# 廖雪峰的Git教程

阅读: 4765657



史上最浅显易懂的Git教程！

为什么要编写这个教程？因为我在学习Git的过程中，买过书，也在网上Google了一堆Git相关的文章和教程，但令人失望的是，这些教程不是难得令人发指，就是简单得一笔带过，或者，只支离破碎地介绍Git的某几个命令，还有直接从Git手册粘贴帮助文档的，总之，初学者很难找到一个由浅入深，学完后能立刻上手的Git教程。

既然号称史上最浅显易懂的Git教程，那这个教程有什么让你怦然心动的特点呢？

首先，本教程绝对面向初学者，没有接触过版本控制概念的读者也可以轻松入门，不必担心起步难度；

其次，本教程实用性超强，边学边练，一点也不觉得枯燥。而且，你所学的Git命令是“充分且必要”的，掌握了这些东西，你就可以通过Git轻松地完成你的工作。

文字+图片还看不明白？有视频！！！

本教程只会让你成为Git用户，不会让你成为Git专家。很多Git命令只有那些专家才明白（事实上我也不明白，因为我不是Git专家），但我保证这些命令可能你一辈子都不会用到。既然Git是一个工具，就没必要把时间浪费在那些“高级”但几乎永远不会用到的命令上。一旦你真的非用不可了，到时候再自行Google或者请教专家也未迟。

如果你是一个开发人员，想用上这个世界上目前最先进的分布式版本控制系统，那么，赶快开始学习吧！

关于作者

[廖雪峰](http://weibo.com/liaoxuefeng)，十年软件开发经验，业余产品经理，精通Java/Python/Ruby/Visual Basic/Objective C等，对开源框架有深入研究，著有《Spring 2.0核心技术与最佳实践》一书，多个业余开源项目托管在[GitHub](https://github.com/michaelliao)，欢迎微博交流：

## Git简介

阅读: 672517

Git是什么？

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

Git有什么特点？简单来说就是：高端大气上档次！

那什么是版本控制系统？

如果你用Microsoft Word写过长篇大论，那你一定有这样的经历：

想删除一个段落，又怕将来想恢复找不回来怎么办？有办法，先把当前文件“另存为……”一个新的Word文件，再接着改，改到一定程度，再“另存为……”一个新文件，这样一直改下去，最后你的Word文档变成了这样：



过了一周，你想找回被删除的文字，但是已经记不清删除前保存在哪个文件里了，只好一个一个文件去找，真麻烦。

看着一堆乱七八糟的文件，想保留最新的一个，然后把其他的删掉，又怕哪天会用上，还不敢删，真郁闷。

更要命的是，有些部分需要你的财务同事帮助填写，于是你把文件Copy到U盘里给她（也可能通过Email发送一份给她），然后，你继续修改Word文件。一天后，同事再把Word文件传给你，此时，你必须想想，发给她之后到你收到她的文件期间，你作了哪些改动，得把你的改动和她的部分合并，真困难。

于是你想，如果有一个软件，不但能自动帮我记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆类似的文件了，也不需要把文件传来传去。如果想查看某次改动，只需要在软件里瞄一眼就可以，岂不是很方便？

这个软件用起来就应该像这个样子，能记录每次文件的改动：

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **用户** | **说明** | **日期** |
| 1 | 张三 | 删除了软件服务条款5 | 7/12 10:38 |
| 2 | 张三 | 增加了License人数限制 | 7/12 18:09 |
| 3 | 李四 | 财务部门调整了合同金额 | 7/13 9:51 |
| 4 | 张三 | 延长了免费升级周期 | 7/14 15:17 |

这样，你就结束了手动管理多个“版本”的史前时代，进入到版本控制的20世纪。

Git的诞生

阅读: 587201

很多人都知道，Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。

Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！

你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

集中式vs分布式

阅读: 568784

Linus一直痛恨的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。



当然，Git的优势不单是不必联网这么简单，后面我们还会看到Git极其强大的分支管理，把SVN等远远抛在了后面。

CVS作为最早的开源而且免费的集中式版本控制系统，直到现在还有不少人在用。由于CVS自身设计的问题，会造成提交文件不完整，版本库莫名其妙损坏的情况。同样是开源而且免费的SVN修正了CVS的一些稳定性问题，是目前用得最多的集中式版本库控制系统。

