目录

介绍 0

Git 常用命令清单 1

Git 工作流 2

- 集中式工作流 2.1
- 功能分支工作流 2.2
- Gitflow 工作流 2.3
- Forking 工作流 2.4

Git 注意事项 3

git简介

Git 是一款免费的、开源的、分布式的版本控制系统。旨在快速高效地处理无论规模大小的任何软件工程。

每一个 Git克隆 都是一个完整的文件库，含有全部历史记录和修订追踪能力，不依赖于网络连接或中心服务器。其最大特色就是“分支”及“合并”操作非常快速、简便。

Git的一个重要特性就是对分布式开发的支持，我们看一个简单例子：假如你把开发任务从公司带回家，晚饭后突然有了灵感，要对代码进行修改，不巧家里的电脑不能连接到公司的文件库，你怎么下载要修改的文件？即使你将文件用优盘带回去了，那么修改之后，又怎样提交？SVN 对这种情况没有解决方案，而 Git 可以！因为 Git 在每个用户硬盘上都创建了完整的文件库，不需要、也不存在一个“中心服务器”，你只要能连接上任何一个团队成员的电脑，就能将代码提交到文件库去（有点像 P2P）。

从网上的用户评价来看，Git 最大的优势就是“快”！对于大型的联合开发项目，用 SVN 进行版本控制管理时速度很慢，但是用 Git 就快很多。还有一个说法是：SVN 有的功能 Git 都有，而 Git 的某些特色 SVN 根本做不到。

简单来说就是：高端大气上档次！

Git for windows

下载地址

Git教程

点击查看

由于水平有限，文中错误之处在所难免，敬请指正！

编者：苦少

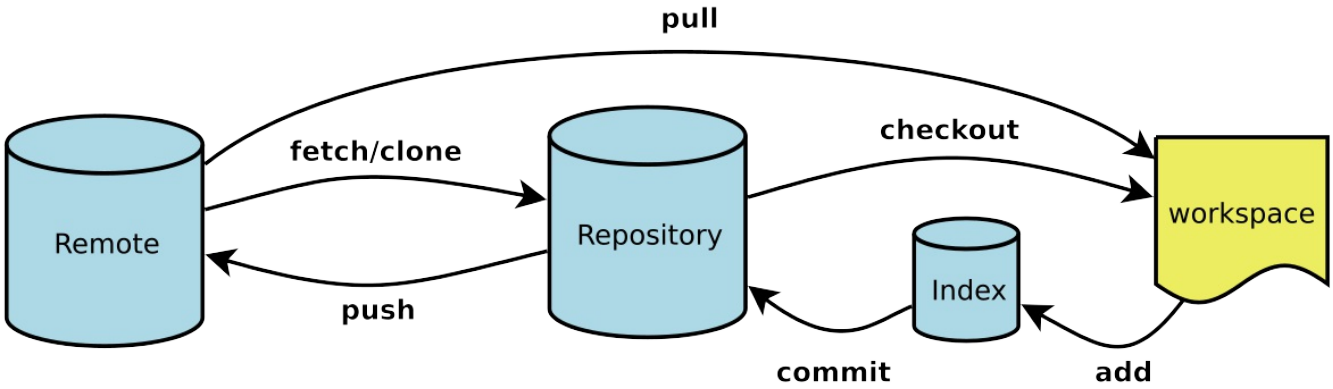
百度Hl：kkccjj_2011

邮箱：xiachunhui@hnjing.com

当前版本：v0.0.1

Git 常用命令清单

一般来说，日常使用只要记住下图6个命令，就可以了。但是熟练使用，恐怕要记住60～100个命令。



下面是我整理的常用 Git 命令清单。几个专用名词的译名如下。

> Workspace：工作区
>
> Index / Stage：暂存区
>
> Repository：仓库区（或本地仓库）
>
> Remote：远程仓库

新建代码库

> git-1.sh

```
# 在当前目录新建一个Git代码库
$ git init

# 新建一个目录，将其初始化为Git代码库
$ git init [project-name]

# 下载一个项目和它的整个代码历史
$ git clone [url]
```

配置

Git的设置文件为.gitconfig，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。

> git-2.sh

```
# 显示当前的Git配置
$ git config --list

# 编辑Git配置文件
$ git config -e [--global]

# 设置提交代码时的用户信息
$ git config [--global] user.name "[name]"
$ git config [--global] user.email "[email address]"
```

增加/删除文件

> git-3.sh

```
# 添加指定文件到暂存区
$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录
$ git add [dir]

# 添加当前目录的所有文件到暂存区
$ git add .

# 添加每个变化前，都会要求确认
# 对于同一个文件的多处变化，可以实现分次提交
$ git add -p

# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...

# 停止追踪指定文件，但该文件会保留在工作区
$ git rm --cached [file]

# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

代码提交

> git-4.sh

```
# 提交暂存区到仓库区
$ git commit -m [message]

# 提交暂存区的指定文件到仓库区
$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区
$ git commit -a

# 提交时显示所有diff信息
$ git commit -v

# 使用一次新的commit，替代上一次提交
# 如果代码没有任何新变化，则用来改写上一次commit的提交信息
$ git commit --amend -m [message]

# 重做上一次commit，并包括指定文件的新变化
$ git commit --amend [file1] [file2] ...
```

分支

> git-5.sh

```
# 列出所有本地分支
$ git branch

# 列出所有远程分支
$ git branch -r

# 列出所有本地分支和远程分支
$ git branch -a

# 新建一个分支，但依然停留在当前分支
$ git branch [branch-name]

# 新建一个分支，并切换到该分支
$ git checkout -b [branch]

# 新建一个分支，指向指定commit
$ git branch [branch] [commit]

# 新建一个分支，与指定的远程分支建立追踪关系
$ git branch --track [branch] [remote-branch]

# 切换到指定分支，并更新工作区
$ git checkout [branch-name]

# 切换到上一个分支
$ git checkout -

# 建立追踪关系，在现有分支与指定的远程分支之间
$ git branch --set-upstream [branch] [remote-branch]

# 合并指定分支到当前分支
$ git merge [branch]

# 选择一个commit，合并进当前分支
$ git cherry-pick [commit]

# 删除分支
$ git branch -d [branch-name]

# 删除远程分支
$ git push origin --delete [branch-name]
$ git branch -dr [remote/branch]
```

标签

> git-6.sh

```
# 列出所有tag
$ git tag

# 新建一个tag在当前commit
$ git tag [tag]

# 新建一个tag在指定commit
$ git tag [tag] [commit]

# 删除本地tag
$ git tag -d [tag]

# 删除远程tag
$ git push origin :refs/tags/[tagName]

# 查看tag信息
$ git show [tag]

# 提交指定tag
$ git push [remote] [tag]

# 提交所有tag
$ git push [remote] --tags

# 新建一个分支，指向某个tag
$ git checkout -b [branch] [tag]
```

查看信息

> git-7.sh

```
# 显示有变更的文件
$ git status

# 显示当前分支的版本历史
$ git log

# 显示commit历史，以及每次commit发生变更的文件
$ git log --stat

# 搜索提交历史，根据关键词
$ git log -S [keyword]

# 显示某个commit之后的所有变动，每个commit占据一行
$ git log [tag] HEAD --pretty=format:%s

# 显示某个commit之后的所有变动，其"提交说明"必须符合搜索条件
$ git log [tag] HEAD --grep feature

# 显示某个文件的版本历史，包括文件改名
$ git log --follow [file]
$ git whatchanged [file]

# 显示指定文件相关的每一次diff
```

```
$ git log -p [file]

# 显示过去5次提交
$ git log -5 --pretty --oneline

# 显示所有提交过的用户，按提交次数排序
$ git shortlog -sn

# 显示指定文件是什么人在什么时间修改过
$ git blame [file]

# 显示暂存区和工作区的差异
$ git diff

# 显示暂存区和上一个commit的差异
$ git diff --cached [file]

# 显示工作区与当前分支最新commit之间的差异
$ git diff HEAD

# 显示两次提交之间的差异
$ git diff [first-branch]...[second-branch]

# 显示今天你写了多少行代码
$ git diff --shortstat "@{0 day ago}"

# 显示某次提交的元数据和内容变化
$ git show [commit]

# 显示某次提交发生变化的文件
$ git show --name-only [commit]

# 显示某次提交时，某个文件的内容
$ git show [commit]:[filename]

# 显示当前分支的最近几次提交
$ git reflog
```

远程同步

> git-8.sh

```
# 下载远程仓库的所有变动
$ git fetch [remote]

# 显示所有远程仓库
$ git remote -v

# 显示某个远程仓库的信息
$ git remote show [remote]

# 增加一个新的远程仓库，并命名
$ git remote add [shortname] [url]

# 取回远程仓库的变化，并与本地分支合并
$ git pull [remote] [branch]

# 上传本地指定分支到远程仓库
$ git push [remote] [branch]

# 强行推送当前分支到远程仓库，即使有冲突
$ git push [remote] --force

# 推送所有分支到远程仓库
$ git push [remote] --all
```

撤销

> git-9.sh

```
# 恢复暂存区的指定文件到工作区
$ git checkout [file]

# 恢复某个commit的指定文件到暂存区和工作区
$ git checkout [commit] [file]

# 恢复暂存区的所有文件到工作区
$ git checkout .

# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变
$ git reset [file]

# 重置暂存区与工作区，与上一次commit保持一致
$ git reset --hard

# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变
$ git reset [commit]

# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致
$ git reset --hard [commit]

