# Git版本控制与工作流

这篇文章是针对git版本控制和工作流的总结，如果有些朋友之前还没使用过git，对git的基本概念和命令不是很熟悉，可以从以下基本教程入手：

## ****基本概念****

**Git是什么？**

Git是**分布式**版本控制系统，与SVN类似的**集中化**版本控制系统相比，集中化版本控制系统虽然能够令多个团队成员一起协作开发，但有时如果中央服务器宕机的话，谁也无法在宕机期间提交更新和协同开发。甚至有时，中央服务器磁盘故障，恰巧又没有做备份或备份没及时，那就可能有丢失数据的风险。

但Git是分布式的版本控制系统，客户端不只是提取最新版本的快照，而且将整个代码仓库镜像复制下来。如果任何协同工作用的服务器发生故障了，也可以用任何一个代码仓库来恢复。而且在协作服务器宕机期间，你也可以提交代码到本地仓库，当协作服务器正常工作后，你再将本地仓库同步到远程仓库。

## ****为什么要使用Git****

* 能够对文件版本控制和多人协作开发
* 拥有强大的分支特性，所以能够灵活地以不同的工作流协同开发
* 分布式版本控制系统，即使协作服务器宕机，也能继续提交代码或文件到本地仓库，当协作服务器恢复正常工作时，再将本地仓库同步到远程仓库。
* 当团队中某个成员完成某个功能时，通过pull request操作来通知其他团队成员，其他团队成员能够review code后再合并代码。

## ****Git有哪些特性****

* 文件三种状态(modified, staged, committed)
* 直接记录快照，而非差异比较
* 多数操作仅添加操作
* 近乎所有操作都是本地执行
* 时刻保持数据完整性

有关以上特性的详细解释，请查看Pro git的**git基础章节**

## ****Git基本工作流程****

1. 在git版本控制的目录下修改某个文件
2. 使用git add命令对修改后的文件快照，保存到暂存区域
3. 使用git commit命令提交更新，将保存在暂存区域的文件快照永久转储到 Git 目录中

## ****Git基本技巧****

* 自动补全
* Git 命令别名

关于具体如何使用自动补全和命名别名技巧，请查看Pro git的**技巧和窍门**

## ****Git版本控制****

**创建仓库**

* git init
* git clone
* git config

**保存修改**

* git add
* git commit

**查看仓库**

* git status
* git log --oneline

**撤销修改**

**查看之前的commit**

* git checkout <commit> <file>
* git checkout <commit>
* git checkout <branch>

**撤销公共修改**

* git revert <commit>

**撤销本地修改**

* git reset
* git clean

**重写Git历史记录**

* git commit --amend
* git rebase
* git reflog

## ****Git协作开发****

**分支**

* git branch
* git checkout
* git merge

**仓库同步**

* git remote
* git fetch
* git pull
* git push

## ****Git工作流****

由于git拥有强大的分支特性，它的工作流比较灵活而缺乏约束，于是参考Atlassian Git Tutorial的Comparing Workflows章节提供四种Git工作流：

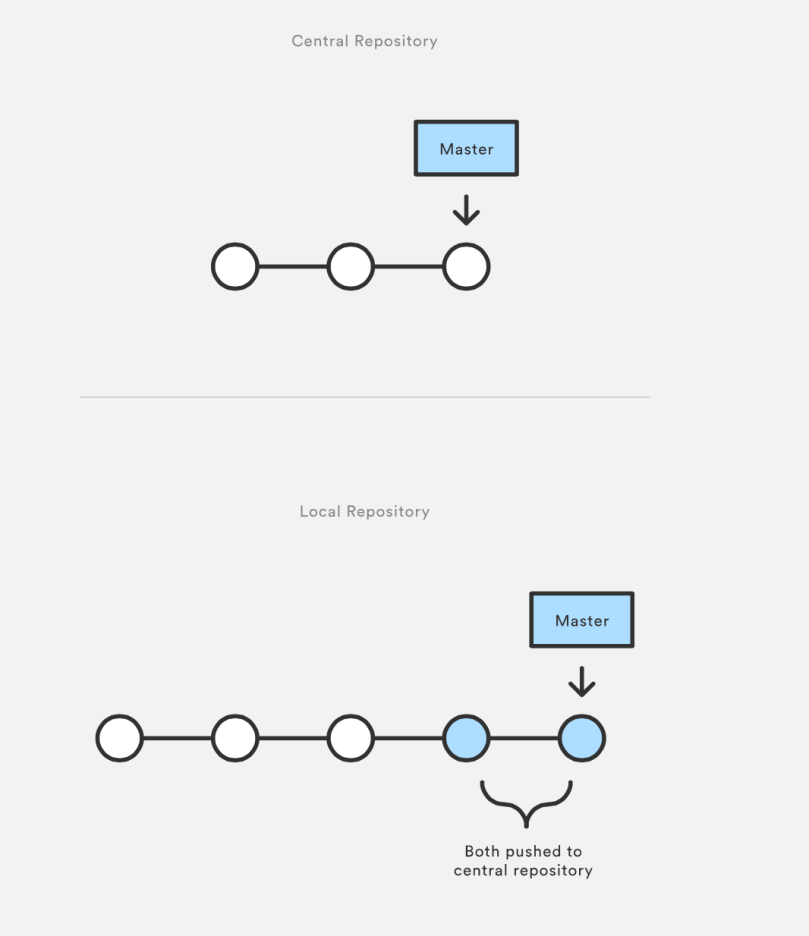
* Centralized Workflow
* Feature Branch Workflow
* Gitflow Workflow
* Forking Workflow

