[使用git和github进行协同开发流程](https://segmentfault.com/a/1190000002413519)[[1]](#footnote-1)

* [github](https://segmentfault.com/t/github/blogs)
* [git](https://segmentfault.com/t/git/blogs)

[**戴嘉华**](https://segmentfault.com/u/livoras) 2014年12月08日发布

<https://segmentfault.com/a/1190000002413519>

## 目录

* 前言
* 仓库（Repository）  
    
  + 源仓库
  + 开发者仓库
* 分支（Branch）  
    
  + 永久性分支
  + 暂时性分支
* 工作流（workflow）
* 总结
* 参考资料

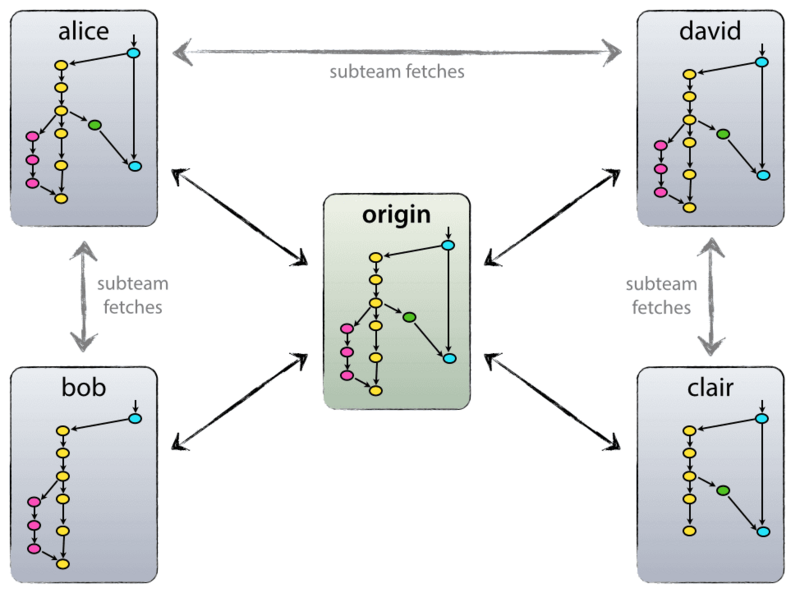
# 前言

（本文假设各位已经对基本git的基本概念、操作有一定的理解，如无相关git知识，可以参考[Pro Git](http://git-scm.com/book/zh/)这本书进行相关的学习和练习）

很多项目开发都会采用git这一优秀的分布式版本管理工具进行项目版本管理，使用github开源平台作为代码仓库托管平台。由于git的使用非常灵活，在实践当中衍生了很多种不同的工作流程，不同的项目、不同的团队会有不同的协作方式。

本文将介绍一种前人已经在各种大小项目中经过千锤百炼总结出来的一种比较成功的git工作流，这种工作流已经被成功用于许多团队开发当中。掌握git，掌握这种工作流，对大家以后的学习、开发工作大有好处。

先上一张图吓大家一下：



上面一张图展示了一种使用git进行项目协同开发的模式，接下来会进行详细介绍。

## 仓库（Repository）

在项目的开始到结束，我们会有两种仓库。一种是源仓库（origin），一种是开发者仓库。上图中的每个矩形都表示一个仓库，正中间的是我们的源仓库，而其他围绕着源仓库的则是开发者仓库。

### 源仓库

在项目的开始，项目的发起者构建起一个项目的最原始的仓库，我们把它称为origin，例如我们的PingHackers网站，origin就是这个[PingHackers/blog](https://github.com/PingHackers/blog)了。源仓库的有两个作用：

1. 汇总参与该项目的各个开发者的代码
2. 存放趋于稳定和可发布的代码

源仓库应该是受保护的，开发者不应该直接对其进行开发工作。只有项目管理者（通常是项目发起人）能对其进行较高权限的操作。

### 开发者仓库

上面说过，任何开发者都不会对源仓库进行直接的操作，源仓库建立以后，每个开发者需要做的事情就是把源仓库的“复制”一份，作为自己日常开发的仓库。这个复制，也就是github上面的fork。

