**Gitlab使用说明**

* **1、配置本地git环境**

**1.1安装**

分别介绍在windows下，或Linux下，或使用开发环境自带的git插件

**1.1.1 git for windows**

如果不依托开发环境（vs2013、eclips）自带的git插件，就需要手动安装git环境

官方链接：<https://git-scm.com/download/win>

然而并下不动，所以使用国内CSDN的下载：

[http://download.csdn.net/detail/u010668495/9231643#comment](http://download.csdn.net/detail/u010668495/9231643" \l "comment)

下载后按步骤安装即可

装好后有GUI和Git Bash可以用（GMD和CMD没有区别）

（添加了安装包在仓库）

**1.1.2 git in linux**

在linux上的安装方法参见：

<https://git-scm.com/download/linux>

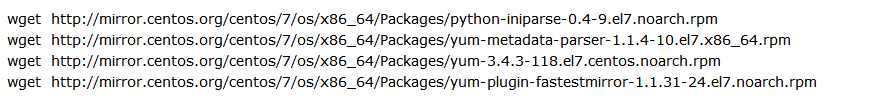
这里说一下RedHat这种yum源不能用的系统应该怎么办

首先我们应该把RHEL的yum源替换为Centos的，比如网易源，这样就可以利用yum源下载安装了。

参考网址：

<http://www.cnblogs.com/wutengbiao/p/4188986.html>

此处应该注意：



这里的这几个文件应该和RHEL版本保持一致，比如RH6，就应该去6那里下。

打开：<http://mirror.centos.org/centos/>

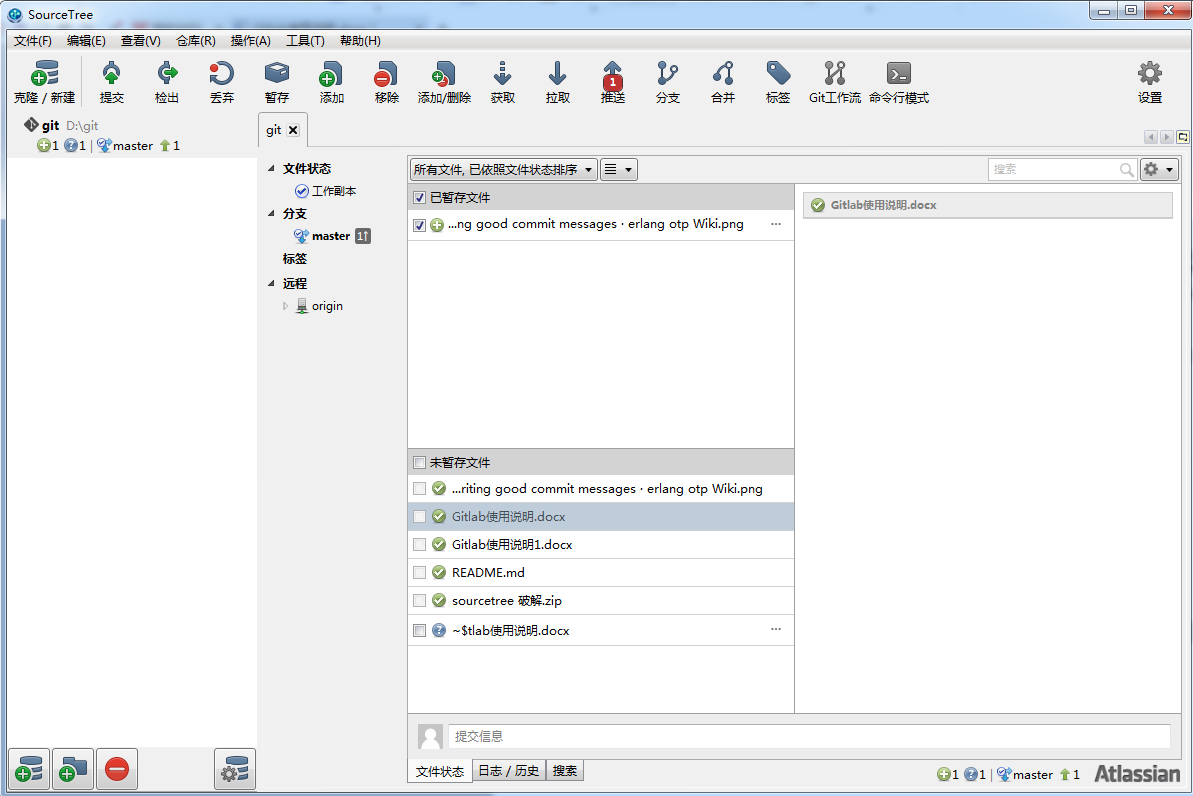
后选择6/os/你的系统类型（32位 or 64位），然后搜索以上类似的文件名来wget。