# 重置当前HEAD为指定commit，但保持暂存区和工作区不变
$ git reset --keep [commit]

# 新建一个commit，用来撤销指定commit
# 后者的所有变化都将被前者抵消，并且应用到当前分支
$ git revert [commit]

# 暂时将未提交的变化移除，稍后再移入
$ git stash
$ git stash pop
```

其他

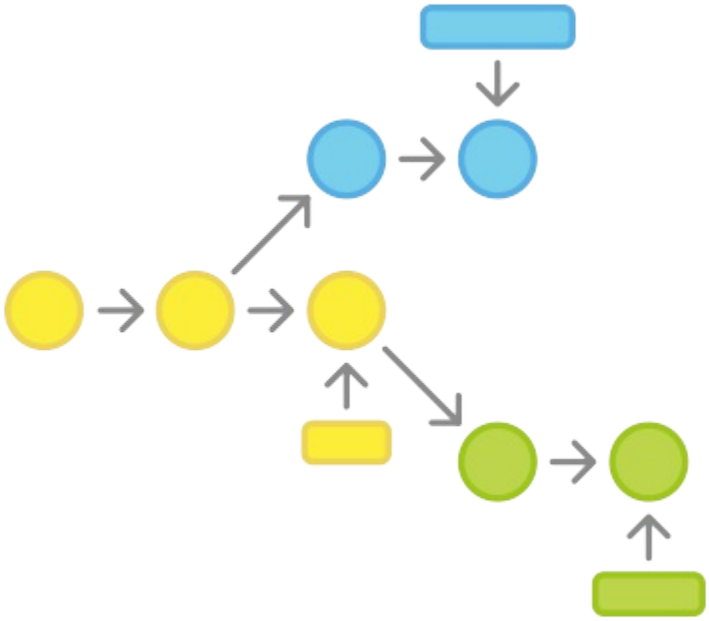
> git-10.sh

```
# 生成一个可供发布的压缩包
$ git archive
```

Git 工作流

工作流有各式各样的用法，但也正因此使得在实际工作中如何上手使用变得很头大。这篇指南通过总览公司团队中最常用的几种 `Git` 工作流让大家可以上手使用。

在阅读的过程中请记住，本文中的几种工作流是作为方案指导而不是条例规定。在展示了各种工作流可能的用法后，你可以从不同的工作流中挑选或揉合出一个满足你自己需求的工作流。

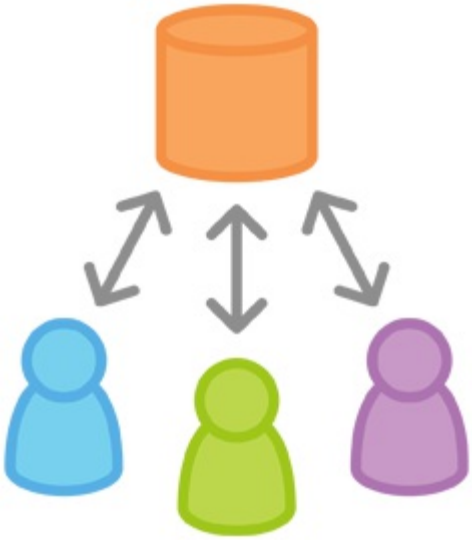


概述

集中式工作流

如果你的开发团队成员已经很熟悉 `Subversion`，集中式工作流让你无需去适应一个全新流程就可以体验 `Git` 带来的收益。这个工作流也可以作为向更 `Git` 风格工作流迁移的友好过渡。

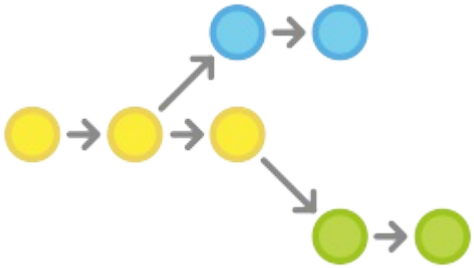
了解更多 »



功能分支工作流

功能分支工作流以集中式工作流为基础，不同的是为各个新功能分配一个专门的分支来开发。这样可以在把新功能集成到正式项目前，用`Pull Requests`的方式讨论变更。

了解更多 »



Gitflow 工作流

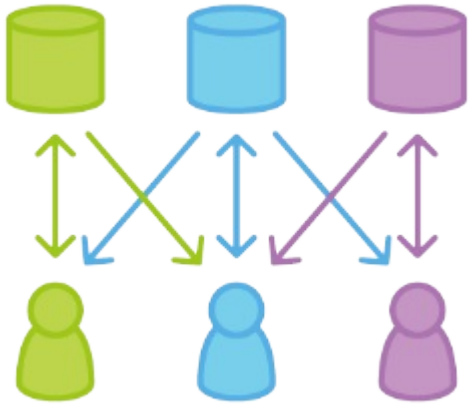
Gitflow工作流通过为功能开发、发布准备和维护分配独立的分支，让发布迭代过程更流畅。严格的分支模型也为大型项目提供了一些非常必要的结构。

了解更多 »

Forking 工作流

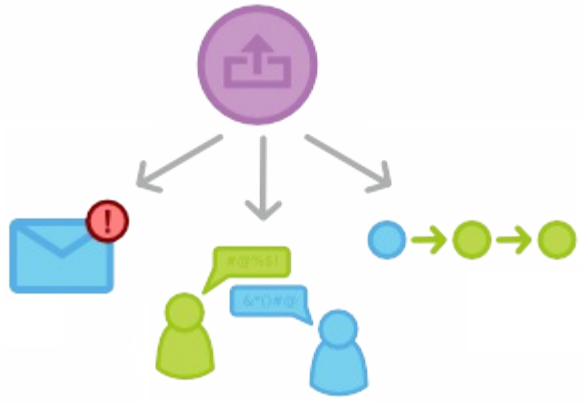
`Forking` 工作流是分布式工作流，充分利用了 `Git` 在分支和克隆上的优势。可以安全可靠地管理大团队的开发者（ `developer` ），并能接受不信任贡献者（ `contributor` ）的提交。

了解更多 »

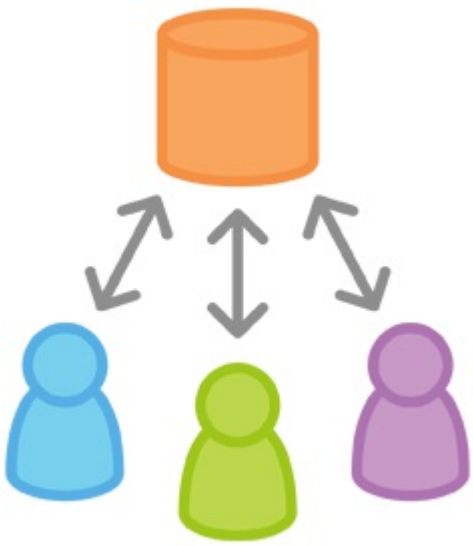


`Pull Requests`

`Pull requests` 是 `Bitbucket` 提供的让开发者更方便地进行协作的功能，提供了友好的 `Web` 界面可以在提议的修改合并到正式项目之前对修改进行讨论。



集中式工作流



转到分布式版本控制系统看起来像个令人生畏的任务，但不改变已用的工作流你也可以用上 `Git` 带来的收益。团队可以用和 `Subversion` 完全不变的方式来开发项目。

但使用 `Git` 加强开发的工作流， `Git` 有相比 `SVN` 的几个优势。 首先，每个开发可以有属于自己的整个工程的本地拷贝。隔离的环境让各个开发者的工作和项目的其他部分修改独立开来 —— 即自由地提交到自己的本地仓库，先完全忽略上游的开发，直到方便的时候再把修改反馈上去。

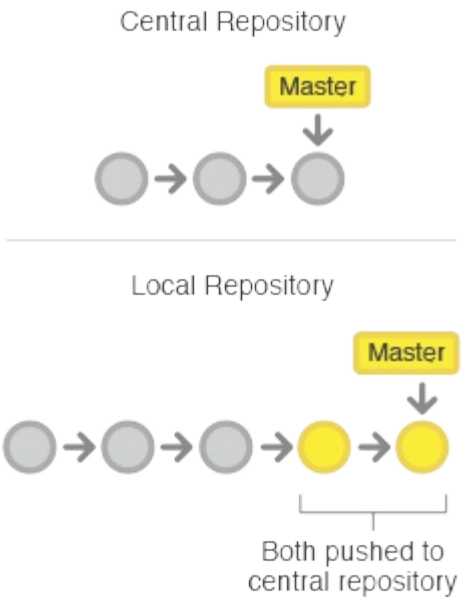
其次， `Git` 提供了强壮的分支和合并模型。不像 `SVN` ， `Git` 的分支设计成可以做为一种用来在仓库之间集成代码和分享修改的『失败安全』的机制。

工作方式

像 `Subversion` 一样，集中式工作流以中央仓库作为项目所有修改的单点实体。相比 `SVN` 缺省的开发分支 `trunk` ， `Git` 叫做 `master` ，所有修改提交到这个分支上。本工作流只用到 `master` 这一个分支。

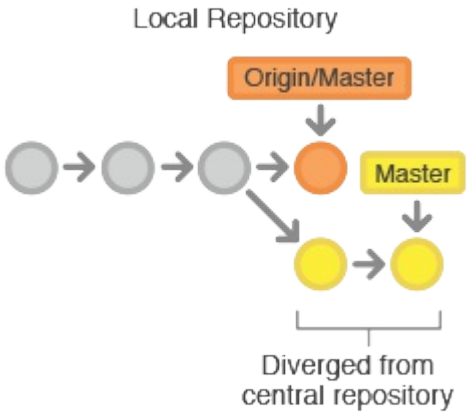
开发者开始先克隆中央仓库。在自己的项目拷贝中像 `SVN` 一样的编辑文件和提交修改；但修改是存在本地的，和中央仓库是完全隔离的。开发者可以把和上游的同步延后到一个方便时间点。

要发布修改到正式项目中，开发者要把本地 `master` 分支的修改『推』到中央仓库中。这相当于 `svn commit` 操作，但 `push` 操作会把所有还不在中央仓库的本地提交都推上去。



冲突解决

中央仓库代表了正式项目，所以提交历史应该被尊重且是稳定不变的。如果开发者本地的提交历史和中央仓库有分歧，`Git` 会拒绝 `push` 提交否则会覆盖已经在中央库的正式提交。



在开发者提交自己功能修改到中央库前，需要先 `fetch` 在中央库的新增提交， `rebase` 自己提交到中央库提交历史之上。 这样做的意思是在说，『我要把自己的修改加到别人已经完成的修改上。』最终的结果是一个完美的线性历史，就像以前的 `SVN` 的工作流中一样。

如果本地修改和上游提交有冲突， `Git` 会暂停 `rebase` 过程，给你手动解决冲突的机会。 `Git` 解决合并冲突，用和生成提交一样的 `git status` 和 `git add` 命令，很一致方便。还有一点，如果解决冲突时遇到麻烦， `Git` 可以很简单中止整个 `rebase` 操作，重来一次（或者让别人来帮助解决）。

示例

让我们一起逐步分解来看看一个常见的小团队如何用这个工作流来协作的。有两个开发者小明和小红，看他们是如何开发自己的功能并提交到中央仓库上的。

有人先初始化好中央仓库

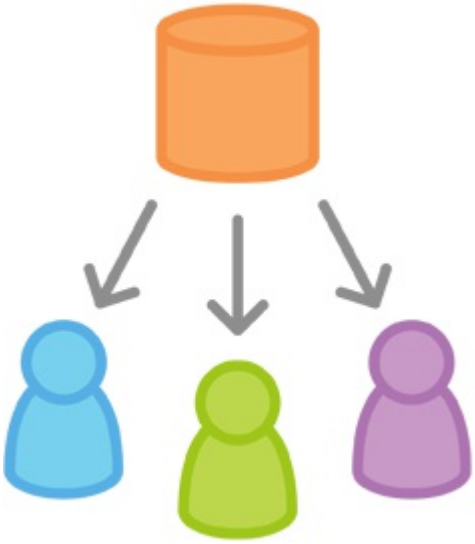
第一步，有人在服务器上创建好中央仓库。如果是新项目，你可以初始化一个空仓库；否则你要导入已有的 `Git` 或 `SVN` 仓库。

中央仓库应该是个裸仓库（ `bare repository` ），即没有工作目录（ `working directory` ）的仓库。可以用下面的命令创建：