除了免费的外，还有收费的集中式版本控制系统，比如IBM的ClearCase（以前是Rational公司的，被IBM收购了），特点是安装比Windows还大，运行比蜗牛还慢，能用ClearCase的一般是世界500强，他们有个共同的特点是财大气粗，或者人傻钱多。

微软自己也有一个集中式版本控制系统叫VSS，集成在Visual Studio中。由于其反人类的设计，连微软自己都不好意思用了。

分布式版本控制系统除了Git以及促使Git诞生的BitKeeper外，还有类似Git的Mercurial和Bazaar等。这些分布式版本控制系统各有特点，但最快、最简单也最流行的依然是Git！

## 安装Git

阅读: 1134070

最早Git是在Linux上开发的，很长一段时间内，Git也只能在Linux和Unix系统上跑。不过，慢慢地有人把它移植到了Windows上。现在，Git可以在Linux、Unix、Mac和Windows这几大平台上正常运行了。

要使用Git，第一步当然是安装Git了。根据你当前使用的平台来阅读下面的文字：

在Linux上安装Git

首先，你可以试着输入git，看看系统有没有安装Git：

$ git

The program 'git' is currently not installed. You can install it by typing:

sudo apt-get install git

像上面的命令，有很多Linux会友好地告诉你Git没有安装，还会告诉你如何安装Git。

如果你碰巧用Debian或Ubuntu Linux，通过一条sudo apt-get install git就可以直接完成Git的安装，非常简单。

老一点的Debian或Ubuntu Linux，要把命令改为sudo apt-get install git-core，因为以前有个软件也叫GIT（GNU Interactive Tools），结果Git就只能叫git-core了。由于Git名气实在太大，后来就把GNU Interactive Tools改成gnuit，git-core正式改为git。

如果是其他Linux版本，可以直接通过源码安装。先从Git官网下载源码，然后解压，依次输入：./config，make，sudo make install这几个命令安装就好了。

在Mac OS X上安装Git

如果你正在使用Mac做开发，有两种安装Git的方法。

一是安装homebrew，然后通过homebrew安装Git，具体方法请参考homebrew的文档：<http://brew.sh/>。

第二种方法更简单，也是推荐的方法，就是直接从AppStore安装Xcode，Xcode集成了Git，不过默认没有安装，你需要运行Xcode，选择菜单“Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择“Command Line Tools”，点“Install”就可以完成安装了。



Xcode是Apple官方IDE，功能非常强大，是开发Mac和iOS App的必选装备，而且是免费的！

在Windows上安装Git

实话实说，Windows是最烂的开发平台，如果不是开发Windows游戏或者在IE里调试页面，一般不推荐用Windows。不过，既然已经上了微软的贼船，也是有办法安装Git的。

Windows下要使用很多Linux/Unix的工具时，需要Cygwin这样的模拟环境，Git也一样。Cygwin的安装和配置都比较复杂，就不建议你折腾了。不过，有高人已经把模拟环境和Git都打包好了，名叫msysgit，只需要下载一个单独的exe安装程序，其他什么也不用装，绝对好用。

msysgit是Windows版的Git，从<https://git-for-windows.github.io>下载（网速慢的同学请移步[国内镜像](https://pan.baidu.com/s/1kU5OCOB#list/path=%2Fpub%2Fgit)），然后按默认选项安装即可。

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！



安装完成后，还需要最后一步设置，在命令行输入：

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

注意git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

## 创建版本库

阅读: 1312176

什么是版本库呢？版本库又名仓库，英文名**repository**，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

$ mkdir learngit

$ cd learngit

$ pwd

/Users/michael/learngit

pwd命令用于显示当前目录。在我的Mac上，这个仓库位于/Users/michael/learngit。

如果你使用Windows系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

第二步，通过git init命令把这个目录变成Git可以管理的仓库：

$ git init

Initialized empty Git repository in /Users/michael/learngit/.git/

瞬间Git就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），细心的读者可以发现当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