**1.1.3 visual studio自带的git环境**



在团队资源管理器页面自带git环境，不需要装任何第三方插件

#### 1.1.4 （推荐）使用sourcetree



图形化界面和选项，基本告别命令行，各个开发环境通用，不用再装别的了。

<http://download.csdn.net/download/micro_hz/9419272>

安装包已经在下载的库中。

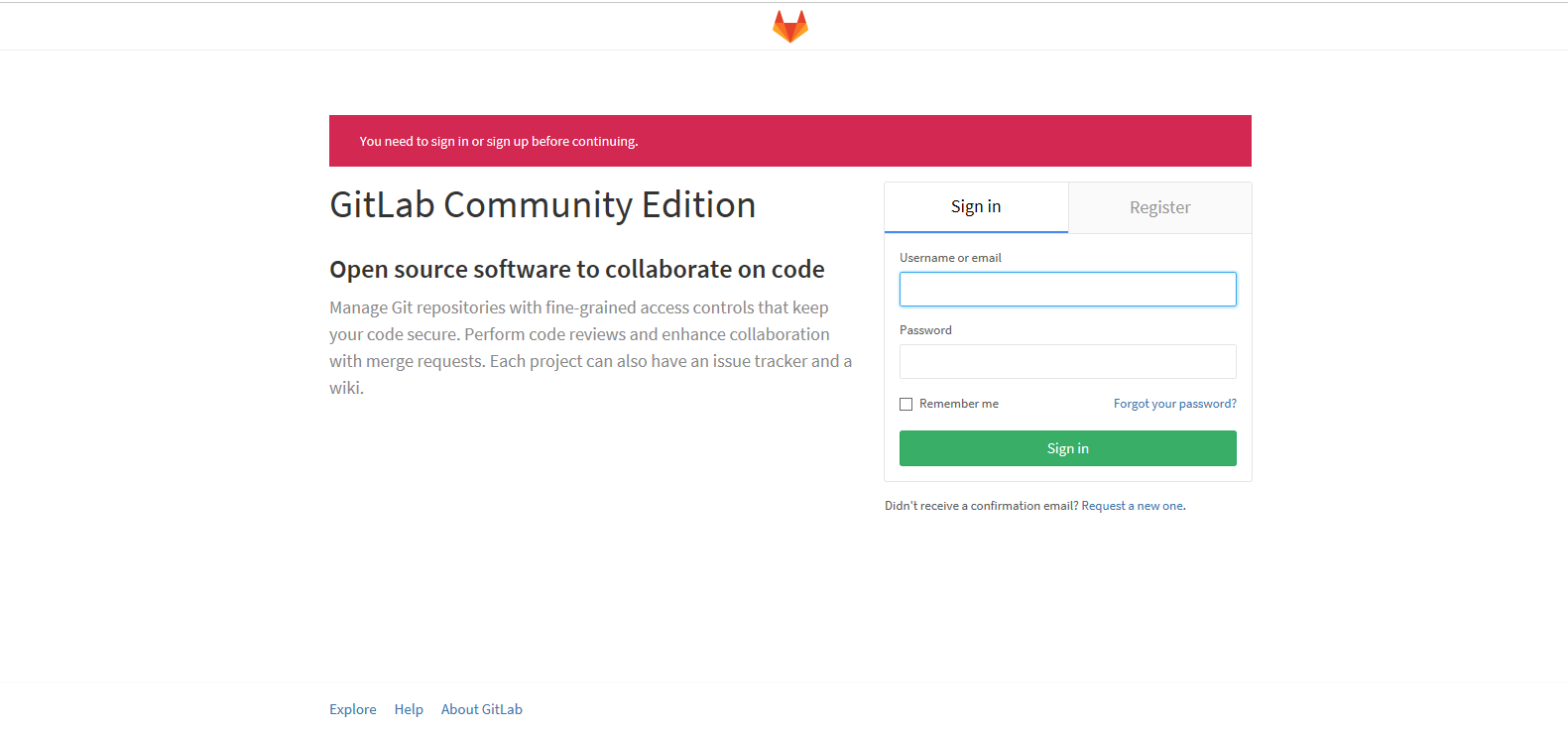
在安装后，有一个界面是“正在下载版本控制系统”。那个界面点X，然后选择压缩包解压后的文件夹内执行程序。

Test

## GitLab使用介绍

### 2.1 注册

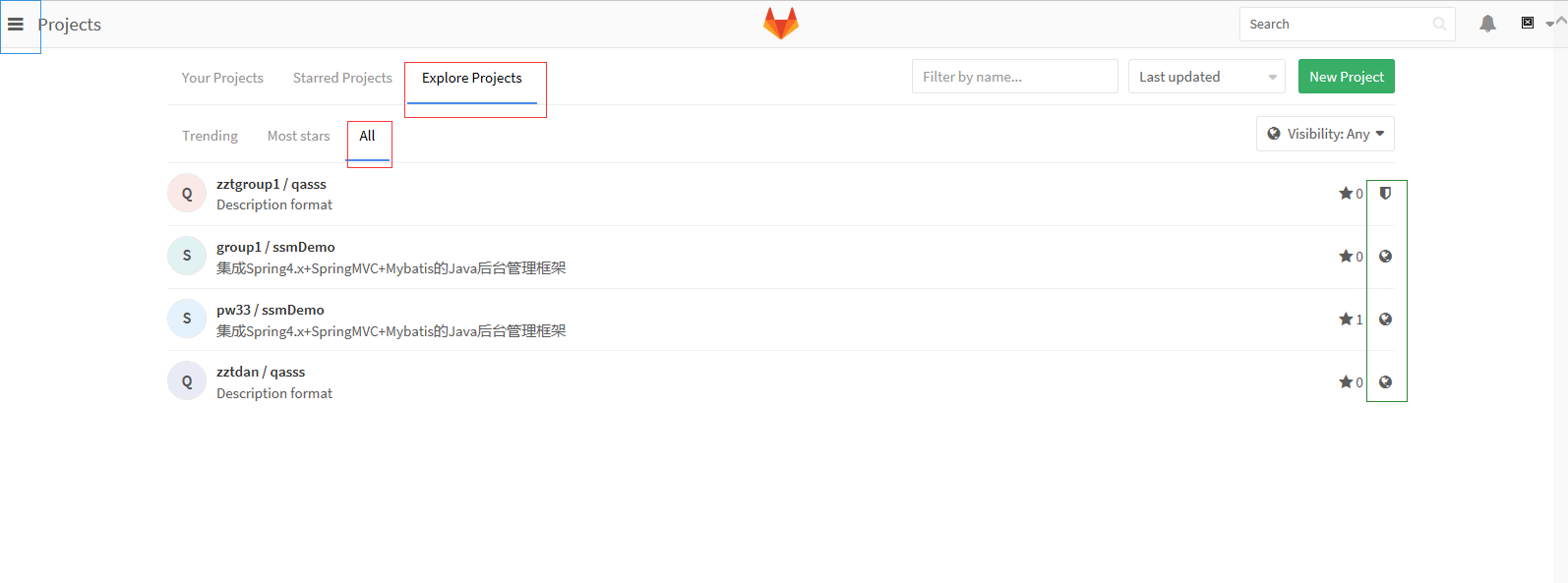
先打开公司提供的gitlab网址（现在是172.16.4.192），进入后界面如下：