以上工作流只是参考指南，而不是具体规则。你可以根据自己实际情况来选择适合自己的工作流或微调来满足自己的需要。

## ****Centralized Workflow****

过渡到分布式版本控制系统看起来像一个艰巨的任务，但如果你充分利用好git的话，你不必改变你既有的工作流，你的团队可以采用与之前使用SVN一样的方式来开发项目。

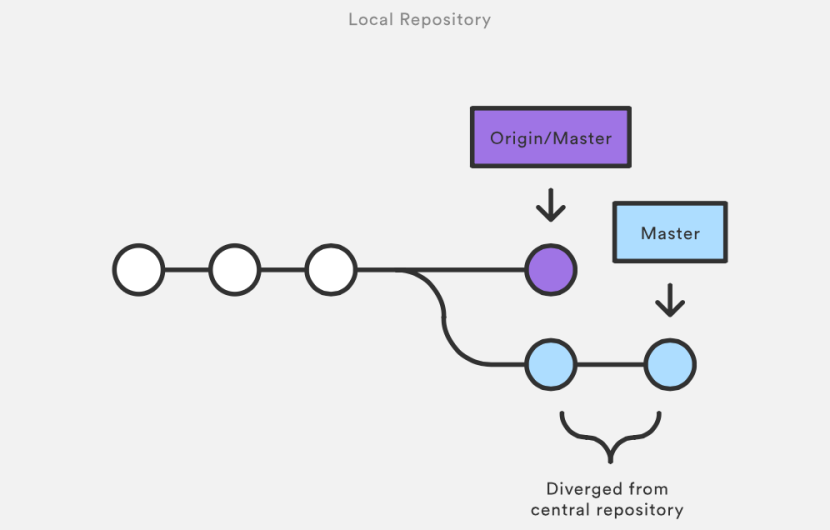
**如何工作**



Centralized Workflow

1. 从远程仓库(central repository)克隆工程到本地仓库(local repository) ---git clone
2. 在本地仓库编辑文件和提交更新 ---git add和git commit
3. fetch远程仓库已更新的commit到本地仓库和rebase到已更新的commit的上面 ---git fetch和gitrebase或git pull --rebase
4. push本地主分支(master branch)到远程仓库 ---git push