如果你没有看到.git目录，那是因为这个目录默认是隐藏的，用ls -ah命令就可以看见。

也不一定必须在空目录下创建Git仓库，选择一个已经有东西的目录也是可以的。不过，不建议你使用自己正在开发的公司项目来学习Git，否则造成的一切后果概不负责。

## 把文件添加到版本库

首先这里再明确一下，所有的版本控制系统，其实只能跟踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。版本控制系统可以告诉你每次的改动，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

不幸的是，Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

因为文本是有编码的，比如中文有常用的GBK编码，日文有Shift\_JIS编码，如果没有历史遗留问题，强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

使用Windows的童鞋要特别注意：

千万不要使用Windows自带的**记事本**编辑任何文本文件。原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载[Notepad++](http://notepad-plus-plus.org/)代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可：



言归正传，现在我们编写一个readme.txt文件，内容如下：

Git is a version control system.

Git is free software.

一定要放到learngit目录下（子目录也行），因为这是一个Git仓库，放到其他地方Git再厉害也找不到这个文件。

和把大象放到冰箱需要3步相比，把一个文件放到Git仓库只需要两步。

第一步，用命令git add告诉Git，把文件添加到仓库：

$ git add readme.txt

执行上面的命令，没有任何显示，这就对了，Unix的哲学是“没有消息就是好消息”，说明添加成功。

第二步，用命令git commit告诉Git，把文件提交到仓库：

$ git commit -m "wrote a readme file"

[master (root-commit) cb926e7] wrote a readme file

1 file changed, 2 insertions(+)

create mode 100644 readme.txt

简单解释一下git commit命令，-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

嫌麻烦不想输入-m "xxx"行不行？确实有办法可以这么干，但是强烈不建议你这么干，因为输入说明对自己对别人阅读都很重要。实在不想输入说明的童鞋请自行Google，我不告诉你这个参数。

git commit命令执行成功后会告诉你，1个文件被改动（我们新添加的readme.txt文件），插入了两行内容（readme.txt有两行内容）。

为什么Git添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：

$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "add 3 files."

小结

现在总结一下今天学的两点内容：

初始化一个Git仓库，使用git init命令。

添加文件到Git仓库，分两步：

* 第一步，使用命令git add <file>，注意，可反复多次使用，添加多个文件；
* 第二步，使用命令git commit，完成。

## 时光机穿梭

阅读: 763139

我们已经成功地添加并提交了一个readme.txt文件，现在，是时候继续工作了，于是，我们继续修改readme.txt文件，改成如下内容：

Git is a distributed version control system.

Git is free software.

现在，运行git status命令看看结果：

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")

git status命令可以让我们时刻掌握仓库当前的状态，上面的命令告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

虽然Git告诉我们readme.txt被修改了，但如果能看看具体修改了什么内容，自然是很好的。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的readme.txt，所以，需要用git diff这个命令看看：

$ git diff readme.txt

diff --git a/readme.txt b/readme.txt

index 46d49bf..9247db6 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,2 +1,2 @@

-Git is a version control system.

+Git is a distributed version control system.

Git is free software.

git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，我们在第一行添加了一个“distributed”单词。

知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是git add：

$ git add readme.txt

同样没有任何输出。在执行第二步git commit之前，我们再运行git status看看当前仓库的状态：

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: readme.txt

#

git status告诉我们，将要被提交的修改包括readme.txt，下一步，就可以放心地提交了：

$ git commit -m "add distributed"

[master ea34578] add distributed

1 file changed, 1 insertion(+), 1 deletion(-)

提交后，我们再用git status命令看看仓库的当前状态：

$ git status

# On branch master

nothing to commit (working directory clean)

Git告诉我们当前没有需要提交的修改，而且，工作目录是干净（working directory clean）的。

小结

* 要随时掌握工作区的状态，使用git status命令。
* 如果git status告诉你有文件被修改过，用git diff可以查看修改内容。

### 版本回退

阅读: 843000

现在，你已经学会了修改文件，然后把修改提交到Git版本库，现在，再练习一次，修改readme.txt文件如下：

Git is a distributed version control system.

Git is free software distributed under the GPL.