```
ssh user@host
git init --bare /path/to/repo.git
```

确保写上有效的 `user` （ `SSH` 的用户名）， `host` （服务器的域名或IP地址）， `/path/to/repo.git` （你想存放仓库的位置）。 注意，为了表示是一个裸仓库，按照约定加上 `.git` 扩展名到仓库名上。

所有人克隆中央仓库

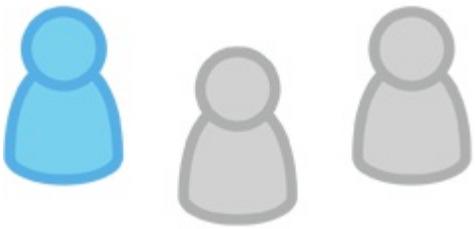


下一步，各个开发者创建整个项目的本地拷贝。通过 `git clone` 命令完成：

```
git clone ssh://user@host/path/to/repo.git
```

基于你后续会持续和克隆的仓库做交互的假设，克隆仓库时 `Git` 会自动添加远程别名 `origin` 指回『父』仓库。

小明开发功能



在小明的本地仓库中，他使用标准的 `Git` 过程开发功能：编辑、暂存（ `Stage` ）和提交。 如果你不熟悉暂存区（ `Staging Area` ），这里说明一下：暂存区 用来准备一个提交，但可以不用把工作目录中所有的修改内容都包含进来。 这样你可以创建一个高度聚焦的提交，尽管你本地修改很多内容。

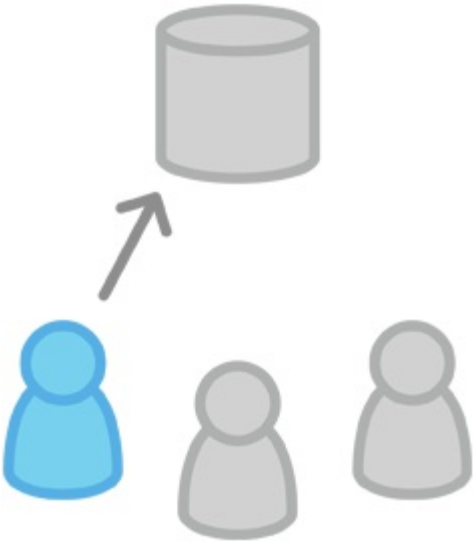
```
git status # 查看本地仓库的修改状态
git add # 暂存文件
git commit # 提交文件
```

请记住，因为这些命令生成的是本地提交，小明可以按自己需求反复操作多次，而不用担心中央仓库上有了什么操作。 对需要多个更简单更原子分块的大功能，这个做法是很有用的。

小红开发功能

与此同时，小红在自己的本地仓库中用相同的编辑、暂存和提交过程开发功能。和小明一样，她也不关心中央仓库有没有新提交； 当然更不关心小明在他的本地仓库中的操作，因为所有本地仓库都是私有的。

小明发布功能

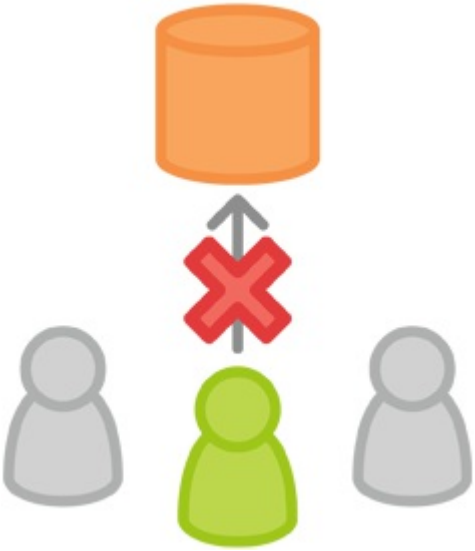


一旦小明完成了他的功能开发，会发布他的本地提交到中央仓库中，这样其它团队成员可以看到他的修改。他可以用下面的 `git push` 命令：

```
git push

# 【译注】：
# 原文用的命令是 git push origin master
#
# 主流的git版本，可以省略后面2个参数：远程仓库别名、推送分支，
# 因为这2个参数缺省分别就是 origin 、 当前分支（本文目前的示例就是master）。
# 这样的用法更简单自然，我平时就是这么用的。
```

注意， `origin` 是在小明克隆仓库时 `Git` 创建的远程中央仓库别名。 `master` 参数告诉 `Git` 推送的分支。 由于中央仓库自从小明克隆以来还没有被更新过，所以 `push` 操作不会有冲突，成功完成。

小红试着发布功能



一起来看看在小明发布修改后，小红 `push` 修改会怎么样？她使用完全一样的 `push` 命令：

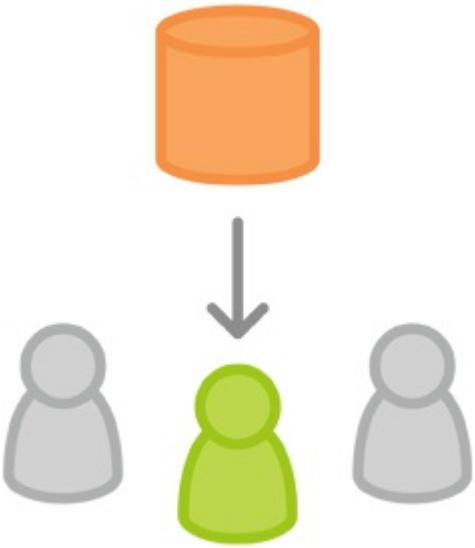
```
git push

# 【译注】：
# 原文用的命令是 git push origin master
# 原因同上
```

但她的本地历史已经和中央仓库有分岐了，`Git` 拒绝操作并给出下面很长的出错消息：

```
error: failed to push some refs to '/path/to/repo.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

这避免了小红覆写正式的提交。她要先 `pull` 小明的更新到她的本地仓库合并上她的本地修改后，再重试。

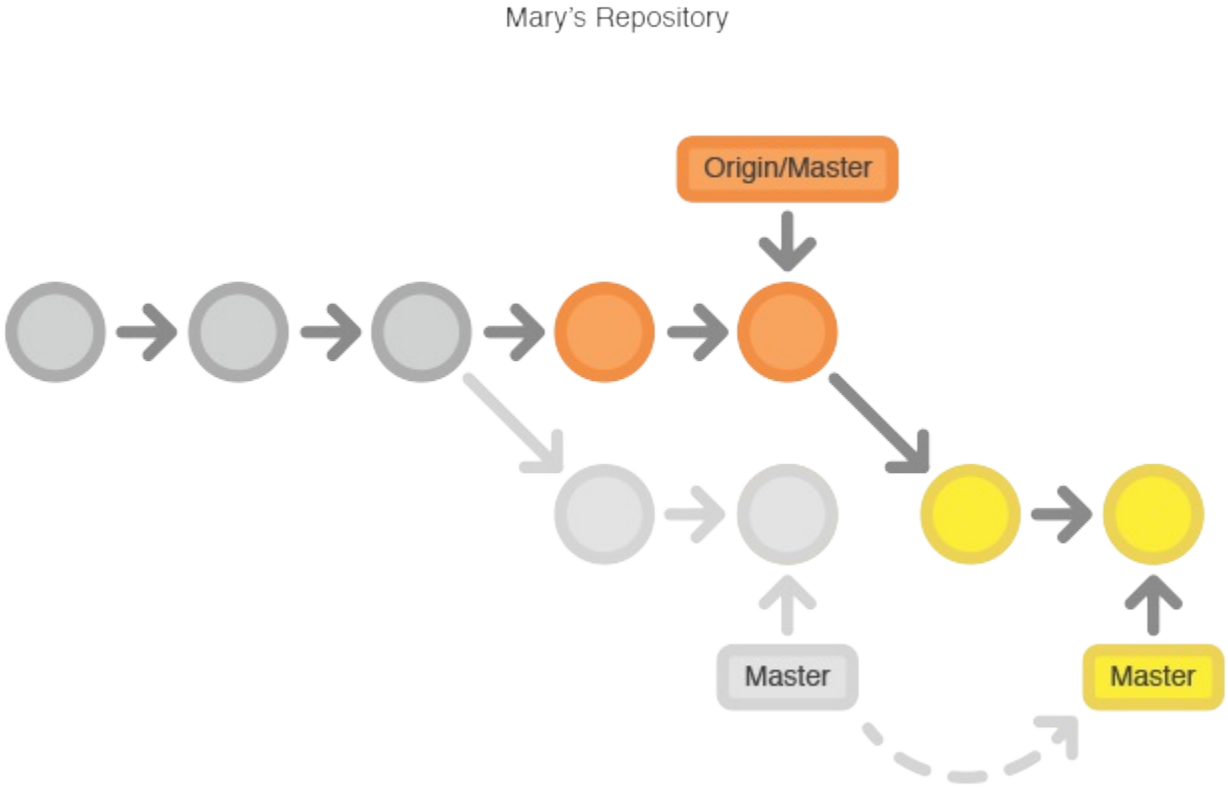
小红在小明的提交之上 `rebase`



小红用 `git pull` 合并上游的修改到自己的仓库中。 这条命令类似 `svn update` ——拉取所有上游提交命令到小红的本地仓库，并尝试和她的本地修改合并：

