Git与GitHub总结

什么是git？

Git是目前世界上最先进的分布式版本控制系统，能记录每次文件的改动。

CVS及SVN都是集中式的版本控制系统，而Git是分布式版本控制系统，集中式和分布式版本控制系统有什么区别呢？

先说集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。

那分布式版本控制系统与集中式版本控制系统有何不同呢？首先，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。

### 什么是 Github?

github是一个基于git的代码托管平台，付费用户可以建私人仓库，我们一般的免费用户只能使用公共仓库，也就是代码要公开。

如何使用？

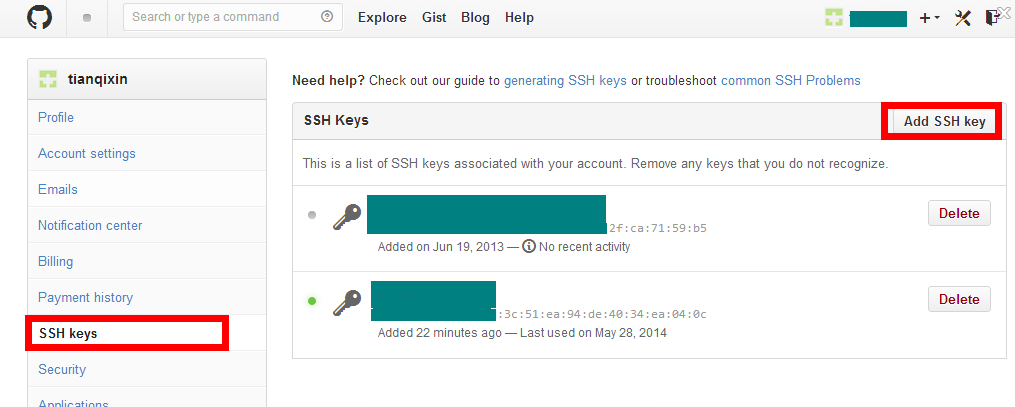
### **配置Git**

首先在本地创建ssh key；

$ ssh-keygen -t rsa -C "your\_email@youremail.com"

后面的your\_email@youremail.com改为你在github上注册的邮箱，之后会要求确认路径和输入密码，我们这使用默认的一路回车就行。成功的话会在~/下生成.ssh文件夹，进去，打开id\_rsa.pub，复制里面的key。

回到github上，进入 Account Settings（账户配置），左边选择SSH Keys，Add SSH Key,title随便填，粘贴在你电脑上生成的key。



为了验证是否成功，在git bash下输入：

$ ssh -T git@github.com

如果是第一次的会提示是否continue，输入yes就会看到：You've successfully authenticated, but GitHub does not provide shell access 。这就表示已成功连上github。

接下来我们要做的就是把本地仓库传到github上去，在此之前还需要设置username和email，因为github每次commit都会记录他们。

$ git config --global user.name "your name"

$ git config --global user.email "your\_email@youremail.com"

进入要上传的仓库，右键git bash，添加远程地址：

$ git remote add origin git@github.com:yourName/yourRepo.git

后面的yourName和yourRepo表示你再github的用户名和刚才新建的仓库，加完之后进入.git，打开config，这里会多出一个remote "origin"内容，这就是刚才添加的远程地址，也可以直接修改config来配置远程地址。

<h3创建新仓库< h3="" style="color: rgb(51, 51, 51); font-family: "Helvetica Neue", Helvetica, "PingFang SC", "Hiragino Sans GB", "Microsoft YaHei", "Noto Sans CJK SC", "WenQuanYi Micro Hei", Arial, sans-serif; font-size: 13px; font-style: normal; font-variant-ligatures: normal; font-variant-caps: normal; font-weight: 400; letter-spacing: normal; orphans: 2; text-align: start; text-indent: 0px; text-transform: none; white-space: normal; widows: 2; word-spacing: 0px; -webkit-text-stroke-width: 0px; background-color: rgb(255, 255, 255); text-decoration-style: initial; text-decoration-color: initial;">