然后尝试提交：

$ git add readme.txt

$ git commit -m "append GPL"

[master 3628164] append GPL

1 file changed, 1 insertion(+), 1 deletion(-)

像这样，你不断对文件进行修改，然后不断提交修改到版本库里，就好比玩RPG游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打Boss之前，你会手动存盘，以便万一打Boss失败了，可以从最近的地方重新开始。Git也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

现在，我们回顾一下readme.txt文件一共有几个版本被提交到Git仓库里了：

版本1：wrote a readme file

Git is a version control system.

Git is free software.

版本2：add distributed

Git is a distributed version control system.

Git is free software.

版本3：append GPL

Git is a distributed version control system.

Git is free software distributed under the GPL.

当然了，在实际工作中，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容，不然要版本控制系统干什么。版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用git log命令查看：

$ git log

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 15:11:49 2013 +0800

append GPL

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 14:53:12 2013 +0800

add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 19 17:51:55 2013 +0800

wrote a readme file

git log命令显示从最近到最远的提交日志，我们可以看到3次提交，最近的一次是append GPL，上一次是add distributed，最早的一次是wrote a readme file。 如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：

$ git log --pretty=oneline

3628164fb26d48395383f8f31179f24e0882e1e0 append GPL

ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed

cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file

需要友情提示的是，你看到的一大串类似3628164...882e1e0的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

每提交一个新版本，实际上Git就会把它们自动串成一条时间线。如果使用可视化工具查看Git历史，就可以更清楚地看到提交历史的时间线：



好了，现在我们启动时光穿梭机，准备把readme.txt回退到上一个版本，也就是“add distributed”的那个版本，怎么做呢？

首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交3628164...882e1e0（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

现在，我们要把当前版本“append GPL”回退到上一个版本“add distributed”，就可以使用git reset命令：

$ git reset --hard HEAD^

HEAD is now at ea34578 add distributed

--hard参数有啥意义？这个后面再讲，现在你先放心使用。

看看readme.txt的内容是不是版本add distributed：

$ cat readme.txt

Git is a distributed version control system.

Git is free software.

果然。

还可以继续回退到上一个版本wrote a readme file，不过且慢，然我们用git log再看看现在版本库的状态：

$ git log

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 14:53:12 2013 +0800

add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 19 17:51:55 2013 +0800

wrote a readme file

最新的那个版本append GPL已经看不到了！好比你从21世纪坐时光穿梭机来到了19世纪，想再回去已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个append GPL的commit id是3628164...，于是就可以指定回到未来的某个版本：

$ git reset --hard 3628164

HEAD is now at 3628164 append GPL

版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

再小心翼翼地看看readme.txt的内容：

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

果然，我胡汉三又回来了。

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD从指向append GPL：



改为指向add distributed：



然后顺便把工作区的文件更新了。所以你让HEAD指向哪个版本号，你就把当前版本定位在哪。

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的commit id怎么办？

在Git中，总是有后悔药可以吃的。当你用$ git reset --hard HEAD^回退到add distributed版本时，再想恢复到append GPL，就必须找到append GPL的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：

$ git reflog

ea34578 HEAD@{0}: reset: moving to HEAD^

3628164 HEAD@{1}: commit: append GPL

ea34578 HEAD@{2}: commit: add distributed

cb926e7 HEAD@{3}: commit (initial): wrote a readme file

终于舒了口气，第二行显示append GPL的commit id是3628164，现在，你又可以乘坐时光机回到未来了。

小结

现在总结一下：

* HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。
* 穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。
* 要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

### 工作区和暂存区

阅读: 555152

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。

先来看名词解释。

### 工作区（Working Directory）

就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：



### 版本库（Repository）

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



分支和HEAD的概念我们以后再讲。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

俗话说，实践出真知。现在，我们再练习一遍，先对readme.txt做个修改，比如加上一行内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

然后，在工作区新增一个LICENSE文本文件（内容随便写）。

先用git status查看一下状态：

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# LICENSE

no changes added to commit (use "git add" and/or "git commit -a")

Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。

现在，使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: LICENSE

# modified: readme.txt

#

现在，暂存区的状态就变成这样了：



所以，git add命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行git commit就可以一次性把暂存区的所有修改提交到分支。