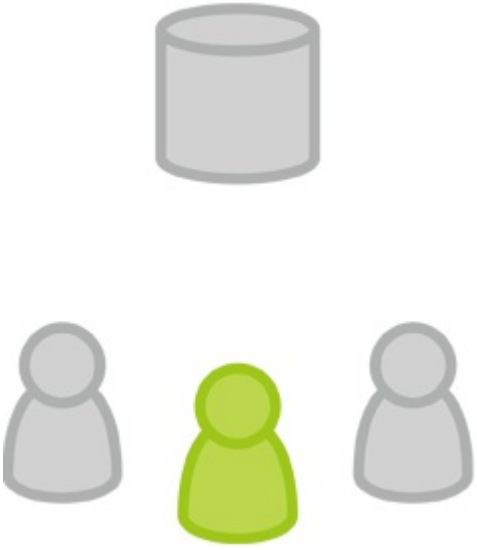
```
git pull --rebase

# 【译注】：
# 原文用的命令是 git pull --rebase origin master
# 原因同上
```

`--rebase` 选项告诉 `Git` 把小红的提交移到同步了中央仓库修改后的 `master` 分支的顶部，如下图所示：



如果你忘加了这个选项，`pull` 操作仍然可以完成，但每次 `pull` 操作要同步中央仓库中别人修改时，提交历史会以一个多余的『合并提交』结尾。 对于集中式工作流，最好是使用 `rebase` 而不是生成一个合并提交。

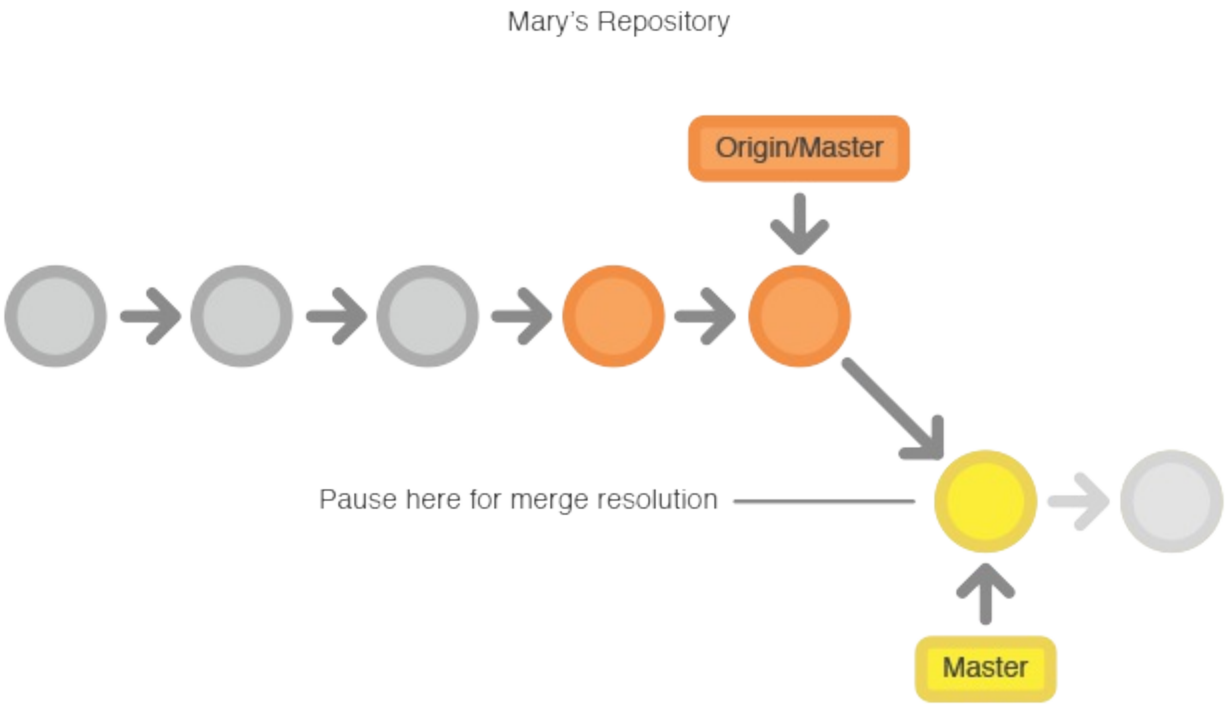
小红解决合并冲突



`rebase` 操作过程是把本地提交一次一个地迁移到更新了的中央仓库 `master` 分支之上。 这意味着可能要解决在迁移某个提交时出现的合并冲突，而不是解决包含了所有提交的大型合并时所出现的冲突。 这样的方式让你尽可能保持每个提交的聚焦和项目历史的整洁。反过来，简化了哪里引入 `Bug` 的分析，如果有必要，回滚修改也可以做到对项目影响最小。

如果小红和小明的功能是不相关的，不大可能在 `rebase` 过程中有冲突。如果有， `Git` 在合并有冲突的提交处暂停 `rebase` 过程，输出下面的信息并带上相关的指令：

```
CONFLICT (content): Merge conflict in <some-file>
```



`Git` 很赞的一点是，任何人可以解决他自己的冲突。在这个例子中，小红可以简单的运行 `git status` 命令来查看哪里有问题。 冲突文件列在 `Unmerged paths` （未合并路径）一节中：

```
# Unmerged paths:
# (use "git reset HEAD <some-file>..." to unstage)
# (use "git add/rm <some-file>..." as appropriate to mark resolution)
#
# both modified: <some-file>
```

接着小红编辑这些文件。修改完成后，用老套路暂存这些文件，并让 `git rebase` 完成剩下的事：

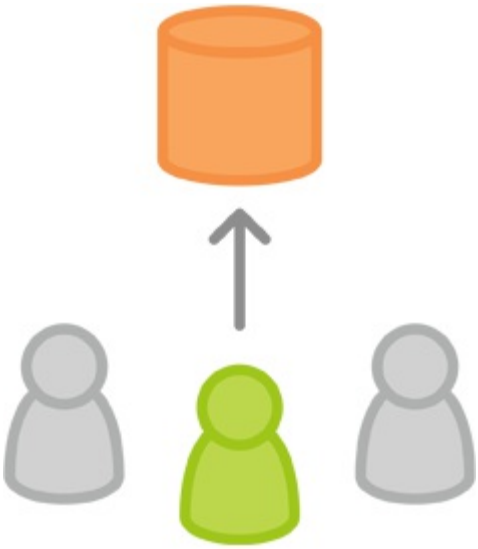
```
git add <some-file>
git rebase --continue
```

要做的就这些了。 `Git` 会继续一个一个地合并后面的提交，如其它的提交有冲突就重复这个过程。

如果你碰到了冲突，但发现搞不定，不要惊慌。只要执行下面这条命令，就可以回到你执行 `git pull --rebase` 命令前的样子：

```
git rebase --abort
```

小红成功发布功能



小红完成和中央仓库的同步后，就能成功发布她的修改了：

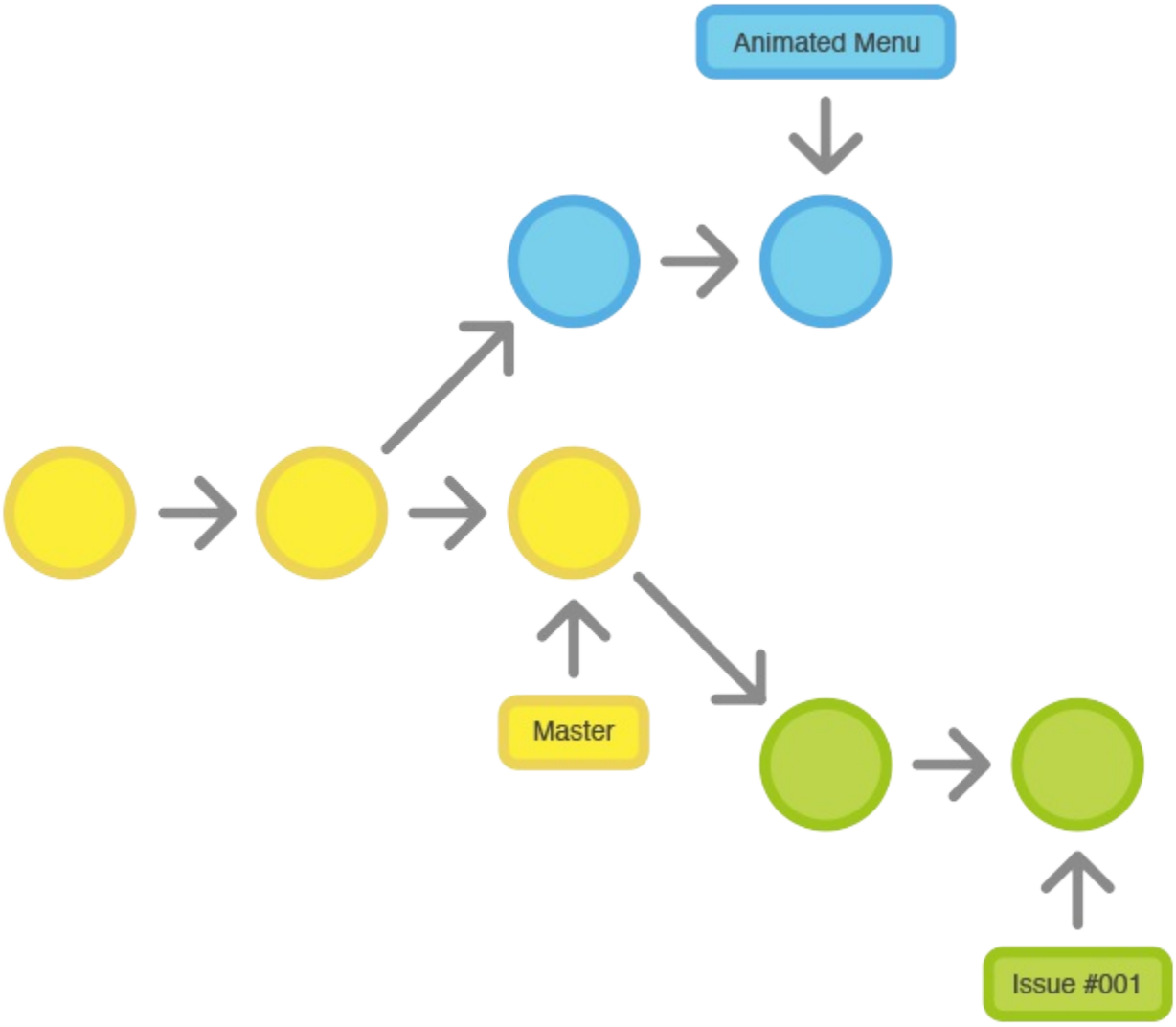
```
git push

# 【译注】：
# 原文用的命令是 git push origin master
# 原因同上
```