**管理冲突**



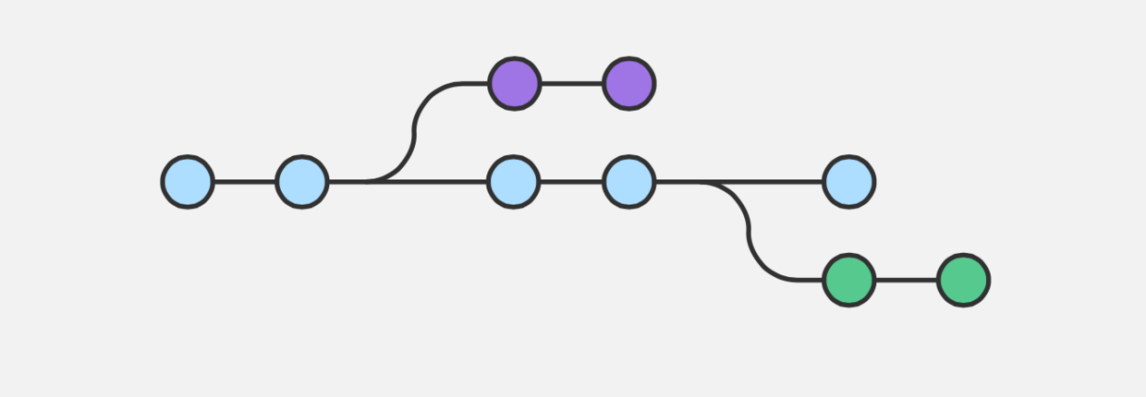
File Conflicts

* **何时发生冲突：**在开发者发布它们功能之前，他们需要fetch远程仓库已更新的commit到本地仓库和rebase到已更新的commit的上面。有时，本地提交与远程提交会发生冲突，git会暂停rebase过程来让你手动解决冲突。
* **如何解决冲突：**你可以使用git status和git add来手动解决合并时冲突。

## ****Feature Branch Workflow****

Feature Branch Workflow的主要思想就是在开发每个功能时都应该创建一个独立的分支而不只是使用主分支。由于每个分支是独立且互不影响，这就意味着主分支不会包含broken code，对持续集成环境是很有帮助的。

**如何工作**



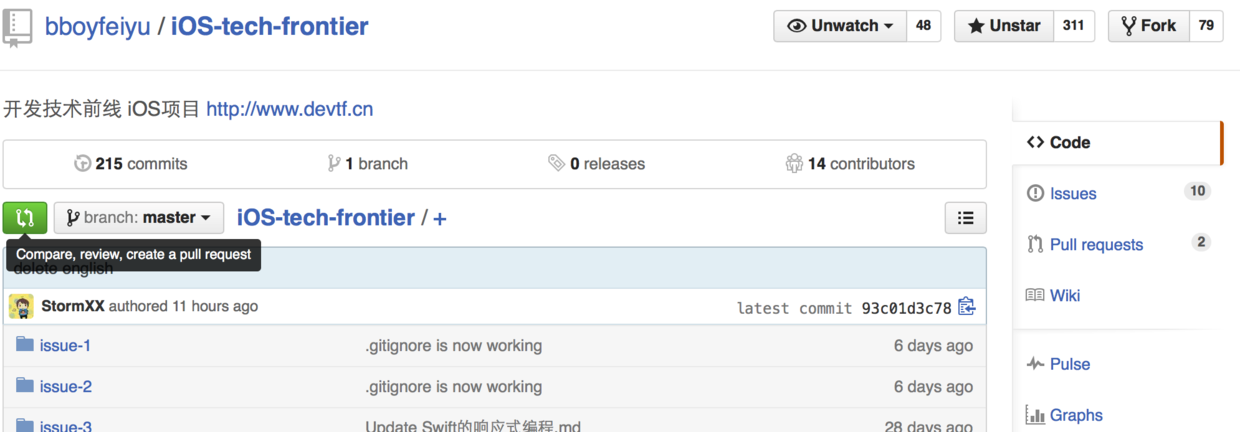
Feature Branch Workflow

1. 仍然使用远程仓库(central repository)和主分支(master branch)仍记录官方工程的历史
2. 开发者每次开发新功能时都创建一个新分支 ---git checkout -b
3. Feature branches应该推送到远程仓库(central repository) ---git push
4. 发送pull request来请求管理员能否合并到主分支(master branch)
5. 发布新功能到远程仓库(central repository)