$ git commit -m "understand how stage works"

[master 27c9860] understand how stage works

2 files changed, 675 insertions(+)

create mode 100644 LICENSE

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

$ git status

# On branch master

nothing to commit (working directory clean)

现在版本库变成了这样，暂存区就没有任何内容了：



小结

暂存区是Git非常重要的概念，弄明白了暂存区，就弄明白了Git的很多操作到底干了什么。

没弄明白暂存区是怎么回事的童鞋，请向上滚动页面，再看一次。

### 管理修改

阅读: 491139

现在，假定你已经完全掌握了暂存区的概念。下面，我们要讨论的就是，为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

为什么说Git管理的是修改，而不是文件呢？我们还是做实验。第一步，对readme.txt做一个修改，比如加一行内容：

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes.

然后，添加：

$ git add readme.txt

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: readme.txt

#

然后，再修改readme.txt：

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

提交：

$ git commit -m "git tracks changes"

[master d4f25b6] git tracks changes

1 file changed, 1 insertion(+)

提交后，再看看状态：

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")

咦，怎么第二次的修改没有被提交？

别激动，我们回顾一下操作过程：

第一次修改 -> git add -> 第二次修改 -> git commit

你看，我们前面讲了，Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，用git diff HEAD -- readme.txt命令可以查看工作区和版本库里面最新版本的区别：

$ git diff HEAD -- readme.txt

diff --git a/readme.txt b/readme.txt

index 76d770f..a9c5755 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,4 +1,4 @@

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

-Git tracks changes.

+Git tracks changes of files.

可见，第二次修改确实没有被提交。

那怎么提交第二次修改呢？你可以继续git add再git commit，也可以别着急提交第一次修改，先git add第二次修改，再git commit，就相当于把两次修改合并后一块提交了：

第一次修改 -> git add -> 第二次修改 -> git add -> git commit

好，现在，把第二次修改提交了，然后开始小结。

小结

现在，你又理解了Git是如何跟踪修改的，每次修改，如果不add到暂存区，那就不会加入到commit中。

### 撤销修改

阅读: 479496

自然，你是不会犯错的。不过现在是凌晨两点，你正在赶一份工作报告，你在readme.txt中添加了一行：

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

在你准备提交前，一杯咖啡起了作用，你猛然发现了“stupid boss”可能会让你丢掉这个月的奖金！

既然错误发现得很及时，就可以很容易地纠正它。你可以删掉最后一行，手动把文件恢复到上一个版本的状态。如果用git status查看一下：

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")

你可以发现，Git会告诉你，git checkout -- file可以丢弃工作区的修改：

$ git checkout -- readme.txt

命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是readme.txt已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次git commit或git add时的状态。

现在，看看readme.txt的文件内容：

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

文件内容果然复原了。

git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到git checkout命令。

现在假定是凌晨3点，你不但写了一些胡话，还git add到暂存区了：

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

$ git add readme.txt

庆幸的是，在commit之前，你发现了这个问题。用git status查看一下，修改只是添加到了暂存区，还没有提交：

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: readme.txt

#

Git同样告诉我们，用命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区：

$ git reset HEAD readme.txt

Unstaged changes after reset:

M readme.txt

git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

再用git status查看一下，现在暂存区是干净的，工作区有修改：

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")

还记得如何丢弃工作区的修改吗？

$ git checkout -- readme.txt

$ git status

# On branch master

nothing to commit (working directory clean)

整个世界终于清静了！

现在，假设你不但改错了东西，还从暂存区提交到了版本库，怎么办呢？还记得[版本回退](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)一节吗？可以回退到上一个版本。不过，这是有条件的，就是你还没有把自己的本地版本库推送到远程。还记得Git是分布式版本控制系统吗？我们后面会讲到远程版本库，一旦你把“stupid boss”提交推送到远程版本库，你就真的惨了……

小结

又到了小结时间。

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考[版本回退](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)一节，不过前提是没有推送到远程库。

### 删除文件

阅读: 390798

在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：

$ git add test.txt

$ git commit -m "add test.txt"