创建新文件夹，打开，然后执行 git init 以创建新的 git 仓库。

### **检出仓库**

执行如下命令以创建一个本地仓库的克隆版本：

git clone /path/to/repository

如果是远端服务器上的仓库，你的命令会是这个样子：

git clone username@host:/path/to/repository

### **工作流**

你的本地仓库由 git 维护的三棵"树"组成。第一个是你的 工作目录，它持有实际文件；第二个是 暂存区（Index），它像个缓存区域，临时保存你的改动；最后是 HEAD，它指向你最后一次提交的结果。

你可以提出更改（把它们添加到暂存区），使用如下命令：  
git add <filename>  
git add \*  
这是 git 基本工作流程的第一步；使用如下命令以实际提交改动：  
git commit -m "代码提交信息"  
现在，你的改动已经提交到了 **HEAD**，但是还没到你的远端仓库。



### **推送改动**

你的改动现在已经在本地仓库的 **HEAD** 中了。执行如下命令以将这些改动提交到远端仓库：  
git push origin master  
可以把 *master* 换成你想要推送的任何分支。   
  
如果你还没有克隆现有仓库，并欲将你的仓库连接到某个远程服务器，你可以使用如下命令添加：  
git remote add origin <server>  
如此你就能够将你的改动推送到所添加的服务器上去了。

### **分支**

分支是用来将特性开发绝缘开来的。在你创建仓库的时候，*master* 是"默认的"分支。在其他分支上进行开发，完成后再将它们合并到主分支上。



创建一个叫做"feature\_x"的分支，并切换过去：  
git checkout -b feature\_x  
切换回主分支：  
git checkout master  
再把新建的分支删掉：  
git branch -d feature\_x  
除非你将分支推送到远端仓库，不然该分支就是 *不为他人所见的*：  
git push origin <branch>

### **更新与合并**

要更新你的本地仓库至最新改动，执行：  
git pull  
以在你的工作目录中 *获取（fetch）* 并 *合并（merge）* 远端的改动。  
要合并其他分支到你的当前分支（例如 master），执行：  
git merge <branch>  
在这两种情况下，git 都会尝试去自动合并改动。遗憾的是，这可能并非每次都成功，并可能出现*冲突（conflicts）*。 这时候就需要你修改这些文件来手动合并这些*冲突（conflicts）*。改完之后，你需要执行如下命令以将它们标记为合并成功：  
git add <filename>  
在合并改动之前，你可以使用如下命令预览差异：  
git diff <source\_branch> <target\_branch>

### **标签**

为软件发布创建标签是推荐的。这个概念早已存在，在 SVN 中也有。你可以执行如下命令创建一个叫做 *1.0.0* 的标签：  
git tag 1.0.0 1b2e1d63ff  
*1b2e1d63ff* 是你想要标记的提交 ID 的前 10 位字符。可以使用下列命令获取提交 ID：  
git log  
你也可以使用少一点的提交 ID 前几位，只要它的指向具有唯一性。

### **替换本地改动**

假如你操作失误（当然，这最好永远不要发生），你可以使用如下命令替换掉本地改动：  
git checkout -- <filename>  
此命令会使用 HEAD 中的最新内容替换掉你的工作目录中的文件。已添加到暂存区的改动以及新文件都不会受到影响。

假如你想丢弃你在本地的所有改动与提交，可以到服务器上获取最新的版本历史，并将你本地主分支指向它：  
git fetch origin  
git reset --hard origin/master

### **实用小贴士**

内建的图形化 git：  
gitk  
彩色的 git 输出：  
git config color.ui true  
显示历史记录时，每个提交的信息只显示一行：  
git config format.pretty oneline  
交互式添加文件到暂存区：  
git add -i

问题？

# 如何解决failed to push some refs to git

Administrator@PC-20150110FGWU /K/cocos2d/yc (master)

$ git push -u origin master

To git@github.com:yangchao0718/cocos2d.git

 ! [rejected]        master -> master (non-fast-forward)

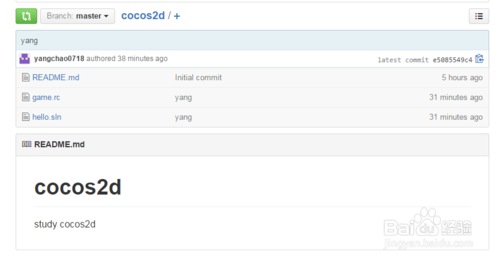
error: failed to push some refs to 'git@github.com:yangchao0718/cocos2d.git

hint: Updates were rejected because the tip of your current branch is behin

hint: its remote counterpart. Integrate the remote changes (e.g.