**Pull Request**

Pull request是一种当开发者完成一个新功能后向其他团队成员发送通知的机制。它的使用过程如下：

* 开发者可以通过Github或Bitbucket发送pull request



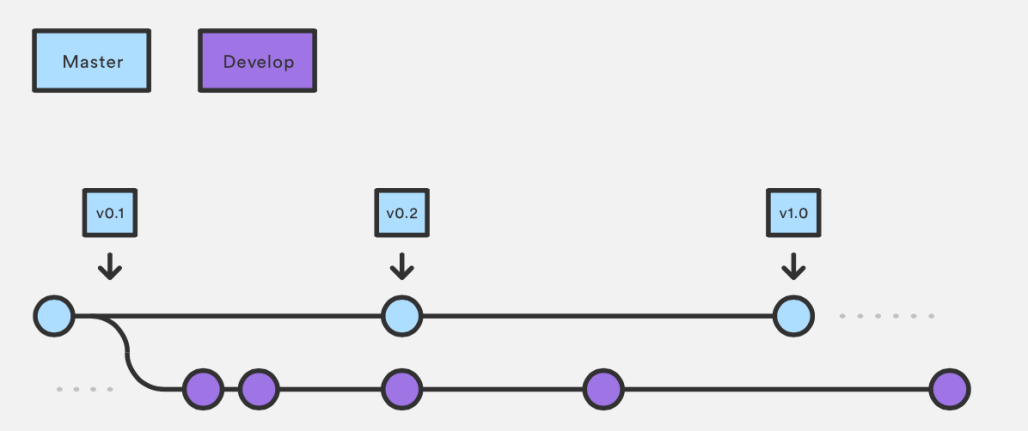
Pull request on Github

* 其他的团队成员审查、讨论和修改代码
* 项目维护者合并新增功能分支到主分支(master branch)，然后关闭pull request

## ****Gitflow Workflow****

Feature Branch Workflow是一种非常灵活的开发方式。对于一些规模比较大的团队，最好就是给特定的分支赋予不同的角色。除了**功能分支(feature branch)**，Gitflow Workflow还使用独立的分支来**准备发布(preparing)**，**维护(maintaining)**, 和**记录版本(recording releases)**。下面我会逐个介绍这个几个分支：Historical Branches、Feature Branches、Release Branches和Maintenance Branches。