每个开发者所fork的仓库是完全独立的，互不干扰，甚至与源仓库都无关。每个开发者仓库相当于一个源仓库实体的影像，开发者在这个影像中进行编码，提交到自己的仓库中，这样就可以轻易地实现团队成员之间的并行开发工作。而开发工作完成以后，开发者可以向源仓库发送pull request，请求管理员把自己的代码合并到源仓库中，这样就实现了**分布式开发工作**，和最后的集中式的管理。

## 分支（Branch）

分支是git中非常重要的一个概念，也是git这一个工具中的大杀器，必杀技。在其他集中式版本管理工具（SVN/CVS）把分支定位为高级技巧，而在git中，分支操作则是每个开发人员日常工作流。利用git的分支，可以非常方便地进行开发和测试，如果使用git没有让你感到轻松和愉悦，那是因为你还没有学会使用分支。不把分支用出一点翔来，不要轻易跟别人说你用过git。

在文章开头的那张图中，每一个矩形内部纷繁的枝蔓便是git的分支模型。可以看出，每个开发者的仓库都有自己的分支路线，而这些分支路线会通过代码汇总映射到源仓库中去。

我们为git定下一种分支模型，在这种模型中，分支有两类，五种

* 永久性分支  
    
  + master branch：主分支
  + develop branch：开发分支
* 临时性分支  
    
  + feature branch：功能分支
  + release branch：预发布分支
  + hotfix branch：bug修复分支

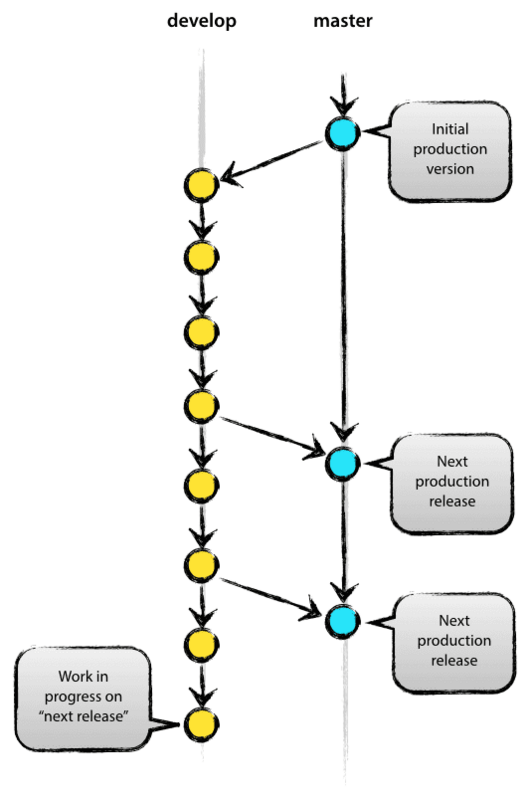
### 永久性分支

永久性分支是寿命无限的分支，存在于整个项目的开始、开发、迭代、终止过程中。永久性分支只有两个master和develop。

**master**：主分支从项目一开始便存在，它用于存放经过测试，已经完全稳定代码；在项目开发以后的任何时刻当中，master存放的代码应该是可作为产品供用户使用的代码。所以，应该随时保持master仓库代码的清洁和稳定，确保入库之前是通过完全测试和代码reivew的。master分支是所有分支中最不活跃的，大概每个月或每两个月更新一次，每一次master更新的时候都应该用git打上tag，说明你的产品有新版本发布了。

**develop**：开发分支，一开始从master分支中分离出来，用于开发者存放基本稳定代码。之前说过，每个开发者的仓库相当于源仓库的一个镜像，每个开发者自己的仓库上也有master和develop。开发者把功能做好以后，是存放到自己的develop中，当测试完以后，可以向管理者发起一个pull request，请求把自己仓库的develop分支合并到源仓库的develop中。