hint: 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

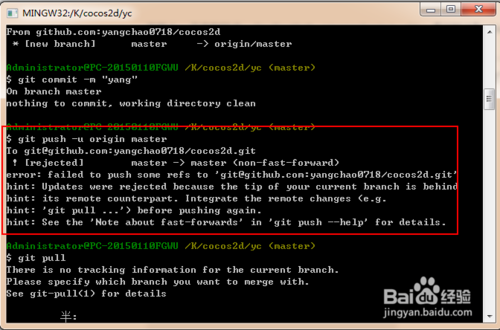
[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=1)

## 工具/原料

git

## 方法/步骤

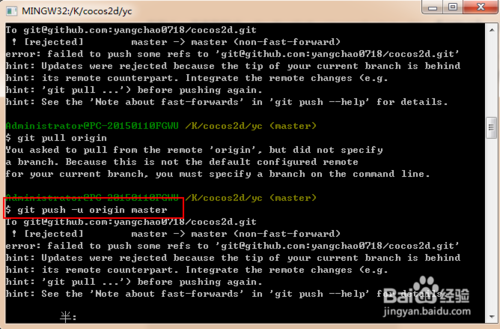
在使用git 对源代码进行push到gitHub时可能会出错，信息如下

[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=2)

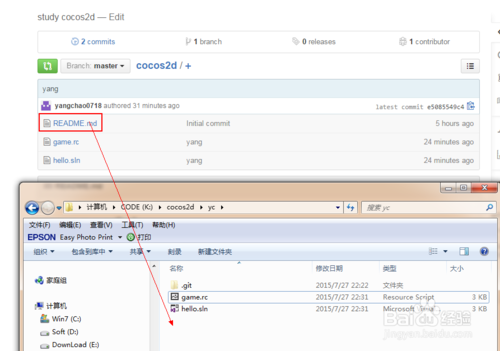
此时很多人会尝试下面的命令把当前分支代码上传到master分支上。

$ git push -u origin master

但依然没能解决问题

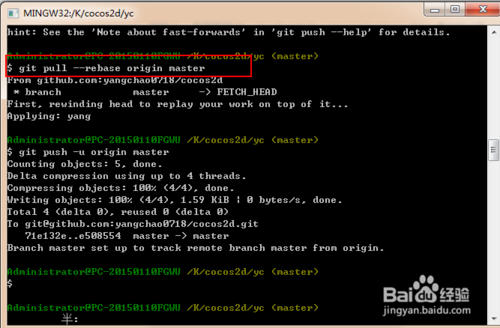
[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=3)

出现错误的主要原因是github中的README.md文件不在本地代码目录中

[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=4)

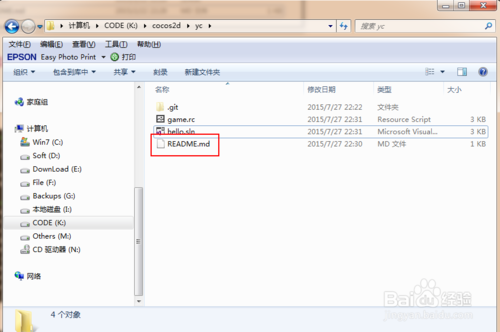
可以通过如下命令进行代码合并【注：pull=fetch+merge]

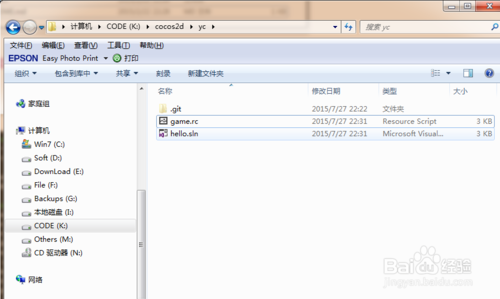
git pull --rebase origin master

[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=5)

5

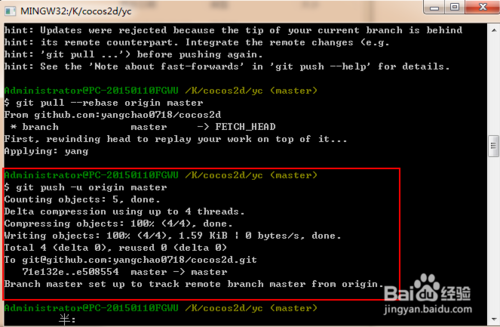
执行上面代码后可以看到本地代码库中多了README.md文件

[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=6)