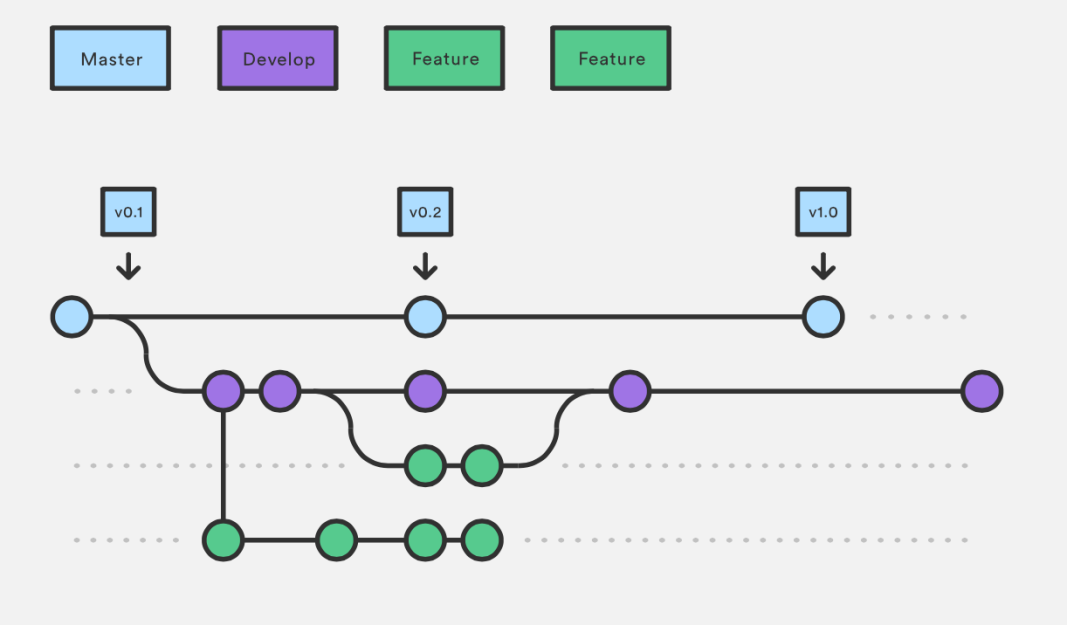
**Historical Branches**



Historical Branches

* **master分支**保存官方发布历史
* **develop分支**衍生出各个feature分支

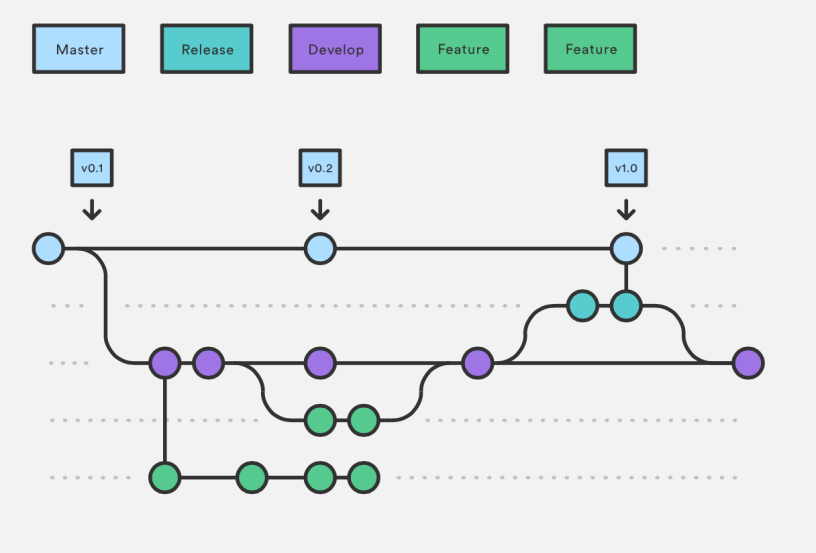
**Feature Branches**



Feature Branches

* **feature分支**使用develop分支作为它们的父类分支
* 当其中一个feature分支完成后，它会合并会develop分支
* feature分支应该从不与master分支直接交互

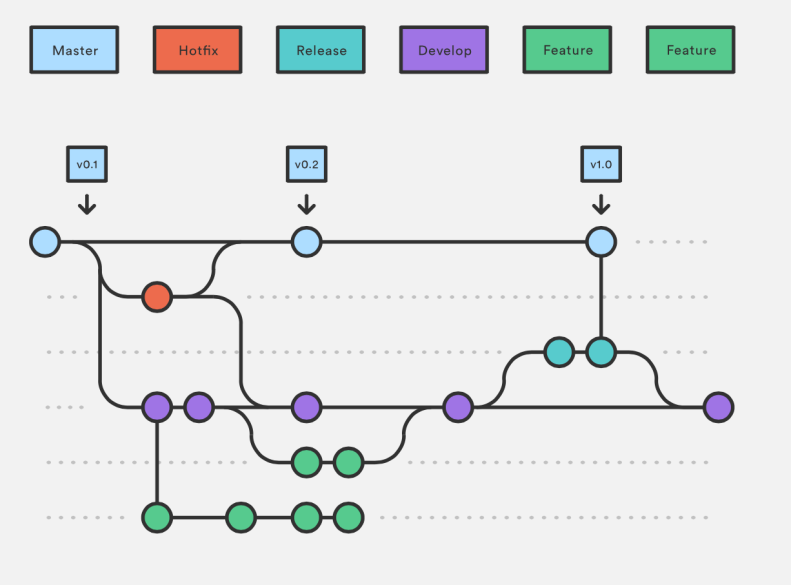
**Release Branches**



Release Branches

* **release分支**主要用来清理释放、测试和更新文档
* 一旦develop分支获得足够的功能来发布时，你可以从develop衍生出一个release分支
* 一旦准备好上架，release合并到master分支并且标记一个版本号
* 另外，还需要合并回develop分支

**Maintenance Branches**



Maintenance Branches.png

* **maintenance分支**用来快速给已发布产品修复bug或微调功能
* 它从master分支直接衍生出来
* 一旦完成修复bug，它应该合并回master分支和develop分支
* master应该被标记一个新的版本号