所有开发者开发好的功能会在源仓库的develop分支中进行汇总，当develop中的代码经过不断的测试，已经逐渐趋于稳定了，接近产品目标了。这时候，我们就可以把develop分支合并到master分支中，发布一个新版本。所以，一个产品不断完善和发布过程就正如下图：

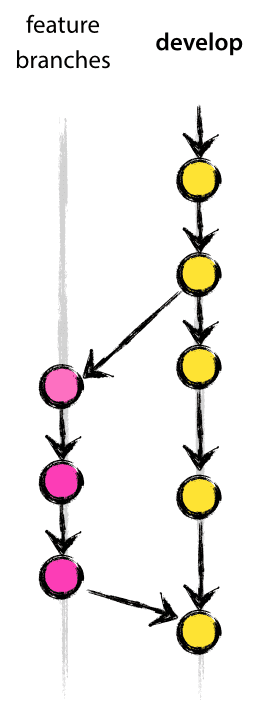


注意，任何人不应该向master直接进行无意义的合并、提交操作。正常情况下，master只应该接受develop的合并，也就是说，master所有代码更新应该源于合并develop的代码。

### 暂时性分支

暂时性分支和永久性分支不同，暂时性分支在开发过程中是一定会被删除的。所有暂时性分支，一般源于develop，最终也一定会回归合并到develop。

**feature**：功能性分支，是用于开发项目的功能的分支，是开发者主要战斗阵地。开发者在本地仓库从develop分支分出功能分支，在该分支上进行功能的开发，开发完成以后再合并到develop分支上，这时候功能性分支已经完成任务，可以删除。功能性分支的命名一般为feature-\*，\*为需要开发的功能的名称。



举一个例子，假设我是一名PingHackers网站的开发者，已经把源仓库fork了，并且clone到了本地。现在要开发PingHackers网站的“讨论”功能。我在本地仓库中可以这样做：

step 1: 切换到develop分支

**>>>** git checkout develop

step 2: 分出一个功能性分支

**>>>** git checkout -b feature-discuss

step 3: 在功能性分支上进行开发工作，多次commit，测试以后...

step 4: 把做好的功能合并到develop中

**>>>** git checkout develop

# 回到develop分支

**>>>** git merge --no-ff feature-discuss

# 把做好的功能合并到develop中

**>>>** git branch -d feature-discuss

# 删除功能性分支

**>>>** git push origin develop

# 把develop提交到自己的远程仓库中

这样，就完成一次功能的开发和提交。

**release**：预发布分支，当产品即将发布的时候，要进行最后的调整和测试，这时候就可以分出一个预发布分支，进行最后的bug fix。测试完全以后，发布新版本，就可以把预发布分支删除。预发布分支一般命名为release-\*。

**hotfix**：修复bug分支，当产品已经发布了，突然出现了重大的bug。这时候就要新建一个hotfix分支，继续紧急的bug修复工作，当bug修复完以后，把该分支合并到master和develop以后，就可以把该分支删除。修复bug分支命名一般为hotfix-\*

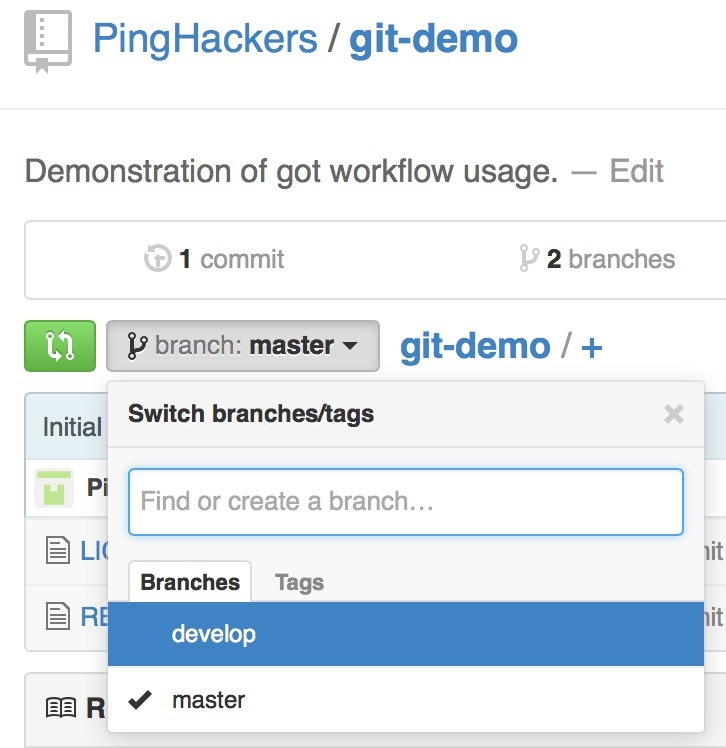
release和hotfix分支离我们还比较遥远。。就不详述，有兴趣的同学可以参考本文最后的参考资料进行学习。