点击register注册后，登录进入gitlab主界面。

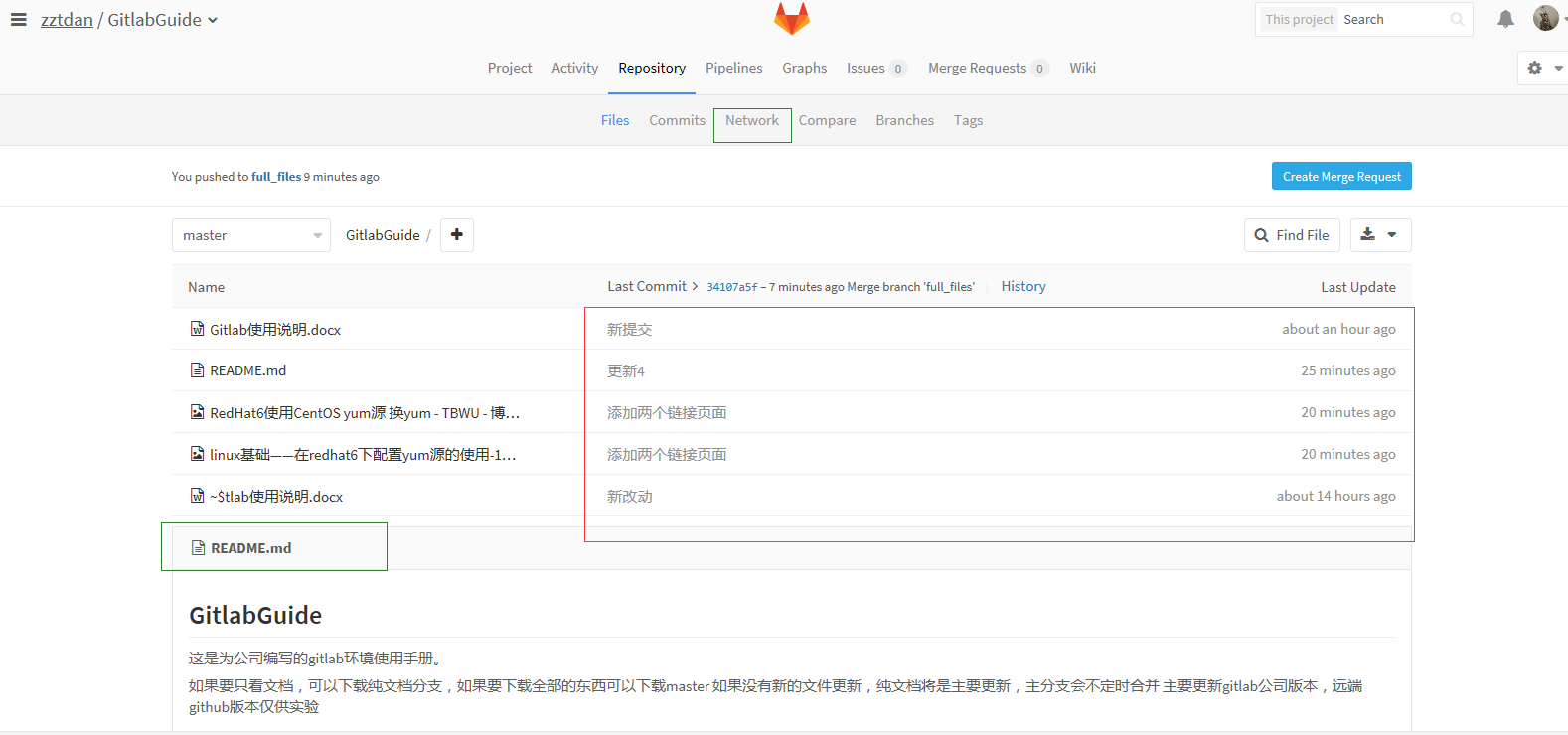
### 2.2 主要工程

在explore project/all界面可以看到现有的所有工程，旁边有搜索功能



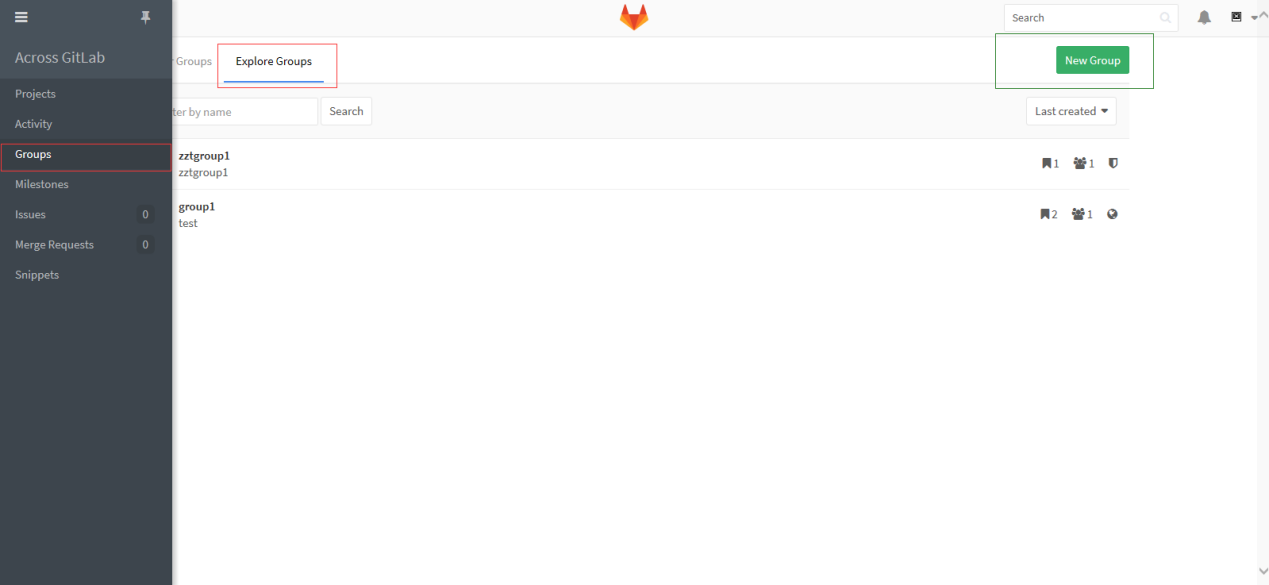
注意上图中蓝色框功能可以打开侧面菜单栏，绿框内显示的是项目库的类型，有私有（只有自己能用）、保护（只有注册用户能用）和公有（大家都能用）。