:beer: 下一站

如你所见，仅使用几个 `Git` 命令我们就可以模拟出传统 `Subversion` 开发环境。对于要从 `SVN` 迁移过来的团队来说这太好了，但没有发挥出 `Git` 分布式本质的优势。

如果你的团队适应了集中式工作流，但想要更流畅的协作效果，绝对值得探索一下功能分支工作流的收益。通过为一个功能分配一个专门的分支，能够做到一个新增功能集成到正式项目之前对新功能进行深入讨论。

功能分支工作流



一旦你玩转了集中式工作流，在开发过程中可以很简单地加上功能分支，用来鼓励开发者之间协作和简化交流。

功能分支工作流背后的核心思路是所有的功能开发应该在一个专门的分支，而不是在 `master` 分支上。 这个隔离可以方便多个开发者在各自的功能上开发而不会弄乱主干代码。 另外，也保证了 `master` 分支的代码一定不会是有问题的，极大有利于集成环境。

功能开发隔离也让 `pull requests` 工作流成为可能， `pull requests` 工作流能为每个分支发起一个讨论，在分支合入正式项目之前，给其它开发者有表示赞同的机会。 另外，如果你在功能开发中有问题卡住了，可以开一个 `pull requests` 来向同学们征求建议。 这些做法的重点就是， `pull requests` 让团队成员之间互相评论工作变成非常方便！

工作方式

功能分支工作流仍然用中央仓库，并且 `master` 分支还是代表了正式项目的历史。 但不是直接提交本地历史到各自的本地 `master` 分支，开发者每次在开始新功能前先创建一个新分支。 功能分支应该有个有描述性的名字，比如 `animated-menu-items` 或 `issue-#1061` ，这样可以让分支有个清楚且高聚焦的用途。

在 `master` 分支和功能分支之间， `Git` 是没有技术上的区别，所以开发者可以用和集中式工作流中完全一样的方式编辑、暂存和提交修改到功能分支上。

另外，功能分支也可以（且应该） `push` 到中央仓库中。这样不修改正式代码就可以和其它开发者分享提交的功能。 由于 `master` 是仅有的一个『特殊』分支，在中央仓库上存多个功能分支不会有任何问题。当然，这样做也可以很方便地备份各自的本地提交。

Pull Requests

功能分支除了可以隔离功能的开发，也使得通过 `Pull Requests` 讨论变更成为可能。 一旦某个开发完成一个功能，不是立即合并到 `master` ，而是 `push` 到中央仓库的功能分支上并发起一个 `Pull Request` 请求去合并修改到 `master` 。 在修改成为主干代码前，这让其它的开发者有机会先去 `Review` 变更。

`Code Review` 是 `Pull Requests` 的一个主要的收益，但 `Pull Requests` 实际上用作讨论代码的常用手段。 你可以把 `Pull Requests` 作为专门给某个分支的讨论。这意味着可以在更早的开发过程中就可以进行 `Code Review` 。 比如，一个开发者开发功能需要帮助时，要做的就是发起一个 `Pull Request` ，相关的人就会自动收到通知，在相关的提交旁边能看到需要帮助解决的问题。

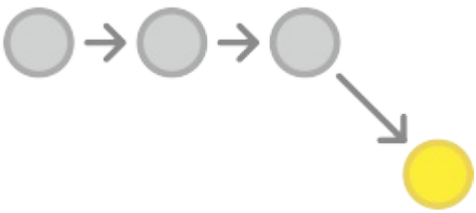
一旦 `Pull Request` 被接受了，发布功能要做的就和集中式工作流就很像了。 首先，确定本地的 `master` 分支和上游的 `master` 分支是同步的。然后合并功能分支到本地 `master` 分支并 `push` 已经更新的本地 `master` 分支到中央仓库。

仓库管理的产品解决方案像 `Bitbucket` 或 `Stash` ，可以良好地支持 `Pull Requests` 。可以看看 `Stash` 的 `Pull Requests` 文档。

示例

下面的示例演示了如何把 `Pull Requests` 作为 `Code Review` 的方式，但注意 `Pull Requests` 可以用于很多其它的目的。

小红开始开发一个新功能



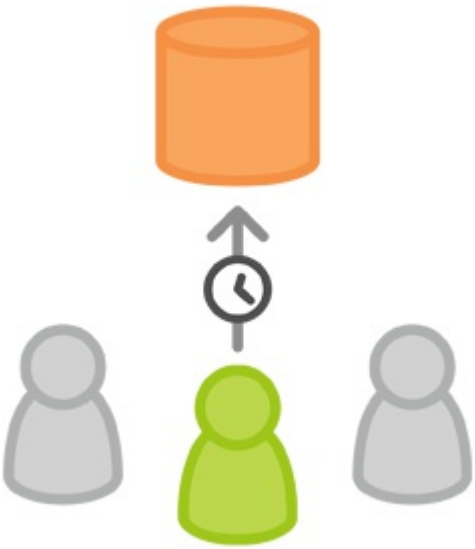
在开始开发功能前，小红需要一个独立的分支。使用下面的命令新建一个分支：

```
git checkout -b marys-feature

# 【译注】：
# 原文用的命令是 git checkout -b marys-feature master
#
# 主流的git版本，可以省略后一个参数：基于哪个分支新建分支，
# 因为这个参数缺省就是 当前分支（本文目前的示例就是master）。
# 这样的用法更简单自然，我平时就是这么用的。
```

这个命令检出一个基于 `master` 名为 `marys-feature` 的分支， `Git` 的 `-b` 选项表示如果分支还不存在则新建分支。 这个新分支上，小红按老套路编辑、暂存和提交修改，按需要提交以实现功能：

```
git status
git add <some-file>
git commit
```

小红要去吃个午饭



早上小红为新功能添加一些提交。 去吃午饭前， `push` 功能分支到中央仓库是很好的做法，这样可以方便地备份，如果和其它开发协作，也让他们可以看到小红的提交。

```
git push -u origin marys-feature
```

这条命令 `push` `marys-feature` 分支到中央仓库（ `origin` ）， `-u` 选项设置本地分支去跟踪远程对应的分支。设置好跟踪的分支后，小红就可以使用 `git push` 命令省去指定推送分支的参数。

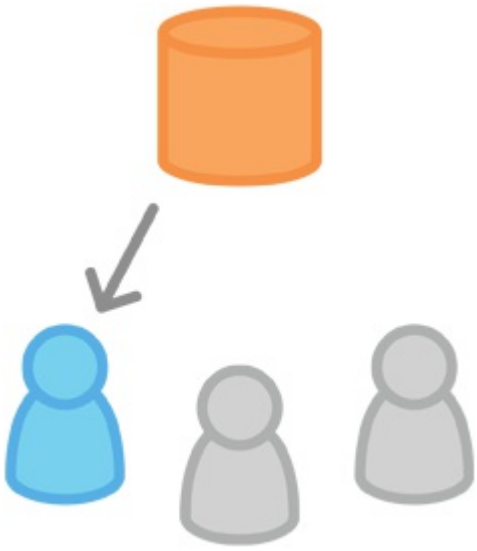
小红完成功能开发

小红吃完午饭回来，完成整个功能的开发。在合并到 `master` 之前， 她发起一个 `Pull Request` 让团队的其它人知道功能已经完成。但首先，她要确认中央仓库中已经有她最近的提交：

```
git push
```

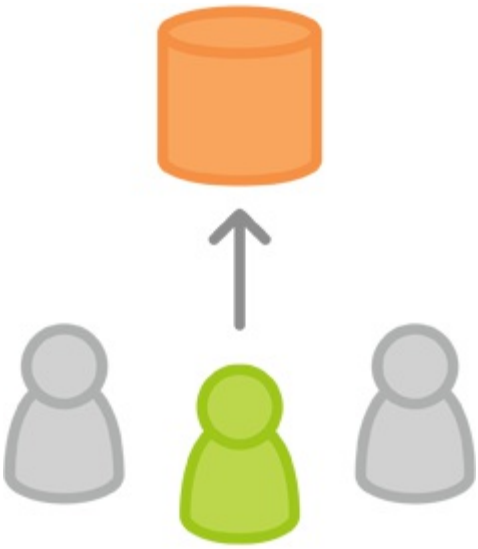
然后，在她的 `Git GUI` 客户端中发起 `Pull Request` ，请求合并 `marys-feature` 到 `master` ，团队成员会自动收到通知。 `Pull Request` 很酷的是可以在相关的提交旁边显示评注，所以你可以对某个变更集提问。

小黑收到 `Pull Request`



小黑收到了 `Pull Request` 后会查看 `marys-feature` 的修改。决定在合并到正式项目前是否要做些修改，且通过 `Pull Request` 和小红来回地讨论。