## 工作流（Workflow）

啰嗦讲了这么多，概念永远是抽象的。对于新手来说，都喜欢一步一步的步骤傻瓜教程，接下来，我们就一步一步来操作上面所说的工作流程，大家感受一下：

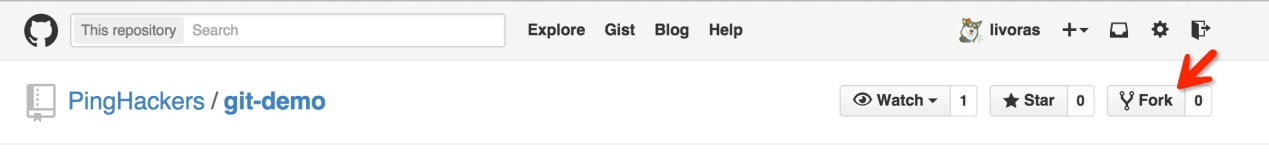
### Step 1：源仓库的构建

这一步通常由项目发起人来操作，我们这里把管理员设为PingHackers，假设PingHackers已经为我们建立起了一个源仓库[PingHackers/git-demo](https://github.com/PingHackers/git-demo)，并且已经初始化了两个永久性分支master和develop，如图：

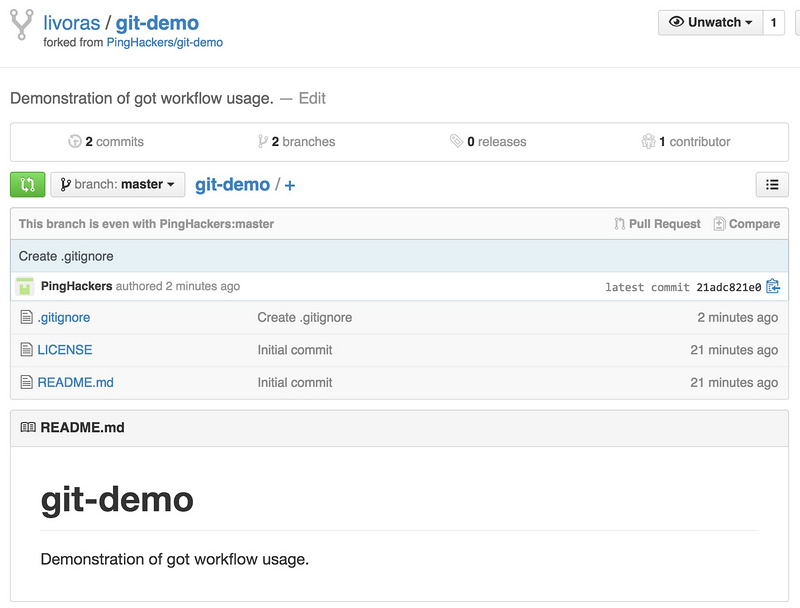


### Step 2：开发者fork源仓库

源仓库建立以后，每个开发就可以去复制一份源仓库到自己的github账号中，然后作为自己开发所用的仓库。假设我是一个项目中的开发者，我就到[PingHackers/git-demo](https://github.com/PingHackers/git-demo)项目主页上去fork：