进入工程后可以看到工程的主分支master的文件情况：



点击network可以看到版本工作流，在该版本库中的readme.md文件将直接显示在页面中。红色框内的是master分支下每个文件的最后更新情况。

### 2.4 工作组



点开左上角目录，可以选择工作组界面。

在工作组界面中可以查看所有工作组，用new group可以新建属于自己的工作组。比如自己要开一个项目，就建立工作组。只有组长可以拉人进入工作组。

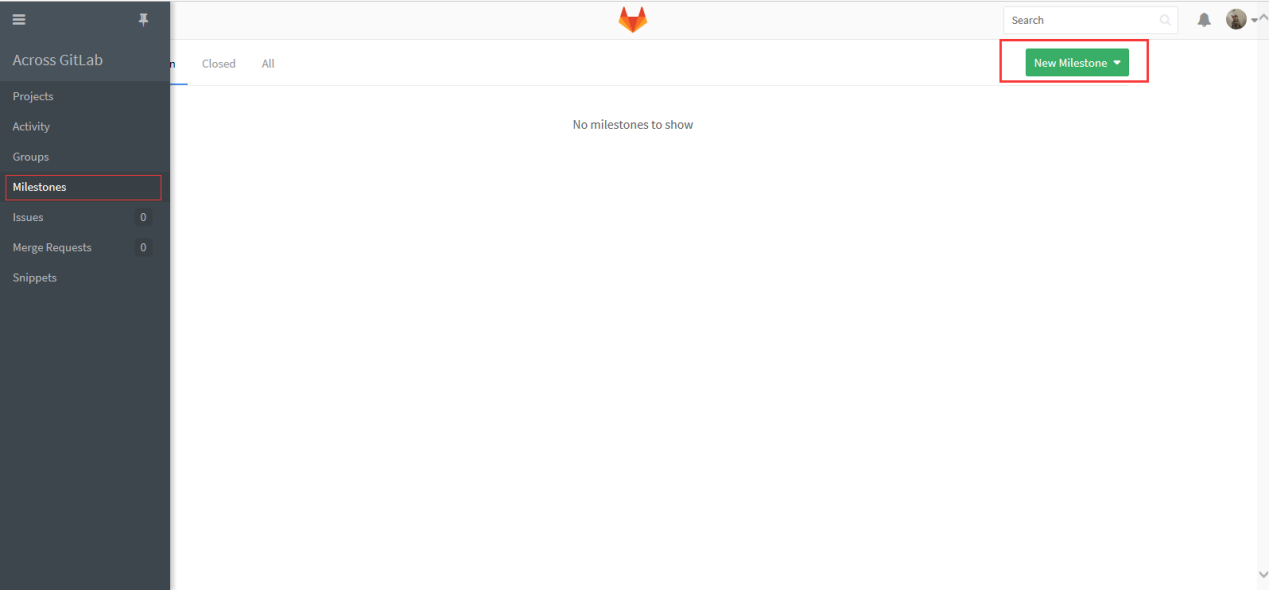
### 2.5 里程碑（milestone）/问题反馈（issue）

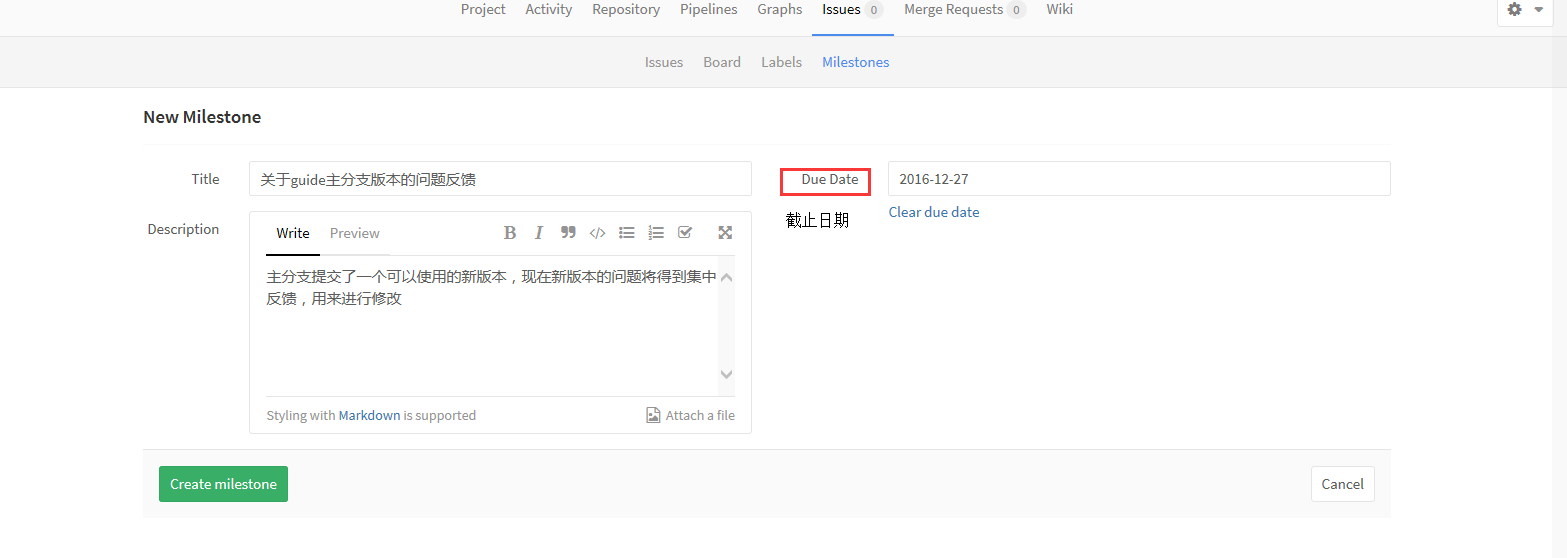
该功能用于给版本或分支提问题，修复bug什么的，是一个带有时限的紧急工作计划。如果在时限前所有issue完美解决，那么里程碑就完成。