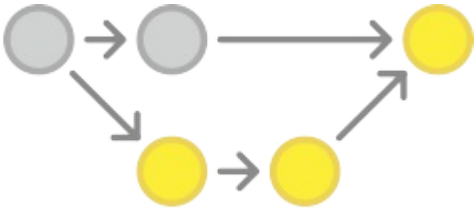
小红再做修改



要再做修改，小红用和功能第一个迭代完全一样的过程。编辑、暂存、提交并 `push` 更新到中央仓库。小红这些活动都会显示在 `Pull Request` 上，小黑可以断续做评注。

如果小黑有需要，也可以把 `marys-feature` 分支拉到本地，自己来修改，他加的提交也会一样显示在 `Pull Request` 上。

小红发布她的功能



一旦小黑可以接受 `Pull Request` ，就可以合并功能到稳定项目代码中（可以由小黑或是小红来做这个操作）：

```
git checkout master
git pull
git pull origin marys-feature
git push
```

无论谁来做合并，首先要检出 `master` 分支并确认它是最新的。然后执行 `git pull origin marys-feature` 合并 `marys-feature` 分支到已经和远程一致的本地 `master` 分支。 你可以使用简单 `git merge marys-feature` 命令，但前面的命令可以保证总是最新的新功能分支。 最后更新的 `master` 分支要重新 `push` 回到 `origin` 。

这个过程常常会生成一个合并提交。有些开发者喜欢有合并提交，因为它像一个新功能和原来代码基线的连通符。 但如果你偏爱线性的提交历史，可以在执行合并时 `rebase` 新功能到 `master` 分支的顶部，这样生成一个快进（ `fast-forward` ）的合并。

【译注】生成一个快进的合并的命令行：（感觉步骤有些多:sweat:，欢迎给出更简捷的做法:two_hearts:）

```
git checkout marys-feature
git pull # 确认是最新的
git pull --rebase origin master # rebase新功能到master分支的顶部

git checkout master
git merge marys-feature # 合并marys-feature分支的修改，因为这个分支之前对齐（rebase）了master，一定是快进合并
git push
```

一些 `GUI` 客户端可以只要点一下『接受』按钮执行好上面的命令来自动化 `Pull Request` 接受过程。 如果你的不能这样，至少在功能合并到 `master` 分支后能自动关闭 `Pull Request` 。

与此同时，小明在做和小红一样的事

当小红和小黑在 `marys-feature` 上工作并讨论她的 `Pull Request` 的时候，小明在自己的功能分支上做完全一样的事。

通过隔离功能到独立的分支上，每个人都可以自主的工作，当然必要的时候在开发者之间分享变更还是比较繁琐的。

下一站

到了这里，但愿你发现了功能分支可以很直接地在集中式工作流的仅有的 `master` 分支上完成多功能的开发。另外，功能分支还使用了 `Pull Request` ，使得可以在你的版本控制 `GUI` 客户端中讨论某个提交。

功能分支工作流是开发项目异常灵活的方式。问题是，有时候太灵活了。对于大型团队，常常需要给不同分支分配一个更具体的角色。 `Gitflow` 工作流是管理功能开发、发布准备和维护的常用模式。

`Gitflow` 工作流

这节介绍的 `Gitflow` 工作流借鉴自在nvie的*Vincent Driessen*。

`Gitflow` 工作流定义了一个围绕项目发布的严格分支模型。虽然比功能分支工作流复杂几分，但提供了用于一个健壮的用于管理大型项目的框架。

`Gitflow` 工作流没有用超出功能分支工作流的概念和命令，而是为不同的分支分配一个很明确的角色，并定义分支之间如何和什么时候进行交互。 除了使用功能分支，在做准备、维护和记录发布也使用各自的分支。当然你可以用上功能分支工作流所有的好处： `Pull Requests` 、隔离实验性开发和更高效的协作。

工作方式

`Gitflow` 工作流仍然用中央仓库作为所有开发者的交互中心。和其它的工作流一样，开发者在本地工作并 `push` 分支到要中央仓库中。

历史分支

相对使用仅有的一个 `master` 分支， `Gitflow` 工作流使用2个分支来记录项目的历史。 `master` 分支存储了正式发布的历史，而 `develop` 分支作为功能的集成分支。 这样也方便 `master` 分支上的所有提交分配一个版本号。

剩下要说明的问题围绕着这2个分支的区别展开。

功能分支

每个新功能位于一个自己的分支，这样可以 `push` 到中央仓库以备份和协作。 但功能分支不是从 `master` 分支上拉出新分支，而是使用 `develop` 分支作为父分支。当新功能完成时，合并回 `develop` 分支。 新功能提交应该从不直接与 `master` 分支交互。

注意，从各种含义和目的上来看，功能分支加上 `develop` 分支就是功能分支工作流的用法。但 `Gitflow` 工作流没有在这里止步。

发布分支

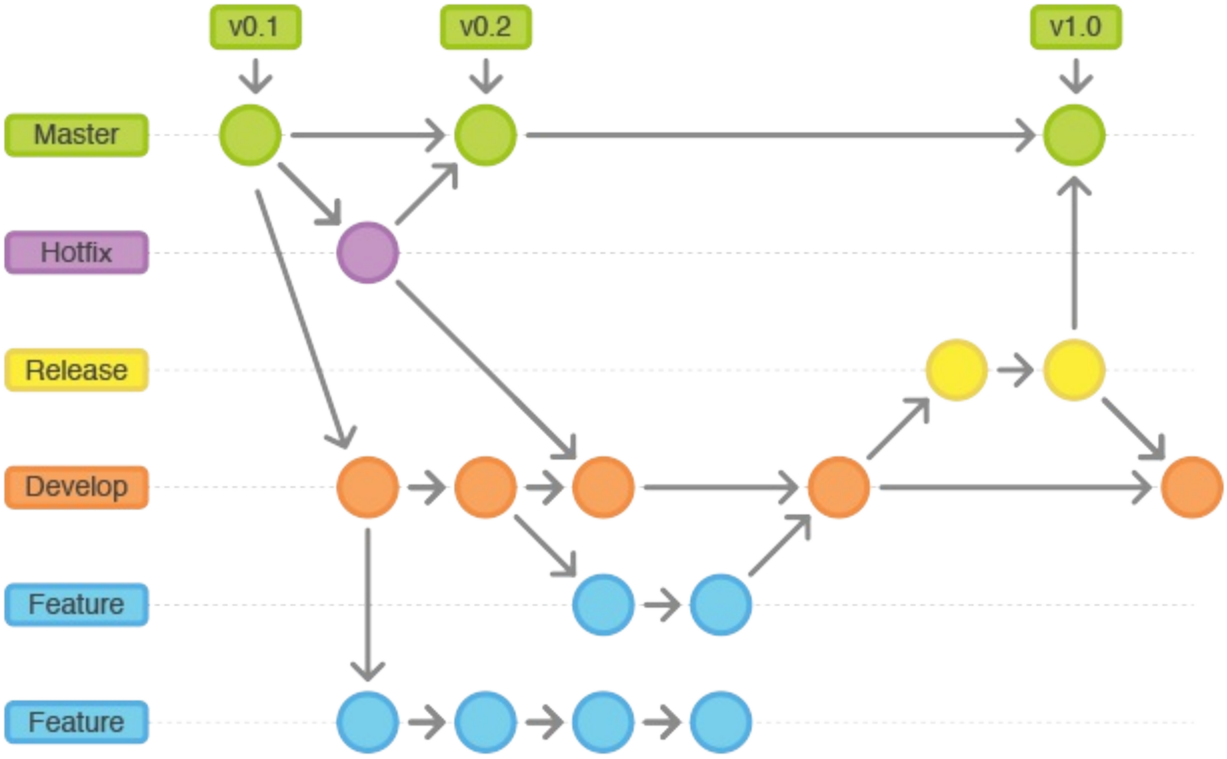
一旦 `develop` 分支上有了做一次发布（或者说快到了既定的发布日）的足够功能，就从 `develop` 分支上 `fork` 一个发布分支。 新建的分支用于开始发布循环，所以从这个时间点开始之后新的功能不能再加到这个分支上—— 这个分支只应该做 `Bug` 修复、文档生成和其它面向发布任务。 一旦对外发布的工作都完成了，发布分支合并到 `master` 分支并分配一个版本号打好 `Tag` 。 另外，这些从新建发布分支以来的做的修改要合并回 `develop` 分支。

使用一个用于发布准备的专门分支，使得一个团队可以在完善当前的发布版本的同时，另一个团队可以继续开发下个版本的功能。 这也打造定义良好的开发阶段（比如，可以很轻松地说，『这周我们要做准备发布版本4.0』，并且在仓库的目录结构中可以实际看到）。

常用的分支约定：

```
用于新建发布分支的分支: develop
用于合并的分支: master
分支命名: release-* 或 release/*
```

维护分支



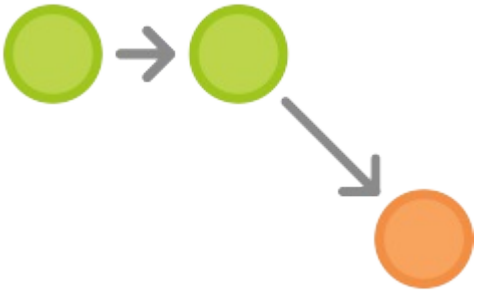
维护分支或说是热修复（ `hotfix` ）分支用于生成快速给产品发布版本（ `production releases` ）打补丁，这是唯一可以直接从 `master` 分支 `fork` 出来的分支。 修复完成，修改应该马上合并回 `master` 分支和 `develop` 分支（当前的发布分支）， `master` 分支应该用新的版本号打好 `Tag` 。

为 `Bug` 修复使用专门分支，让团队可以处理掉问题而不用打断其它工作或是等待下一个发布循环。 你可以把维护分支想成是一个直接在 `master` 分支上处理的临时发布。