fork完以后，我就可以在我自己的仓库列表中看到一个和源仓库一模一样的复制品。这时就应该感叹，你以后要和它相依为命了：



### Step 3：把自己开发者仓库clone到本地

这一步应该不用教，git clone

### Step 4：构建功能分支进行开发

进入仓库中，按照前面说所的构建功能分支的步骤，构建功能分支进行开发、合并，假设我现在要开发一个“讨论”功能：

>>> git checkout develop

# 切换到`develop`分支

>>> git checkout -b feature-discuss

# 分出一个功能性分支

>> touch discuss.js

# 假装discuss.js就是我们要开发的功能

>> git add .

>> git commit -m 'finish discuss feature'

# 提交更改

>>> git checkout develop

# 回到develop分支

>>> git merge --no-ff feature-discuss

# 把做好的功能合并到develop中

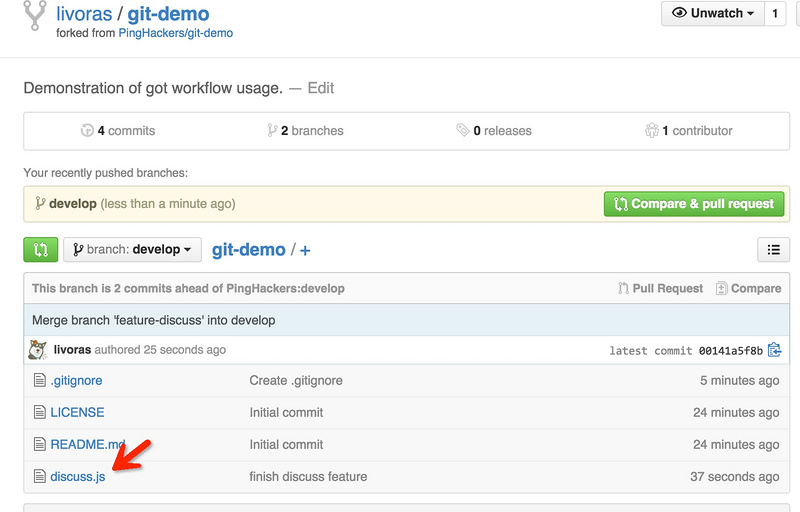
>>> git branch -d feature-discuss

# 删除功能性分支

>>> git push origin develop

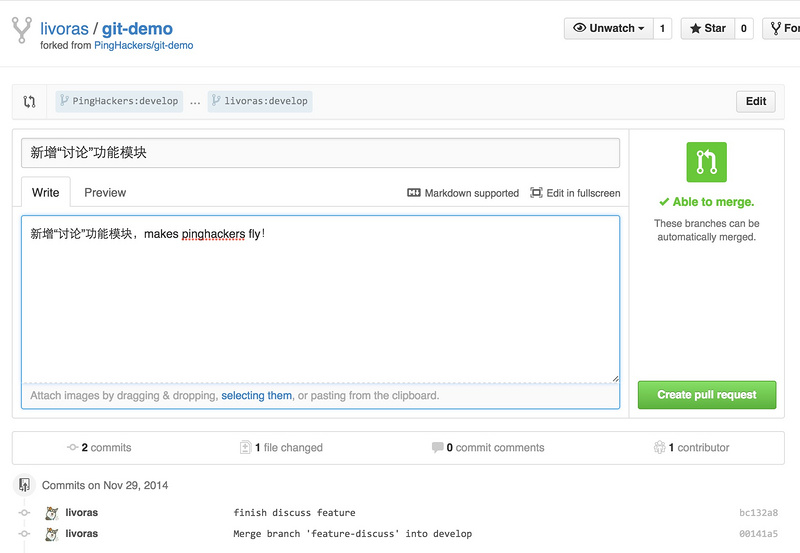
# 把develop提交到自己的远程仓库中

这时候，你上自己github的项目主页中develop分支中看看，已经有discuss.js这个文件了：

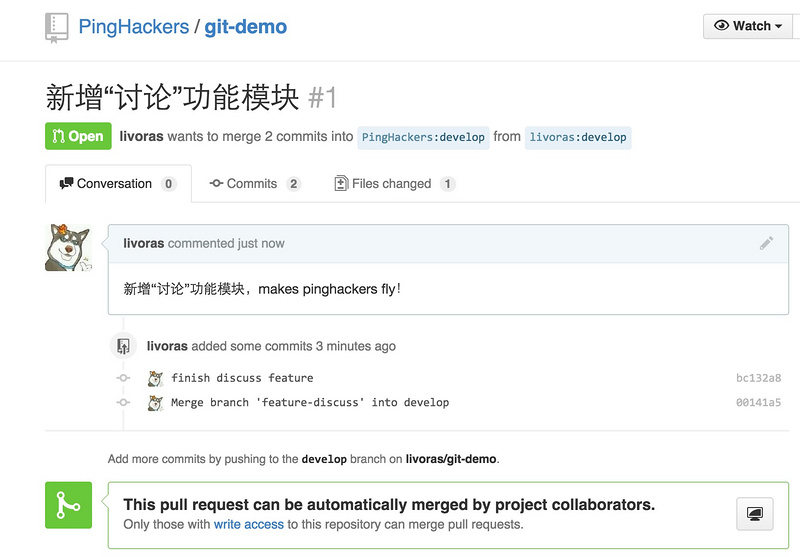