里程碑是依附于项目的，下面展示一个里程碑解决实例。

#### 2.5.1 里程碑解决实例

首先，现在我的gitlab项目提交了新的版本，反应了很多问题，那么我建立了一个里程碑：

建立后画面如下：



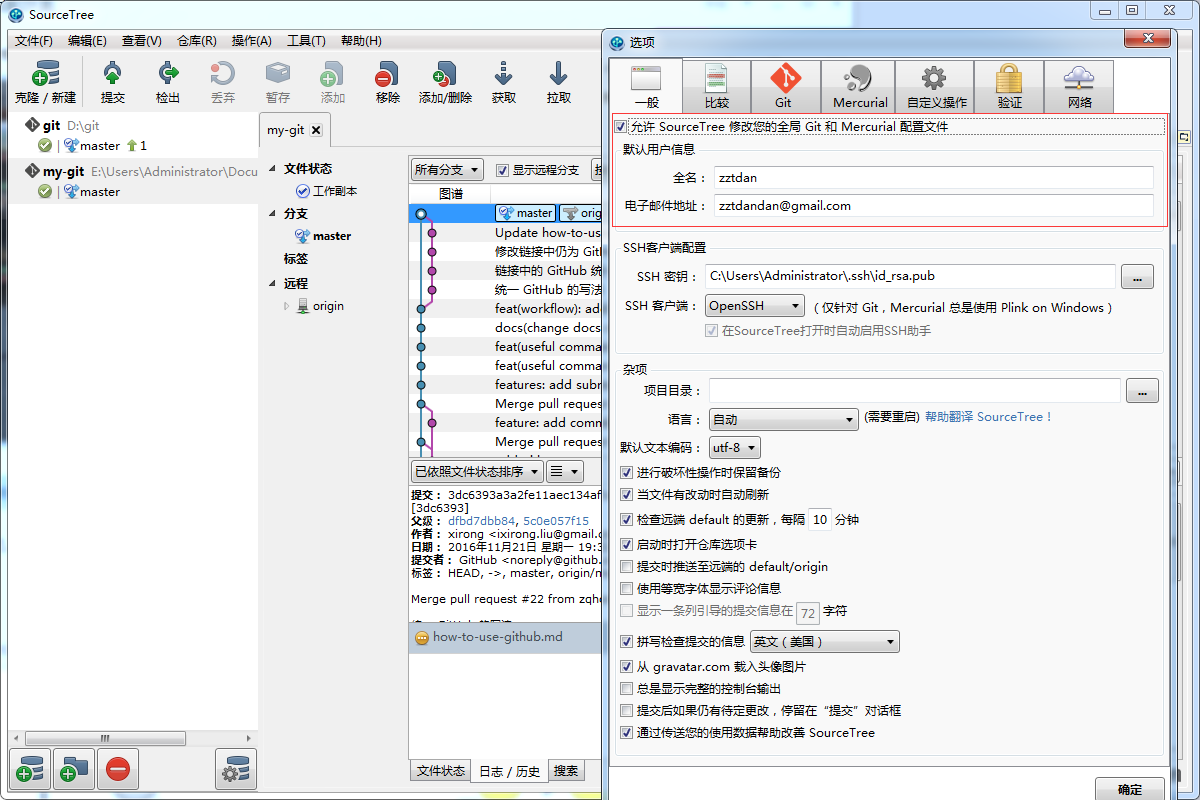
## 3、git使用补遗介绍

### 3.1首先是一些简要的概念介绍

建议直接看：

<http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>

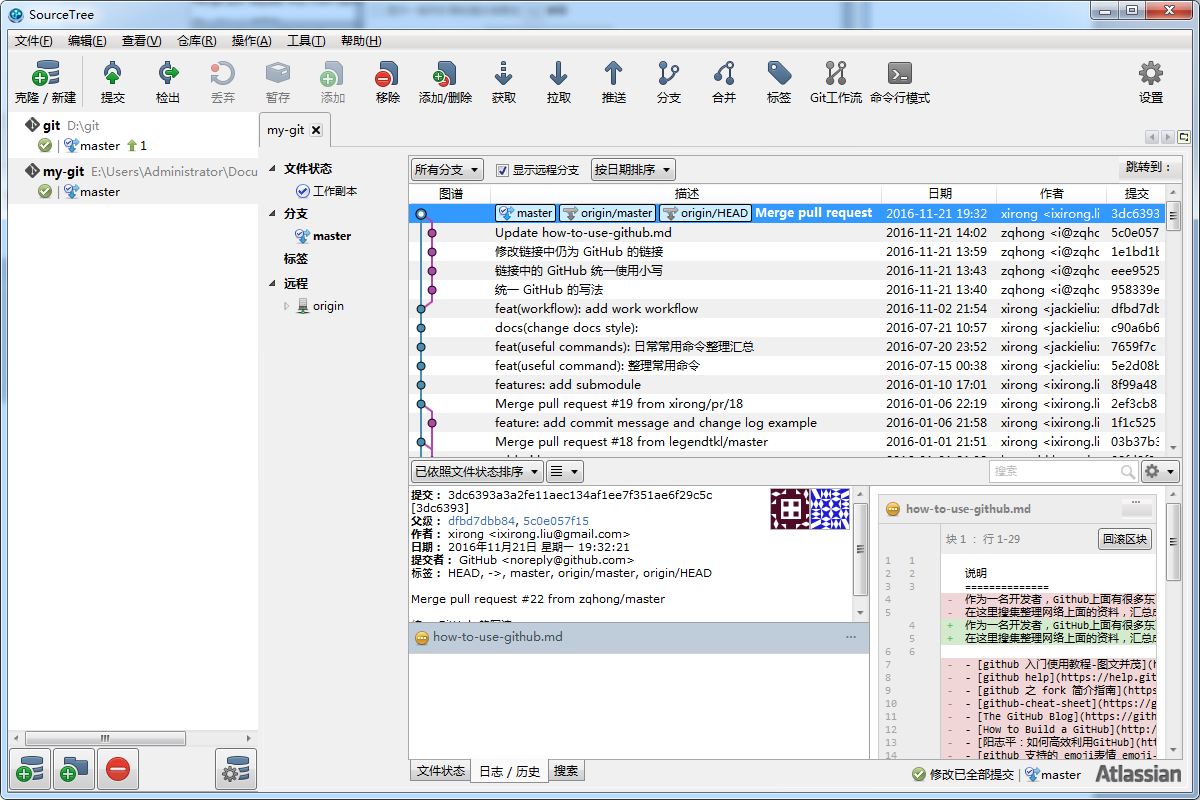
这里主要介绍介绍在source tree(简称ST)中使用



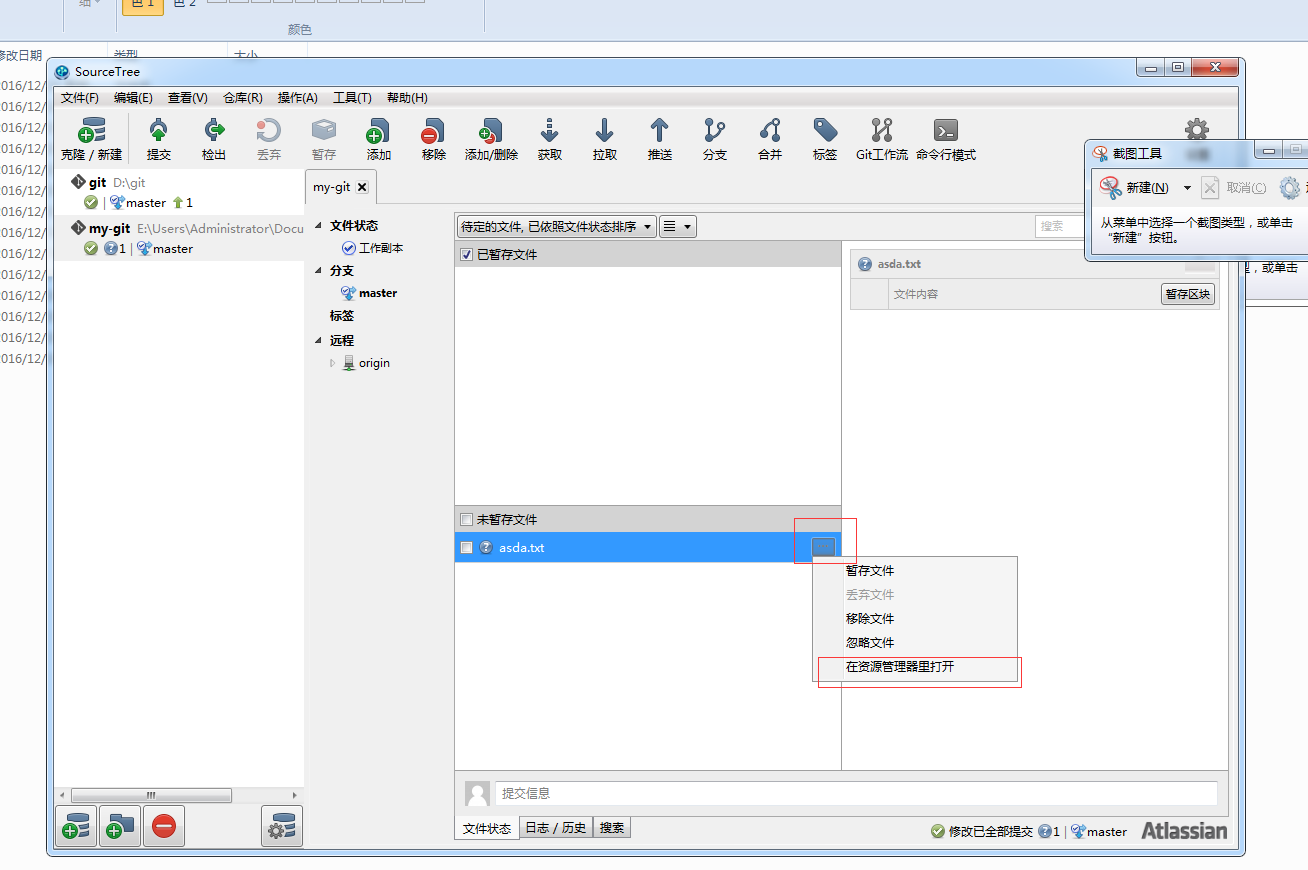
首先在安装好ST后，进入工具——选项设置全局git用户名和邮箱。

接着可以使用按钮新建、克隆库。

在克隆了一个库后的图像如下：



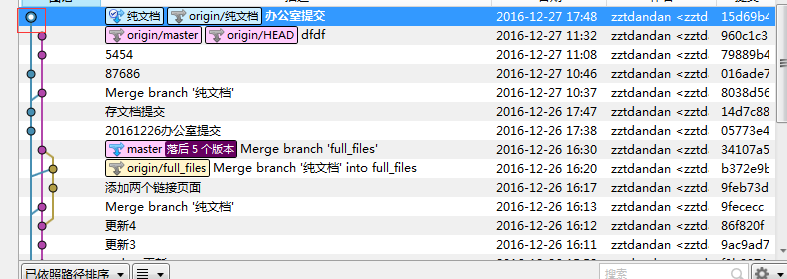
选择文件状态标签（绿框）可以看到在克隆后新做的改动：



可以点击改动文件在管理器中打开，如果该文件可执行也可以直接打开。

如果想将未暂存文件加入暂存版本，点击添加按钮，或者暂存文件前面的框，如果暂存所有文件，点上面未暂存文件左边的框。