示例

下面的示例演示本工作流如何用于管理单个发布循环。假设你已经创建了一个中央仓库。

创建开发分支



第一步为 `master` 分支配套一个 `develop` 分支。简单来做可以本地创建一个空的 `develop` 分支， `push` 到服务器上：

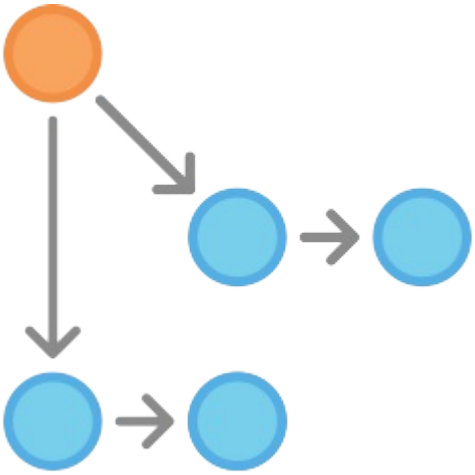
```
git branch develop
git push -u origin develop
```

以后这个分支将会包含了项目的全部历史，而 `master` 分支将只包含了部分历史。其它开发者这时应该克隆中央仓库，建好 `develop` 分支的跟踪分支：

```
git clone ssh://user@host/path/to/repo.git
git checkout -b develop origin/develop

#【译注】当没有本地分支 develop 时，
# 最后一条命令，我使用更简单的 git checkout develop
# 会自动 把 远程分支origin/develop 检出成 本地分支 develop
```

现在每个开发都有了这些历史分支的本地拷贝。

小红和小明开始开发新功能



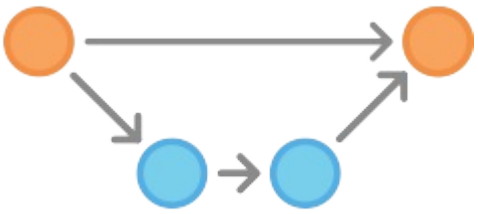
这个示例中，小红和小明开始各自的功能开发。他们需要为各自的功能创建相应的分支。新分支不是基于 `master` 分支，而是应该基于 `develop` 分支：

```
git checkout -b some-feature develop
```

他们用老套路添加提交到各自功能分支上：编辑、暂存、提交：

```
git status
git add <some-file>
git commit
```

小红完成功能开发

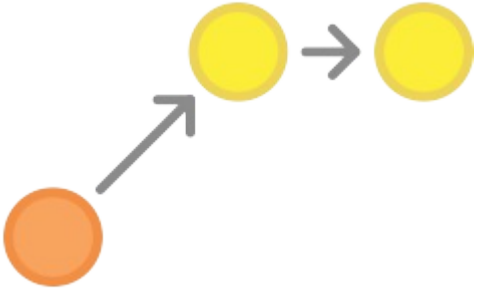


添加了提交后，小红觉得她的功能OK了。如果团队使用 `Pull Requests` ，这时候可以发起一个用于合并到 `develop` 分支。 否则她可以直接合并到她本地的 `develop` 分支后 `push` 到中央仓库：

```
# 拉取远程的develop分支，并且当前分支（本地分支some-feature）合并上远程分支develop
git pull origin develop
git checkout develop
# 本地分支some-feature合并上develop
#【注意】这个分支已经有远程的develop修改了，所以本地develop无需再做远程拉取的操作
git merge some-feature
git push
# 删除本地分支
git branch -d some-feature

#【译注】上面的命令注释为译者添加，以方便理解
# 更多说明参见 Issue #18
```

第一条命令在合并功能前确保 `develop` 分支是最新的。注意，功能决不应该直接合并到 `master` 分支。 冲突解决方法和集中式工作流一样。

小红开始准备发布



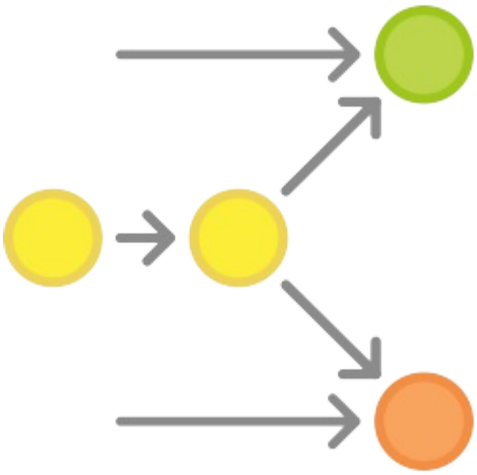
这个时候小明正在实现他的功能，小红开始准备她的第一个项目正式发布。 像功能开发一样，她用一个新的分支来做发布准备。这一步也确定了发布的版本号：

```
git checkout -b release-0.1 develop
```

这个分支是清理发布、执行所有测试、更新文档和其它为下个发布做准备操作的地方，像是一个专门用于改善发布的功能分支。

只要小红创建这个分支并 `push` 到中央仓库，这个发布就是功能冻结的。任何不在 `develop` 分支中的新功能都推到下个发布循环中。

小红完成发布



一旦准备好了对外发布，小红合并修改到 `master` 分支和 `develop` 分支上，删除发布分支。合并回 `develop` 分支很重要，因为在发布分支中已经提交的更新需要在后面的新功能中也要是可用的。 另外，如果小红的团队要求 `Code Review` ，这是一个发起 `Pull Request` 的理想时机。

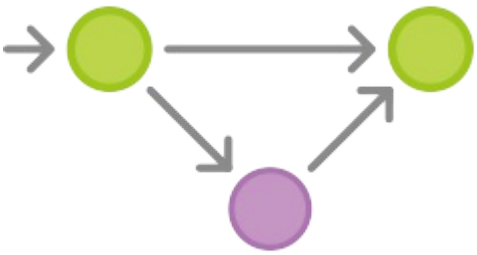
```
git checkout master
git merge release-0.1
git push
git checkout develop
git merge release-0.1
git push
git branch -d release-0.1
```

发布分支是作为功能开发（ `develop` 分支）和对外发布（ `master` 分支）间的缓冲。只要有合并到 `master` 分支，就应该打好 `Tag` 以方便跟踪。

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

`Git` 有提供各种勾子（ `hook` ），即仓库有事件发生时触发执行的脚本。 可以配置一个勾子，在你 `push` 中央仓库的 `master` 分支时，自动构建好对外发布。

最终用户发现 **Bug**



对外发布后，小红回去和小明一起做下个发布的新功能开发，直到有最终用户开了一个 `Ticket` 抱怨当前版本的一个 `Bug` 。 为了处理 `Bug` ，小红（或小明）从 `master` 分支上拉出了一个维护分支，提交修改以解决问题，然后直接合并回 `master` 分支：

```
git checkout -b issue-#001 master
# Fix the bug
git checkout master
git merge issue-#001
git push
```

就像发布分支，维护分支中新加这些重要修改需要包含到 `develop` 分支中，所以小红要执行一个合并操作。然后就可以安全地删除这个分支了：

```
git checkout develop
git merge issue-#001
git push
git branch -d issue-#001
```

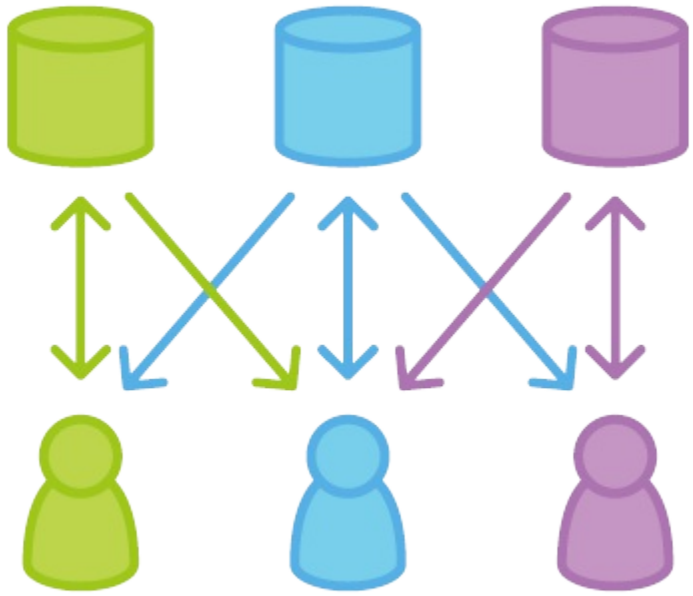
下一站

到了这里，但愿你对集中式工作流、功能分支工作流和 `Gitflow` 工作流已经感觉很舒适了。 你应该也牢固的掌握了本地仓库的潜能， `push` / `pull` 模式和 `Git` 健壮的分支和合并模型。

记住，这里演示的工作流只是可能用法的例子，而不是在实际工作中使用 `Git` 不可违逆的条例。 所以不要畏惧按自己需要对工作流的用法做取舍。不变的目标就是让 `Git` 为你所用。

`Forking` 工作流

`Forking` 工作流和前面讨论的几种工作流有根本的不同。 这种工作流不是使用单个服务端仓库作为『中央』代码基线，而让各个开发者都有一个服务端仓库。 这意味着各个代码贡献者有2个 `Git` 仓库而不是1个：一个本地私有的，另一个服务端公开的。



`Forking` 工作流的一个主要优势是，贡献的代码可以被集成，而不需要所有人都能 `push` 代码到仅有的中央仓库中。 开发者 `push` 到自己的服务端仓库，而只有项目维护者才能 `push` 到正式仓库。 这样项目维护者可以接受任何开发者的提交，但无需给他正式代码库的写权限。

效果就是一个分布式的工作流，能为大型、自发性的团队（包括了不受信的第三方）提供灵活的方式来安全的协作。 也让这个工作流成为开源项目的理想工作流。

工作方式

和其它的 `Git` 工作流一样， `Forking` 工作流要先有一个公开的正式仓库存储在服务器上。 但一个新的开发者想要在项目上工作时，不是直接从正式仓库克隆，而是 `fork` 正式项目在服务器上创建一个拷贝。

这个仓库拷贝作为他个人公开仓库 —— 其它开发者不允许 `push` 到这个仓库，但可以 `pull` 到修改（后面我们很快就会看这点很重要）。 在创建了自己服务端拷贝之后，和之前的工作流一样，开发者执行 `git clone` 命令克隆仓库到本地机器上，作为私有的开发环境。