[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=7)

6

此时再执行语句 git push -u origin master即可完成代码上传到github

[](http://jingyan.baidu.com/album/f3e34a12a25bc8f5ea65354a.html?picindex=8)

# git push & git pull 推送/拉取分支

git push与git pull是一对推送/拉取分支的git命令。   
****git push 使用本地的对应分支来更新对应的远程分支。****

$ git push <远程主机名> <本地分支名>:<远程分支名>

* 1

注意: 命令中的本地分支是指将要被推送到远端的分支，而远程分支是指推送的目标分支，即将本地分支合并到远程分支。   
如果省略远程分支名，则表示将本地分支推送与之存在”追踪关系”的远程分支(通常两者同名)，如果该远程分支不存在，则会被新建。

$ git push origin master

* 1

上面命令表示，将本地的master分支推送到origin主机的master分支。如果后者不存在，则会被新建。   
origin是一个远程厂库地址。

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支，这条命令是删除远程master分支。

$ git push origin :master# 等同于$ git push origin --delete master

* 1
* 2
* 3

上面命令表示删除origin主机的master分支。

如果当前分支与远程分支之间存在追踪关系（即分支名相同），则本地分支和远程分支都可以省略。

$ git push origin

* 1

上面命令表示，将当前分支推送到origin主机的对应分支。

如果当前分支只有一个追踪分支，那么主机名都可以省略。

$ git push

* 1

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。

$ git push -u origin master

* 1

上面命令将本地的master分支推送到origin主机，同时指定origin为默认主机，后面就可以不加任何参数使用git push了。

不带任何参数的git push，默认只推送当前分支，这叫做simple方式。此外，还有一种matching方式，会推送所有有对应的远程分支的本地分支。Git 2.0版本之前，默认采用matching方法，现在改为默认采用simple方式。如果要修改这个设置，可以采用git config命令。

$ git config --global push.default matching# 或者

$ git config --global push.default simple

* 1
* 2
* 3

还有一种情况，就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要使用–all选项。

$ git push --all origin

* 1

上面命令表示，将所有本地分支都推送到origin主机。

如果远程主机的版本比本地版本更新，推送时Git会报错，要求先在本地做git pull合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用–force选项。

$ git push --force origin

* 1

上面命令使用–force选项，结果导致在远程主机产生一个”非直进式”的合并(non-fast-forward merge)。除非你很确定要这样做，否则应该尽量避免使用–force选项。

最后，git push不会推送标签(tag)，除非使用–tags选项。

$ git push origin --tags

* 1

****git pull 获取并合并其他的厂库，或者本地的其他分支。****

git pull 与 git push操作的目的相同，但是操作的目标相反。命令格式如下：

git pull <远程主机> <远程分支>:<本地分支>

* 1

例如：

git pull origin master:my\_test

* 1

上面的命令是将origin厂库的master分支拉取并合并到本地的my\_test分支上。

如果省略本地分支，则将自动合并到当前所在分支上。如下：

git pull origin master

* 1

注：如果你想参与github上的一些优秀的项目，则下面提供一个通用的例子：   
首先，需要一个github的账号，并fork一个你感兴趣的repository。   
下面描述过程中会涉及两个远程主分支，为了很好的区别，我们把fork出来的主分支称为远程A repository，本fork的分支称为远程B repository

$git clone <远程Arepository> #克隆你fork出来的分支

$git remote add <远程Brepository标签> git@github.com:XXXX/ceph.git #添加远程Brepository标签

$git pull <远程B厂库标签> master:master #从远程Brepository的master分支拉取最新objects合并到本地master分支

$git checkout YYYY #切换到要修改的分支上

$git branch develop; git checkout develop #在当前分支的基础上创建一个开发分支，并切换到该分支上，你将在该分支上coding

coding...... #在工作区coding

$git add .#将修改保存到索引区

$git commit -a #将修改提交到本地分区

$git push origin my\_test:my\_test #将本地分支my\_test提交到远程A repository的my\_test分支上

然后在github web界面上将my\_test分支合并到你需改的远程B repository 分支上。等待管理员review，如果有问题，就继续在develop分支当修改，并commit –amend，在之前的commit上修改。知道被meger。