### Step 5：向管理员提交pull request

假设我完成了“讨论”功能（当然，你还可能对自己的develop进行了多次合并，完成了多个功能），经过测试以后，觉得没问题，就可以请求管理员把**自己仓库的develop分支**合并到**源仓库的develop**分支中，这就是传说中的pull request。

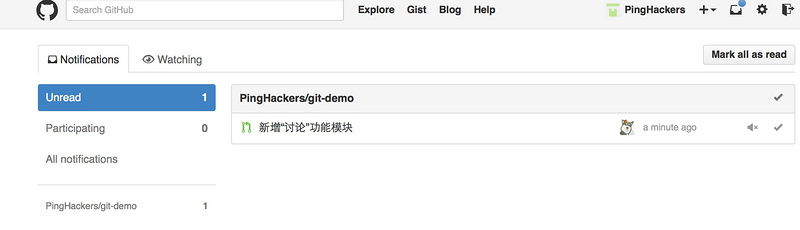


点击上图的绿色按钮，开发者就可以就可以静静地等待管理员对你的提交的评审了。



### Step 6 管理员测试、合并

接下来就是管理员的操作了，作为管理员的PingHackers登陆github，便看到了我对源仓库发起的pull request。



这时候PingHackers需要做的事情就是：

1. **对我的代码进行review**。github提供非常强大的代码review功能：  
   
2. **在他的本地测试新建一个测试分支**，测试我的代码：

>> git checkout develop

# 进入他本地的develop分支

>> git checkout -b livoras-develop

# 从develop分支中分出一个叫livoras-develop的测试分支测试我的代码

>> git pull https://github.com/livoras/git-demo.git develop

# 把我的代码pull到测试分支中，进行测试

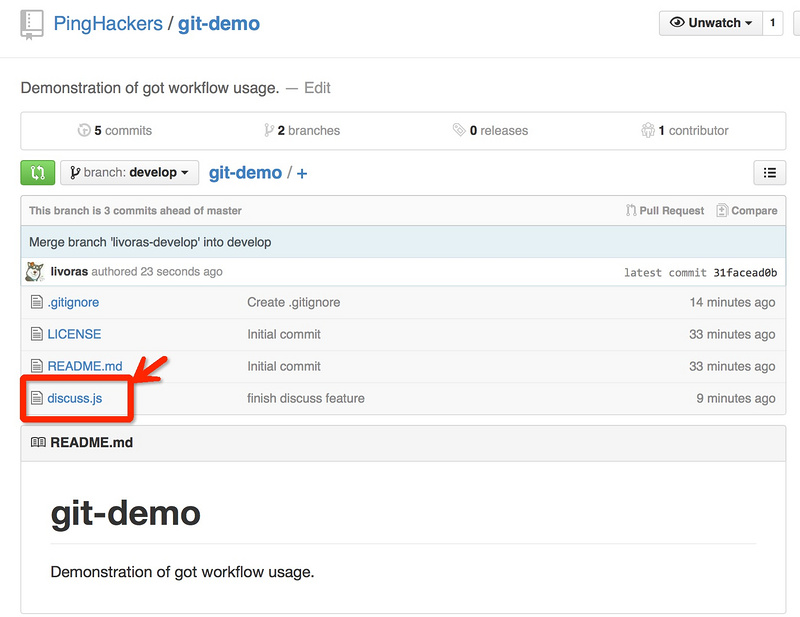
1. **判断是否同意合并到源仓库的develop中**，如果经过测试没问题，可以把我的代码合并到源仓库的develop中：