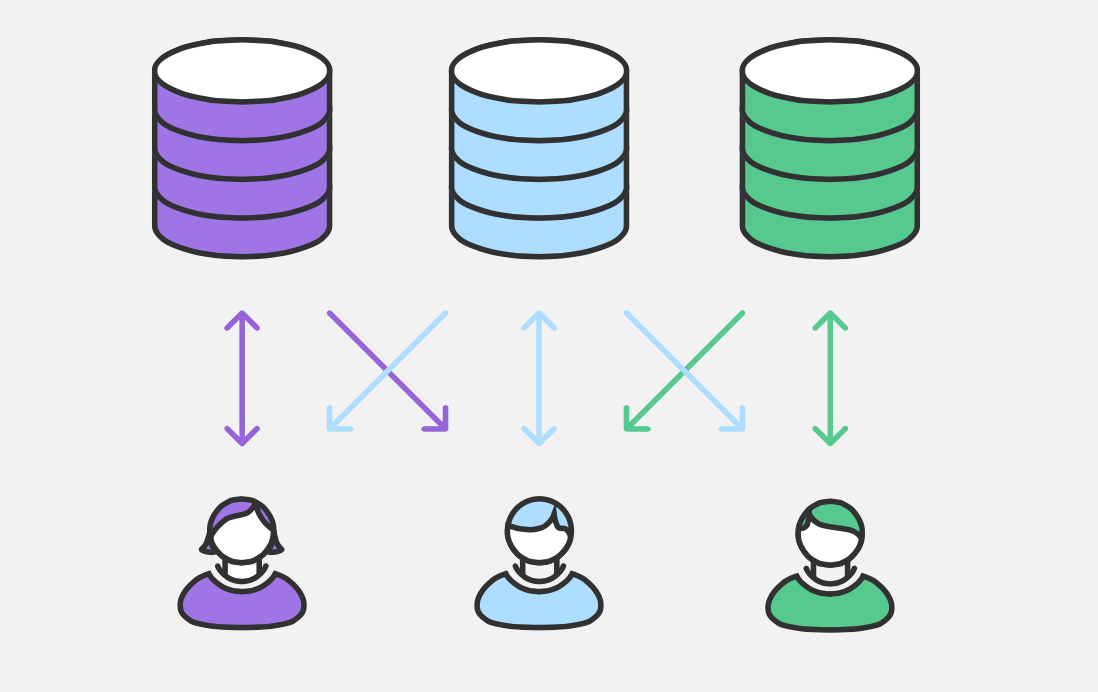
**标记Tags**

使用两个命令来给master分支标记版本号：

* git tag -a 0.1 -m "Initial public release" master
* git push origin master --tags

## ****Forking Workflow****

Forking Workflow与以上讨论的工作流很不同，一个很重要的区别就是它不只是多个开发共享一个远程仓库(central repository)，而是每个开发者都拥有一个独立的服务端仓库。也就是说每个contributor都有两个仓库：本地私有的仓库和远程共享的仓库。

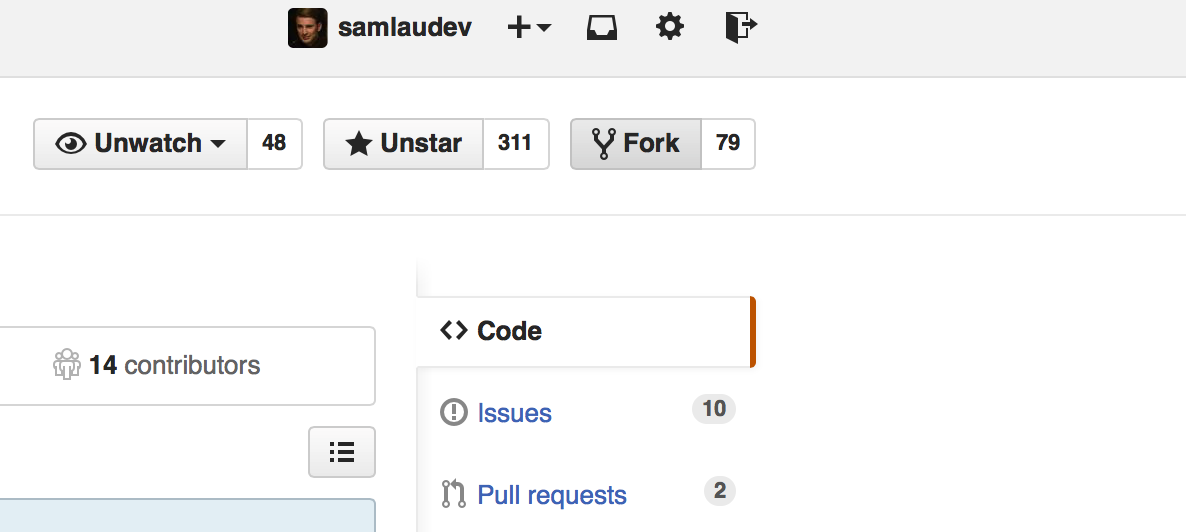


Forking Workflow

Forking Workflow这种工作流主要好处就是每个开发者都拥有自己的远程仓库，可以将提交的commits推送到自己的远程仓库，但只有工程维护者才有权限push提交的commits到官方的仓库，其他开发者在没有授权的情况下不能push。Github很多**开源项目**都是采用Forking Workflow工作流。

**如何工作**

1. 在服务器上有一个官方公共的仓库
2. 开发者fork官方仓库来创建它的拷贝，然后存放在服务器上



Fork official repository.png

1. 当开发者准备好发布本地的commit时，他们push commit到他们自己的公共仓库
2. 在自己的公共仓库发送一个pull request到官方仓库
3. 维护者pull贡献者的commit到他自己的本地仓库
4. 审查代码确保它不会破坏工程，合并它到本地仓库的master分支
5. push master分支到服务器上的官方仓库
6. 其他开发者应该同步官方仓库。

分享来自：IMOOC.COM