情况。但是，我想这个世界太复杂了，应该不存在一种一招鲜吃遍天的放之四海皆准的银弹方案。所以，我们还要根据自己的实际情况来挑选适合我们的协同工作的方式。

而代码的协同工作流属于SCM（Software Configuration Management）的范畴，要挑选好适合自己的方式，我们需要知道软件工程配置管理的本质。根据这么多年来我在各个公司的经历，有互联网的，有金融的，有项目的，有快速迭代的等，我认为团队协同工作的本质不外乎这么几个事儿。

1. 不同的团队能够尽可能并行开发。
2. 不同软件版本和代码的一致性。
3. 不同环境和代码的一致性。
4. 代码总是在稳定和不稳定间交替。我们希望生产线上的代码总是能对应到稳定的代码上来。

基本上述的四个事儿，上述的工作流大都是以建立不同的分支，来做到开发并行、代码和环境版本一致，以及稳定的代码。

要选择适合自己的协同工作流，我们就不得不谈一下软件开发的工作模式。

首先，我们知道软件开发的趋势一定是下面这个样子的。

- 以微服务或是SOA为架构的方式。一个大型软件会被拆分成若干个服务，那么，我们的代码应该也会跟着服务拆解成若干个代码仓库。这样一来，我们的每个代码仓库都会变小，于是我们的协同工作流程就会变简单。

对于每个服务的代码仓库，我们的开发和迭代速度也会变得很快，开发团队也会跟服务一样被拆分成多个小团队。这样一来，Gitflow这种协同工作流程就非常重了，而GitHub这种方式或是功能分支这种方式会更适合我们的开发。

- 以DevOps为主的开发流程。DevOps关注于CI/CD，需要我们有自动化的集成测试和持续部署的工具。这样一来，我们的代码发布速度就会大大加快，每一次提交都能很快地被完整地集成测试，并很快地发布到生产线上。

于是，我们就可以使用更简单的协同工作流程，不需要维护多个版本，也不需要关注不同的运行环境，只需要一套代码，就可以了。GitHub Flow或是功能分支这种方式也更适应这种开发。

你看，如果我们将软件开发升级并简化到SOA服务化以及DevOps上来，那么协同工作流就会变得非常简单。所以，协同工作流的本质，并不是怎么玩好代码仓库的分支策略，而是玩好我们的软件架构和软件开发流程。

当然，服务化和DevOps是每个开发团队需要去努力的目标，但就算是这样，也有某些情况我们需要用重的协同工作的模式。比如，整个公司在做一个大的升级项目，这其中会对代码做一个大的调整（很有可能是一次重大的重构）。

这个时候，可能还有一些并行的开发需要做，如一些小功能的优化，一些线上Bug的处理，我们可能还需要在生产线上做新旧两个版本的A/B测试。在这样的情况下，我们可能会或多或少地使用GitFlow协同工作流。

但是，这样的方式不会是常态，是特殊时期，我们不可能隔三差五地对系统做架构或是对代码做大规模的重构。所以，在大多数情况下，我们还是应该选择一个比较轻量级的协同工作流，而在特殊时期特例特办。

最后，让我用一句话来结束这篇文章——与其花时间在Git协同工作流上，还不如把时间花在调整软件架构和自动化软件生产和运维流程上来，这才是真正简化协同工作流程的根本。

（这篇文章中有大量的Git命令，很难用音频体现出来，所以没有录制音频，还望谅解。）



能蝦	2018-01-17
Git-Flow 和 GitLab-Flow 都有 release 分支，虽然都称为 release，但作用完全不同。 在 Git-Flow 中，release 是介于 dev 和 master 之间，作用是发布前的准备工作（通常应该是用于更全面测试），从 dev 分叉 release 是为了避免阻塞 dev 分支上继续开发新功能；release 属于临时性分支，准备工作完成后是以 Fast-Forward 方式合并到 master；release 上可能会修复一些 bug，所以还要把 release 合并回 dev；发布软件是从 master 上构建。从名称看，release 应该叫做 pre-release，而 master 应该叫 release。 在 GitLab-Flow 中，release 是从 master 分叉出来，可能会有多个 release 分支，对应软件不同版本。从 master 分叉一个 release（版本）后，可能发现还需要合入一些新功能，所以通过 cherry-pick 方式从 master 中挑选某些功能，不能直接从 master 合并，因为仅需要 master 上的某些新功能而不是全部新功能。release 属于长期分支，而且分叉以后不会再与其它分支合并。除非某个版本不再维护，不然对应 release 会持续存在，因为发布出去的软件随时可能发现 bug 需要修复。	
_CountingStars	2018-01-22
请问老师的架构图 示意图 是用什么软件画的 感觉很不错	
码字的路人	2018-01-16
耗子哥，写的很在理。是应该在项目本身就开始根据业务进行拆分。再进行监控等	
小毅	2018-01-16
皓哥，文章里面的说：一个大型软件会被拆分成若干个服务，那么代码应该也会跟着服务拆解成若干个代码仓库～ 这样随着组里面的项目越来越多，开发维护起来会不会不方便？	
tao	2018-06-17
我觉得着团队大小吧。7人以下的团队小而快，直接全部用单master分支就可以，所有人用rebase，防止出现太多分叉。feature分支和bug分支都开发人员自己管理就好	
王磊	2018-06-15
git state还是个git stash?	
作者回复	2018-06-15
写错了，是got stash	
jerry	2018-06-01
我们开发测试uat环境都使用临时分支，会从master分支创建这个临时分支然后把多个功能分支合并到这个临时分支编译发布 整个研发流程大概是功能分支会关联到一个需求项目 这个项目会涉及多个应用的多个分支，当这个需求做完送测的时候会找到这个项目下的所有应用和分支，生成一个送测单 当然这个送测单包含了这个项目的所有分支 比如应用代码分支 配置变更 和数据库变更，然后测试通过后会上线到接口同时批量生成上线单 也是包含所有的变更和一个变更checklist 比如有哪些分支需要合并到master 有哪些配置需要变更 有哪些sql需要执行这样，这些变更一般都是手工执行，最后自动化程序会去检查这些变更是不是执行了，没问题就执行上线流程 发布到uat及发布到生产 当然这些都是自动化程序去做的，只有合并到master 和执行sql是手工执行	
C J J	2018-04-13
嘿，采用什么方式管理代码还是看项目大小，人员多少。	
枫	2018-01-23
皓哥，刚才没留下邮箱：15920199030@139.com	
枫	2018-01-23
皓哥，我java开发做了6年，一直在一家公司做移动的网管系统，感觉自己还处于很低级的阶段，我想成为一个架构师，您可以帮忙指导下，学习的路径吗？谢谢您！	
cellardoor	2018-01-19
有些人喜欢把事情做的很复杂，以至于看不出有什么问题。有些人喜欢把事情做的很简单，看起来似乎有些问题。	
郎哲	2018-01-16
我们目前方式 类「GitLab Flow」小的变动功能上master，稍微负责的打功能分支。在一个目标内打release版本，fixbug在release上并pick到master。现在遇到 多个项目都对release有要求，然后就变成串行了。正考虑为强要求的项目单独release和维护。客户端的不干了 说版本太多不发版本容易出错。目前正忍受串行。	
很大气	2018-01-16
在实际开发中，如果有并行的需求应该怎样管理分支呢？	