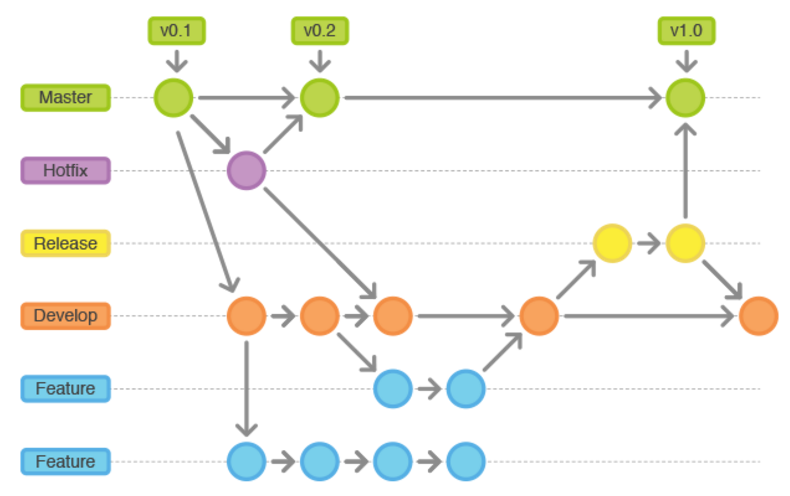
这里说一下上面的暂存按钮，这个按钮的功能是将现在改动过，没有生成版本的工作副本存入git自带的stash，同时将工作副本还原到改动初始点，就是在视图中点亮的版本这个状态



但是要恢复这个暂存的工作副本，需要打命令行，这就不如直接将现有工作副本保存为版本，日后需要时检出来得方便，所以不推荐使用这个暂存功能。

## 4、使用gitlab作为远程仓库进行协作开发

### 4.1 工作流



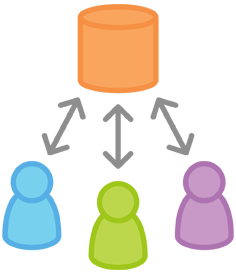
Gitflow工作流定义了一个围绕项目发布的严格分支模型。虽然比功能分支工作流复杂几分，但提供了用于一个健壮的用于管理大型项目的框架。

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。

下面对比一下其他的工作流方式

#### 4.1.1 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。



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 ——

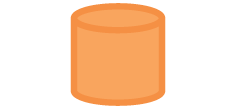
即自由地提交到自己的本地仓库，先完全忽略上游的开发，直到方便的时候再把修改反馈上去。

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。

#### 4.1.1.3 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gitlab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。

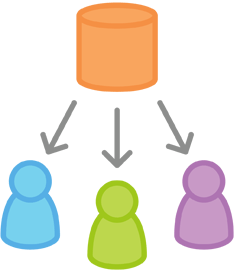
中央仓库应该是个裸仓库（bare repository），即没有工作目录（working directory）的仓库。可以用下面的命令创建：

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

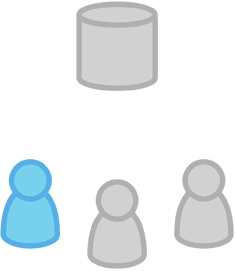


下一步，各个开发者创建整个项目的本地拷贝。通过[git clone](https://www.atlassian.com/git/tutorial/git-basics" \l "!clone" \t "_blank)命令完成：