Gitk

# **使用 Gitk 跟踪 Git 项目源代码**

从subversion到git可能是一种挣扎，试图理解像checkout、commit、branch、remote、rebase这些术语在git世界中意味着什么。我通过在演示库中进行实验，尝试各种命令，并使用gitk来可视化它们的影响。这篇文章分成两部分——读完后，你可能想读第二部分。

gitk屏幕

我在github上创建了一个简单的存储库，以遍历一些场景。我将从创建存储库的本地副本开始:

d:代码> git克隆git@github.com:joshuaflanagan / gitk-demo.git

在d:/code/gitk-demo/.git中初始化了空的Git仓库。

远程:计数对象:9，完成。

远程:压缩对象:100%(4/4)，完成。

远程:总计9(0)，重用0 (0)

接收对象:100%(9/9)，完成。

d:代码> cd gitk-demo

d:codegitk-demo > gitk——所有

gitk概述

d:code>git clone git@github.com:joshuaflanagan/gitk-demo.git

Initialized empty Git repository in d:/code/gitk-demo/.git/

remote: Counting objects: 9, done.

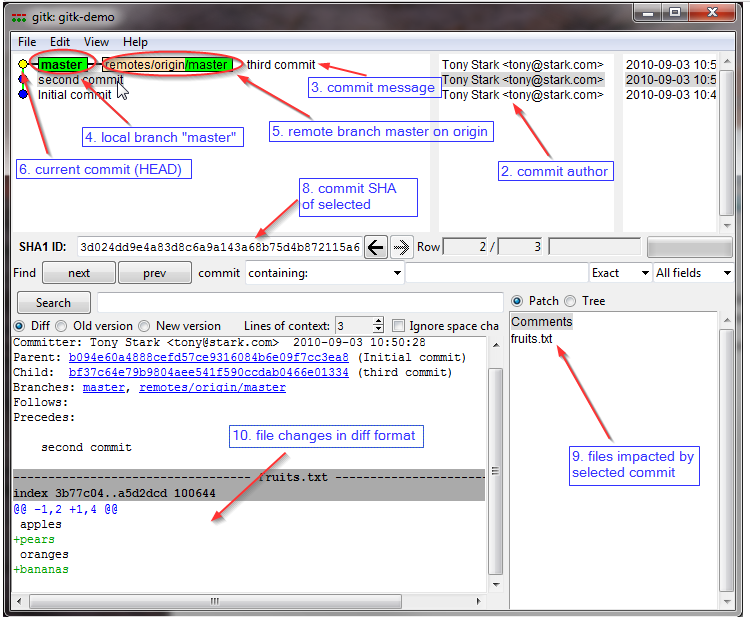
remote: Compressing objects: 100% (4/4), done.

remote: Total 9 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (9/9), done.

d:code>cd gitk-demo

d:codegitk-demo>gitk --all



这张截图里有很多信息:

左上角的窗格显示了对这个存储库的一系列提交，其中最新的是顶部。

托尼·斯塔克(Tony Stark)已经提交了三份报告。

最近提交的提交消息是“第三提交”

这里有一个本地分支，名为“master”，它指向最近的提交。

有一个远程引用分支:来自名为“origin”的远程存储库的“主”分支，它还指向最近的提交。

顶部的黄色圆点表示当前在工作文件夹中的快照(称为HEAD)

我已经突出显示了第二个commit，以便在下窗格中看到它的详细信息。

提交沙(唯一标识符,类似于subversion修订号)第二次提交是3 d024dd9e4a83d8c6a9a143a68b75d4b872115a6

右下角显示受第二次提交影响的文件列表。

左下角显示提交细节，包括完整的diff。

单击右下方窗格中的文件，将左下方窗格中的diff滚动到对应的部分。

关于“master”和“origin”的说明

当您第一次创建一个git存储库时，它从一个名为“master”的分支开始。这个分支没有什么特别之处，除了它是默认的。您可以自由创建一个新的，并且删除master(尽管，我没有看到任何违背默认约定的理由)。

