Git分支管理策略



# 主分支Master

首先，代码库应该有一个、且仅有一个主分支。所有提供给用户使用的正式版本，都在这个主分支上发布。Git主分支的名字，默认叫做Master。它是自动建立的，版本库初始化以后，默认就是在主分支在进行开发。



# 开发分支Develop

主分支只用来分布重大版本，日常开发应该在另一条分支上完成。我们把开发用的分支，叫做Develop。这个分支可以用来生成代码的最新隔夜版本（nightly）。如果想正式对外发布，就在Master分支上，对Develop分支进行"合并"（merge）。

|  |
| --- |
| Git创建Develop分支的命令：git checkout -b develop master  将Develop分支发布到Master分支的命令：  # 切换到Master分支  git checkout master  # 对Develop分支进行合并  git merge --no-ff develop  这里稍微解释一下，上一条命令的--no-ff参数是什么意思。默认情况下，Git执行"快进式合并"（fast-farward merge），会直接将Master分支指向Develop分支。使用--no-ff参数后，会执行正常合并，在Master分支上生成一个新节点。为了保证版本演进的清晰，我们希望采用这种做法。  http://nvie.com/img/merge-without-ff@2x.png |

# 临时性分支

前面讲到版本库的两条主要分支：Master和Develop。前者用于正式发布，后者用于日常开发。其实，常设分支只需要这两条就够了，不需要其他了。

但是，除了常设分支以外，还有一些临时性分支，用于应对一些特定目的的版本开发。临时性分支主要有三种：

* 功能（feature）分支
* 预发布（release）分支
* 修补bug（fixbug）分支

这三种分支都属于临时性需要，使用完以后，应该删除，使得代码库的常设分支始终只有Master和Develop。

## 功能分支（对应我们的开发分支）

第一种是功能分支，它是为了开发某种特定功能，从Develop分支上面分出来的。开发完成后，要再并入Develop。功能分支的名字，可以采用feature-\*的形式命名。

|  |
| --- |
| #创建一个功能分支  git checkout -b feature-x develop  #开发完成后，将功能分支合并到develop分支  git checkout develop  git merge --no-ff feature-x  #删除feature分支  git branch -d feature-x |

## 预发布分支（预生产）

第二种是预发布分支，它是指发布正式版本之前（即合并到Master分支之前），我们可能需要有一个预发布的版本进行测试。

预发布分支是从Develop分支上面分出来的，预发布结束以后，必须合并进Develop和Master分支。它的命名，可以采用release-\*的形式。

|  |
| --- |
| #创建一个预发布分支  git checkout -b release-1.2 develop  #确认没有问题后，合并到master分支  git checkout master  git merge --no-ff release-1.2  # 对合并生成的新节点，做一个标签  git tag -a 1.2  #再合并到develop分支  git checkout develop  git merge --no-ff release-1.2  #最后，删除预发布分支  git branch -d release-1.2 |

## 修补bug分支（bug\_fix分支）

最后一种是修补bug分支。软件正式发布以后，难免会出现bug。这时就需要创建一个分支，进行bug修补。修补bug分支是从Master分支上面分出来的。修补结束以后，再合并进Master和Develop分支。它的命名，可以采用fixbug-\*的形式。

|  |
| --- |
| #创建一个修补bug分支  git checkout -b fixbug-0.1 master  #修补结束后，合并到master分支  git checkout master  git merge --no-ff fixbug-0.1  git tag -a 0.1.1  #再合并到develop分支  git checkout develop  git merge --no-ff fixbug-0.1  #最后，删除"修补bug分支"  git branch -d fixbug-0.1  http://nvie.com/img/hotfix-branches@2x.png |

参考教程：http://nvie.com/posts/a-successful-git-branching-model/

# Git 工作流程

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。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。

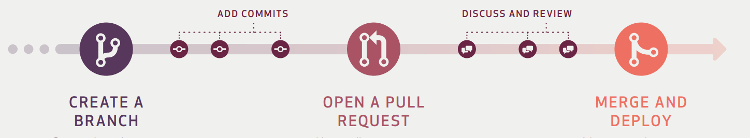
## Git flow

最早诞生、并得到广泛采用的一种工作流程，就是Git flow。其详细介绍，见前三个主标题说明。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。

## 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。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。） |

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

## 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。

### 上游优先

Gitlab flow 的最大原则叫做"上游优先"（upsteam 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

### 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。

### 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。