>> git checkout develop

>> git merge --no-ff livoras-develop

>> git push origin develop

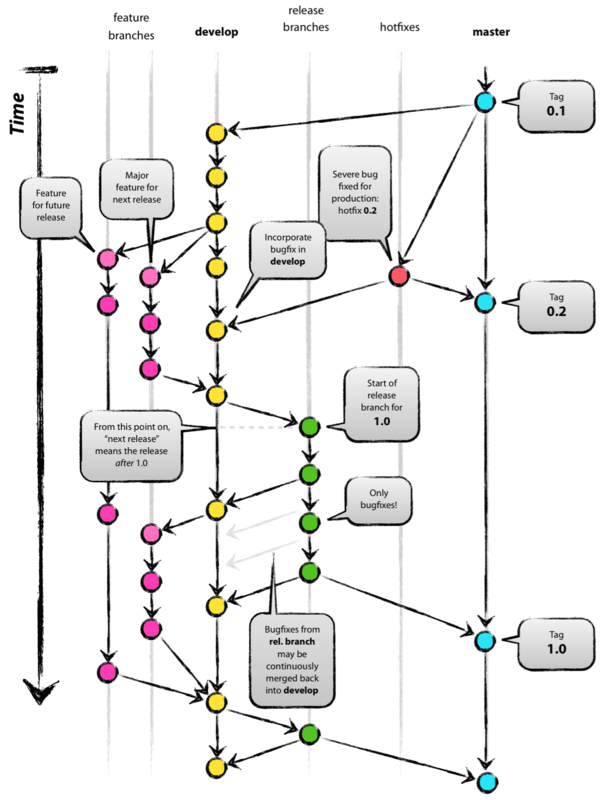
注意，PingHakers一直在操作的仓库是源仓库。所以我们经过上面一系列操作以后，就可以在源仓库主页中看到：



经过辗转曲折的路程，我们的discuss.js终于从我的开发仓库的功能分支到达了源仓库的develop分支中。以上，就是一个git & github协同工作流的基本步骤。

## 总结

git这一个工具博大精深，很难想象竟然有使用如此恶心而又如此灵活和优雅的工具存在；此又为一神器，大家还是多动手，多查资料，让git成为自己的一项基本技能，帮助自己处理各种项目团队协同工作的问题，成为一个高效的开发者、优秀的项目的管理者。送大家一张神图，好好领悟：



最后给出一些参考资料，供参考学习。

## 参考资料

* [A Successful Git Branching Model](http://nvie.com/posts/a-successful-git-branching-model/)

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

* [Understanding the Git Workflow](https://sandofsky.com/blog/git-workflow.html)

https://sandofsky.com/blog/git-workflow.html

* [Github flow](http://scottchacon.com/2011/08/31/github-flow.html)

http://scottchacon.com/2011/08/31/github-flow.html

* [Pro Git](http://git-scm.com/book/zh/)

http://git-scm.com/book/zh/

* [Git分支管理策略](http://www.ruanyifeng.com/blog/2012/07/git.html)

http://www.ruanyifeng.com/blog/2012/07/git.html

1. https://segmentfault.com/a/1190000002413519 [↑](#footnote-ref-1)