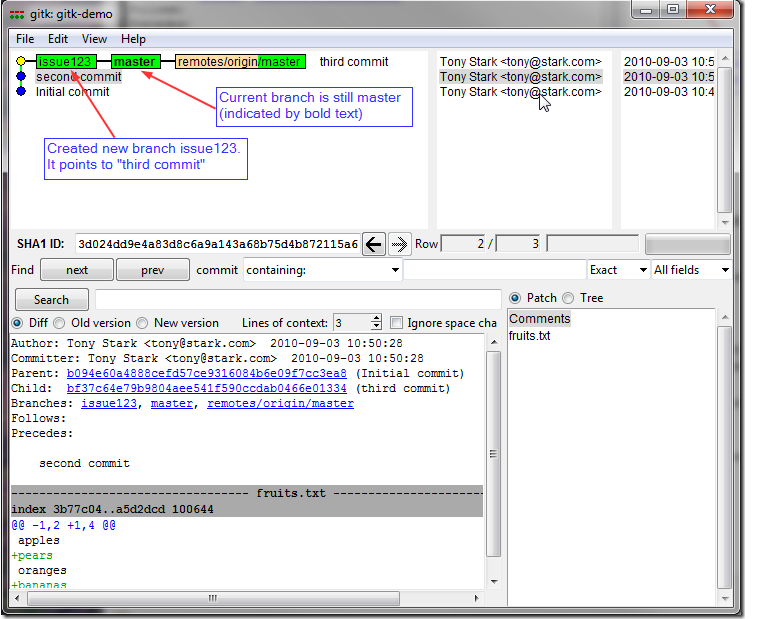
当您第一次克隆一个git存储库时，git将自动为您创建一个名为“origin”的远程存储库。remote只是用来管理引用(url)到其他存储库的名称。“原点”的遥控器没有什么特别之处，除了它是为你创建的。您可以自由创建一个新的和删除原点。事实上，如果您正在使用多个remotes，我建议您删除原点远程，并为同一个存储库创建一个新的，但是使用一个更具描述性的名称。例如，当我使用FubuMVC源代码时，在本地的存储库中，我有一个远程命名的“darth”，它指的是DarthFubuMVC拥有的主存储库，还有一个名为“josh”的远程存储库，它指的是我的fork。如果我保留了“起源”这个名字，我将永远记住我克隆的是哪一个。

分支

当我创建一个分支时会发生什么?

d:codegitk-demo > issue123 git分支

在gitk窗口中按下CTRL-F5刷新存储库视图。



创建分支

我们看到了新的分支标记，它指向了与master和origina/master相同的提交。需要注意的是，“master”是粗体，表示它仍然是当前分支。粗体分支标签相当于命令行输出中的星号:

d:codegitk-demo > git分支

issue123

\*主

现在，如果我切换到新的分支并刷新gitk:

d:codegitk-demo > git checkout issue123

切换到分支“issue123”

(从现在开始，我们将重点关注顶部窗格中的信息，因此我将隐藏gitk的底部部分)

未完待续

# **[Git常用命令](http://www.cnblogs.com/vman/articles/Git_cmds.html)**

**查看、添加、提交、删除、找回，重置修改文件**

git help <command> # 显示command的help

git show # 显示某次提交的内容 git show $id

git co -- <file> # 抛弃工作区修改

git co . # 抛弃工作区修改

git add <file> # 将工作文件修改提交到本地暂存区

git add . # 将所有修改过的工作文件提交暂存区

git rm <file> # 从版本库中删除文件

git rm <file> --cached # 从版本库中删除文件，但不删除文件

git reset <file> # 从暂存区恢复到工作文件

git reset -- . # 从暂存区恢复到工作文件

git reset --hard # 恢复最近一次提交过的状态，即放弃上次提交后的所有本次修改

git ci <file> git ci . git ci -a # 将git add, git rm和git ci等操作都合并在一起做　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　git ci -am "some comments"