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**的用来准备一个提交，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。

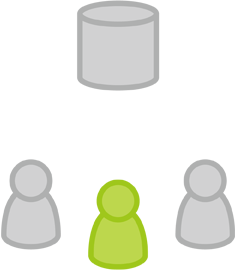
git status # 查看本地仓库的修改状态

git add # 暂存文件

git commit # 提交文件

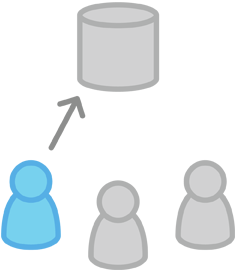
请记住，因为这些命令生成的是本地提交，小明可以按自己需求反复操作多次，而不用担心中央仓库上有了什么操作。  
对需要多个更简单更原子分块的大功能，这个做法是很有用的。

#### 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。

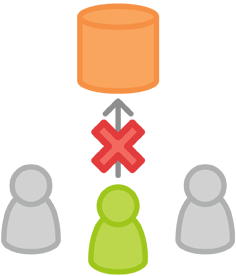
#### 小明发布功能



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](https://www.atlassian.com/git/tutorial/remote-repositories" \l "!push" \t "_blank)：

git push origin master

#### 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 origin master

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

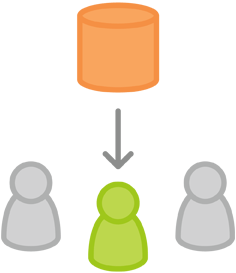
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

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。

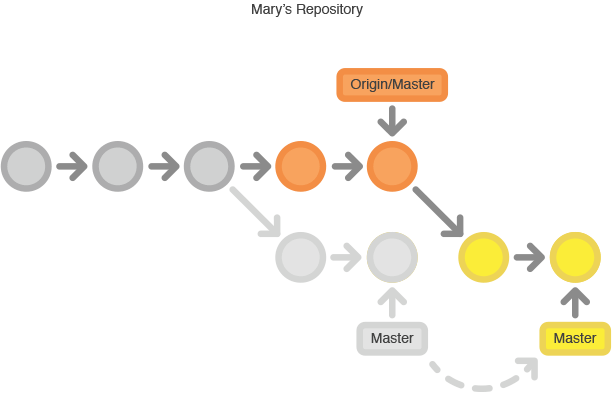
#### 小红在小明的提交之上rebase



小红用[git pull](https://www.atlassian.com/git/tutorial/remote-repositories" \l "!pull" \t "_blank)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：

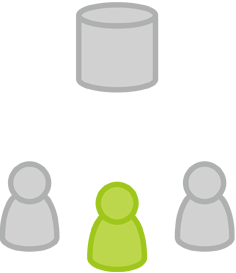
git pull --rebase origin master

--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。

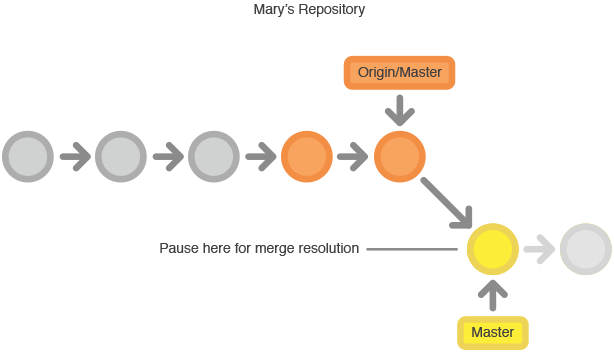
#### 小红解决合并冲突



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](https://www.atlassian.com/git/tutorial/git-basics" \l "!status" \t "_blank)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](https://www.atlassian.com/git/tutorial/rewriting-git-history" \l "!rebase" \t "_blank)完成剩下的事：

git add <some-file>

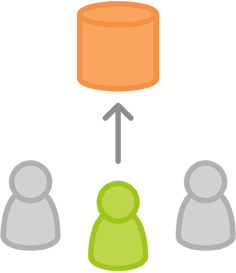
git rebase --continue

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](https://www.atlassian.com/git/tutorial/remote-repositories" \l "!pull" \t "_blank)命令前的样子：

git rebase --abort

#### 小红成功发布功能



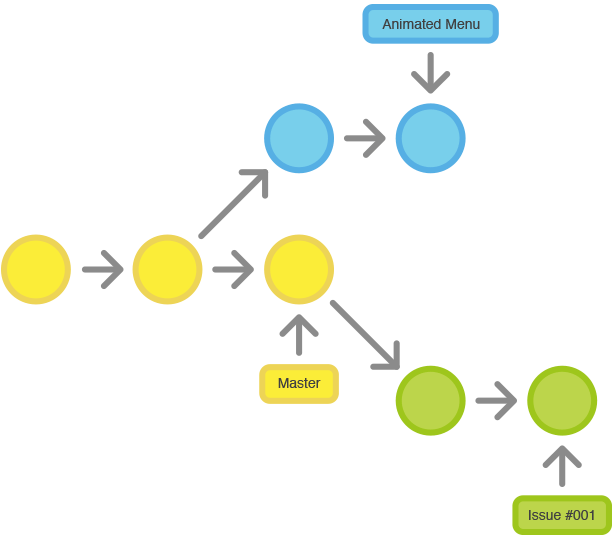
小红完成和中央仓库的同步后，就能成功发布她的修改了：

git push origin master

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。

#### 4.1.2 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。



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工作流成功可能，

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！

#### 4.1.2.1 功能式工作流示例

<http://www.jianshu.com/p/91acec85c3a4>

test3