要提交本地修改时， `push` 提交到自己公开仓库中 —— 而不是正式仓库中。 然后，给正式仓库发起一个 `pull request` ，让项目维护者知道有更新已经准备好可以集成了。 对于贡献的代码， `pull request` 也可以很方便地作为一个讨论的地方。

为了把功能集成到正式代码库，维护者 pull 贡献者的变更到自己的本地仓库中，检查变更以确保不会让项目出错， 合并变更到自己本地的 master 分支， 然后 push master 分支到服务器的正式仓库中。 到此，贡献的提交成为了项目的一部分，其它的开发者应该执行 pull 操作与正式仓库同步自己本地仓库。

正式仓库

在 Forking 工作流中，『官方』仓库的叫法只是一个约定，理解这点很重要。 从技术上来看，各个开发者仓库和正式仓库在 Git 看来没有任何区别。 事实上，让正式仓库之所以正式的唯一原因是它是项目维护者的公开仓库。

Forking 工作流的分支使用方式

所有的个人公开仓库实际上只是为了方便和其它的开发者共享分支。 各个开发者应该用分支隔离各个功能，就像在功能分支工作流和 Gitflow 工作流一样。 唯一的区别是这些分支被共享了。在 Forking 工作流中这些分支会被 pull 到另一个开发者的本地仓库中，而在功能分支工作流和 Gitflow 工作流中是直接被 push 到正式仓库中。

示例

项目维护者初始化正式仓库

和任何使用 Git 项目一样，第一步是创建在服务器上一个正式仓库，让所有团队成员都可以访问到。 通常这个仓库也会作为项目维护者的公开仓库。

公开仓库应该是裸仓库，不管是不是正式代码库。 所以项目维护者会运行像下面的命令来搭建正式仓库：

```
ssh user@host
git init --bare /path/to/repo.git
```

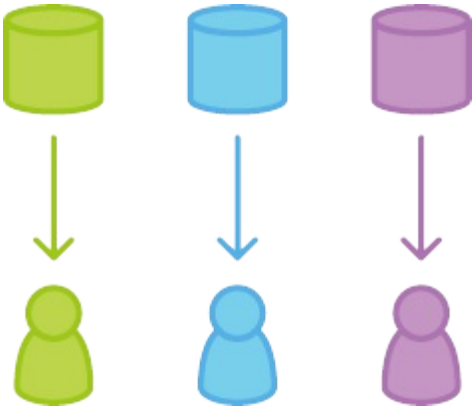
Bitbucket 和 Stash 提供了一个方便的 GUI 客户端以完成上面命令行做的事。 这个搭建中央仓库的过程和前面提到的工作流完全一样。 如果有现存的代码库，维护者也要 push 到这个仓库中。

开发者 fork 正式仓库

其它所有的开发需要 fork 正式仓库。 可以用 git clone 命令用 SSH 协议连通到服务器， 拷贝仓库到服务器另一个位置 —— 是的， fork 操作基本上就只是一个服务端的克隆。 Bitbucket 和 Stash 上可以点一下按钮就让开发者完成仓库的 fork 操作。

这一步完成后，每个开发都在服务端有一个自己的仓库。和正式仓库一样，这些仓库应该是裸仓库。

开发者克隆自己 `fork` 出来的仓库



下一步，各个开发者要克隆自己的公开仓库，用熟悉的 `git clone` 命令。

在这个示例中，假定用 `Bitbucket` 托管了仓库。记住，如果这样的话各个开发者需要有各自的 `Bitbucket` 账号， 使用下面命令克隆服务端自己的仓库：

```
git clone https://user@bitbucket.org/user/repo.git
```

相比前面介绍的工作流只用了一个 `origin` 远程别名指向中央仓库， `Forking` 工作流需要2个远程别名 —— 一个指向正式仓库，另一个指向开发者自己的服务端仓库。别名的名字可以任意命名，常见的约定是使用 `origin` 作为远程克隆的仓库的别名 （这个别名会在运行 `git clone` 自动创建）， `upstream` （上游）作为正式仓库的别名。

```
git remote add upstream https://bitbucket.org/maintainer/repo
```

需要自己用上面的命令创建 `upstream` 别名。这样可以简单地保持本地仓库和正式仓库的同步更新。 注意，如果上游仓库需要认证（比如不是开源的），你需要提供用户：

```
git remote add upstream https://user@bitbucket.org/maintainer/repo.git
```

这时在克隆和 `pull` 正式仓库时，需要提供用户的密码。

开发者开发自己的功能

在刚克隆的本地仓库中，开发者可以像其它工作流一样的编辑代码、提交修改和新建分支：

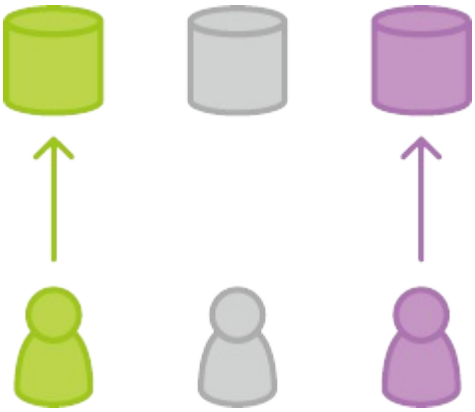
```
git checkout -b some-feature
# Edit some code
git commit -a -m "Add first draft of some feature"
```

所有的修改都是私有的直到 `push` 到自己公开仓库中。如果正式项目已经往前走了，可以用 `git pull` 命令获得新的提交：

```
git pull upstream master
```

由于开发者应该都在专门的功能分支上工作， `pull` 操作结果会都是快进合并。

开发者发布自己的功能



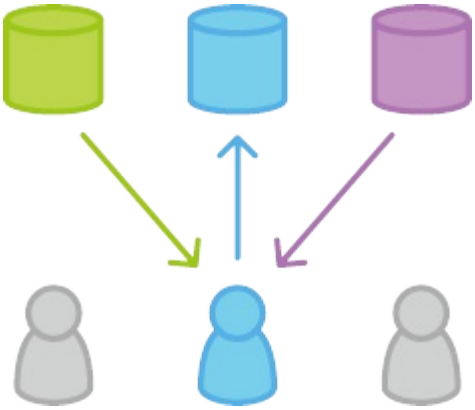
一旦开发者准备好了分享新功能，需要做二件事。 首先，通过 `push` 他的贡献代码到自己的公开仓库中，让其它的开发者都可以访问到。 他的 `origin` 远程别名应该已经有了，所以要做的就是：

```
git push origin feature-branch
```

这里和之前的工作流的差异是， `origin` 远程别名指向开发者自己的服务端仓库，而不是正式仓库。

第二件事，开发者要通知项目维护者，想要合并他的新功能到正式库中。 `Bitbucket` 和 `Stash` 提供了 `Pull Request` 按钮，弹出表单让你指定哪个分支要合并到正式仓库。 一般你会想集成你的功能分支到上游远程仓库的 `master` 分支中。

项目维护者集成开发者的功能



当项目维护者收到 `pull request`，他要做的是决定是否集成它到正式代码库中。有二种方式来做：

1. 直接在 `pull request` 中查看代码
2. `pull` 代码到他自己的本地仓库，再手动合并

第一种做法更简单，维护者可以在 `GUI` 中查看变更的差异，做评注和执行合并。 但如果出现了合并冲突，需要第二种做法来解决。这种情况下，维护者需要从开发者的服务端仓库中 `fetch` 功能分支， 合并到他本地的 `master` 分支，解决冲突：

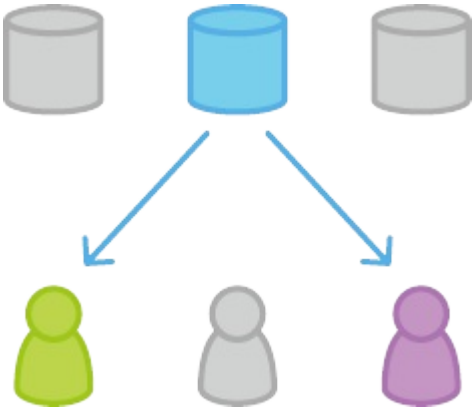
```
git fetch https://bitbucket.org/user/repo feature-branch
# 查看变更
git checkout master
git merge FETCH_HEAD
```

变更集成到本地的 `master` 分支后，维护者要 `push` 变更到服务器上的正式仓库，这样其它的开发者都能访问到：

```
git push origin master
```

注意，维护者的 `origin` 是指向他自己公开仓库的，即是项目的正式代码库。到此，开发者的贡献完全集成到了项目中。

开发者和正式仓库做同步



由于正式代码库往前走了，其它的开发需要和正式仓库做同步：

```
git pull upstream master
```

下一站

如果你之前是使用 `SVN`， `Forking` 工作流可能看起来像是一个激进的范式切换（paradigm shift）。 但不要害怕，这个工作流实际上就是在功能分支工作流之上引入另一个抽象层。 不是直接通过单个中央仓库来分享分支，而是把贡献代码发布到开发者自己的服务端仓库中。

示例中解释了，一个贡献如何从一个开发者流到正式的 `master` 分支中，但同样的方法可以把贡献集成到任一个仓库中。 比如，如果团队的几个人协作实现一个功能，可以在开发之间用相同的方法分享变更，完全不涉及正式仓库。

这使得 `Forking` 工作流对于松散组织的团队来说是个非常强大的工具。任一开发者可以方便地和另一开发者分享变更，任何分支都能有效地合并到正式代码库中。

注意事项

1. 和源码无关的东西，尽量不要进仓库
2. 尽量采用相对小、相对独立的提交

   Git 是作什么用的？Git 不是代码上传工具，也不是网站更新工具，而是软件开发过程的记录工具，为了更加准确的定位每个问题、每个功能修改，就需要在每完成一部分可以称得上是“一项”的工作时，就 commit 一次。哪怕只是修改了一两行，只要产生了必要的功能改变，就有价值记录。

3. 注释格式

   格式属于个人习惯和团队规范范围，有必要采用相对统一的风格。Git 本身不允许空注释，同时建议注释的第一行写简要说明，下面留一行空行，再写详细说明。