git ci --amend # 修改最后一次提交记录

git revert <$id> # 恢复某次提交的状态，恢复动作本身也创建次提交对象

git revert HEAD # 恢复最后一次提交的状态

**查看文件diff**

git diff <file> # 比较当前文件和暂存区文件差异 git diff

git diff <id1><id1><id2> # 比较两次提交之间的差异

git diff <branch1>..<branch2> # 在两个分支之间比较

git diff --staged # 比较暂存区和版本库差异

git diff --cached # 比较暂存区和版本库差异

git diff --stat # 仅仅比较统计信息

**查看提交记录**

git log git log <file> # 查看该文件每次提交记录

git log -p <file> # 查看每次详细修改内容的diff

git log -p -2 # 查看最近两次详细修改内容的diff

git log --stat #查看提交统计信息

**tig**

Mac上可以使用tig代替diff和log，brew install tig

**Git 本地分支管理**

**查看、切换、创建和删除分支**

git br -r # 查看远程分支

git br <new\_branch> # 创建新的分支

git br -v # 查看各个分支最后提交信息

git br --merged # 查看已经被合并到当前分支的分支

git br --no-merged # 查看尚未被合并到当前分支的分支

git co <branch> # 切换到某个分支

git co -b <new\_branch> # 创建新的分支，并且切换过去

git co -b <new\_branch> <branch> # 基于branch创建新的new\_branch

git co $id # 把某次历史提交记录checkout出来，但无分支信息，切换到其他分支会自动删除

git co $id -b <new\_branch> # 把某次历史提交记录checkout出来，创建成一个分支

git br -d <branch> # 删除某个分支

git br -D <branch> # 强制删除某个分支 (未被合并的分支被删除的时候需要强制)

**分支合并和rebase**

git merge <branch> # 将branch分支合并到当前分支

git merge origin/master --no-ff # 不要Fast-Foward合并，这样可以生成merge提交

git rebase master <branch> # 将master rebase到branch，相当于： git co <branch> && git rebase master && git co master && git merge <branch>

**Git补丁管理(方便在多台机器上开发同步时用)**

git diff > ../sync.patch # 生成补丁

git apply ../sync.patch # 打补丁

git apply --check ../sync.patch #测试补丁能否成功

**Git暂存管理**

git stash # 暂存

git stash list # 列所有stash

git stash apply # 恢复暂存的内容

git stash drop # 删除暂存区

**Git远程分支管理**

git pull # 抓取远程仓库所有分支更新并合并到本地

git pull --no-ff # 抓取远程仓库所有分支更新并合并到本地，不要快进合并

git fetch origin # 抓取远程仓库更新

git merge origin/master # 将远程主分支合并到本地当前分支

git co --track origin/branch # 跟踪某个远程分支创建相应的本地分支

git co -b <local\_branch> origin/<remote\_branch> # 基于远程分支创建本地分支，功能同上

git push # push所有分支

git push origin master # 将本地主分支推到远程主分支

git push -u origin master # 将本地主分支推到远程(如无远程主分支则创建，用于初始化远程仓库)

git push origin <local\_branch> # 创建远程分支， origin是远程仓库名

git push origin <local\_branch>:<remote\_branch> # 创建远程分支

git push origin :<remote\_branch> #先删除本地分支(git br -d <branch>)，然后再push删除远程分支

**Git远程仓库管理**

[GitHub](http://blog.jobbole.com/6492/" \o "GitHub如何运作：时间并不决定一切" \t "https://www.cnblogs.com/cspku/articles/_blank)

git remote -v # 查看远程服务器地址和仓库名称

git remote show origin # 查看远程服务器仓库状态

git remote add origin git@ github:robbin/robbin\_site.git # 添加远程仓库地址

git remote set-url origin git@ github.com:robbin/robbin\_site.git # 设置远程仓库地址(用于修改远程仓库地址) git remote rm <repository> # 删除远程仓库

**创建远程仓库**

git clone --bare robbin\_site robbin\_site.git # 用带版本的项目创建纯版本仓库

scp -r my\_project.git git@ git.csdn.net:~ # 将纯仓库上传到服务器上

mkdir robbin\_site.git && cd robbin\_site.git && git --bare init # 在服务器创建纯仓库

git remote add origin git@ github.com:robbin/robbin\_site.git # 设置远程仓库地址

git push -u origin master # 客户端首次提交

git push -u origin develop # 首次将本地develop分支提交到远程develop分支，并且track

git remote set-head origin master # 设置远程仓库的HEAD指向master分支

也可以命令设置跟踪远程库和本地库

git branch --set-upstream master origin/master

git branch --set-upstream develop origin/develop