[master 94cdc44] add test.txt

1 file changed, 1 insertion(+)

create mode 100644 test.txt

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：

$ rm test.txt

这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了：

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add/rm <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# deleted: test.txt

#

no changes added to commit (use "git add" and/or "git commit -a")

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

$ git rm test.txt

rm 'test.txt'

$ git commit -m "remove test.txt"

[master d17efd8] remove test.txt

1 file changed, 1 deletion(-)

delete mode 100644 test.txt

现在，文件就从版本库中被删除了。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

$ git checkout -- test.txt

git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

小结

命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失**最近一次提交后你修改的内容**。

## 远程仓库

阅读: 617182

到目前为止，我们已经掌握了如何在Git仓库里对一个文件进行时光穿梭，你再也不用担心文件备份或者丢失的问题了。

可是有用过集中式版本控制系统SVN的童鞋会站出来说，这些功能在SVN里早就有了，没看出Git有什么特别的地方。

没错，如果只是在一个仓库里管理文件历史，Git和SVN真没啥区别。为了保证你现在所学的Git物超所值，将来绝对不会后悔，同时为了打击已经不幸学了SVN的童鞋，本章开始介绍Git的杀手级功能之一（注意是之一，也就是后面还有之二，之三……）：远程仓库。

Git是分布式版本控制系统，同一个Git仓库，可以分布到不同的机器上。怎么分布呢？最早，肯定只有一台机器有一个原始版本库，此后，别的机器可以“克隆”这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分。

你肯定会想，至少需要两台机器才能玩远程库不是？但是我只有一台电脑，怎么玩？

其实一台电脑上也是可以克隆多个版本库的，只要不在同一个目录下。不过，现实生活中是不会有人这么傻的在一台电脑上搞几个远程库玩，因为一台电脑上搞几个远程库完全没有意义，而且硬盘挂了会导致所有库都挂掉，所以我也不告诉你在一台电脑上怎么克隆多个仓库。

实际情况往往是这样，找一台电脑充当服务器的角色，每天24小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

完全可以自己搭建一台运行Git的服务器，不过现阶段，为了学Git先搭个服务器绝对是小题大作。好在这个世界上有个叫[GitHub](https://github.com/)的神奇的网站，从名字就可以看出，这个网站就是提供Git仓库托管服务的，所以，只要注册一个GitHub账号，就可以免费获得Git远程仓库。

在继续阅读后续内容前，请自行注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

$ ssh-keygen -t rsa -C "youremail@example.com"

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容：



点“Add Key”，你就应该看到已经添加的Key：



为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。

如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

确保你拥有一个GitHub账号后，我们就即将开始远程仓库的学习。

小结

“有了远程仓库，妈妈再也不用担心我的硬盘了。”——Git点读机

### 添加远程库

阅读: 879928

现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：



在Repository name填入learngit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的learngit仓库下运行命令：

$ git remote add origin git@github.com:michaelliao/learngit.git

请千万注意，把上面的michaelliao替换成你自己的GitHub账户名，否则，你在本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

添加后，远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

下一步，就可以把本地库的所有内容推送到远程库上：

$ git push -u origin master

Counting objects: 19, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (19/19), done.

Writing objects: 100% (19/19), 13.73 KiB, done.

Total 23 (delta 6), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [new branch] master -> master

Branch master set up to track remote branch master from origin.

把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：



从现在起，只要本地作了提交，就可以通过命令：

$ git push origin master

把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

### SSH警告

当你第一次使用Git的clone或者push命令连接GitHub时，会得到一个警告：

The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.

RSA key fingerprint is xx.xx.xx.xx.xx.

Are you sure you want to continue connecting (yes/no)?

这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

Warning: Permanently added 'github.com' (RSA) to the list of known hosts.

这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入yes前可以对照[GitHub的RSA Key的指纹信息](https://help.github.com/articles/what-are-github-s-ssh-key-fingerprints/)是否与SSH连接给出的一致。

小结

要关联一个远程库，使用命令git remote add origin git@server-name:path/repo-name.git；

关联后，使用命令git push -u origin master第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令git push origin master推送最新修改；

分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

### 从远程库克隆

阅读: 533969

上次我们讲了先有本地库，后有远程库的时候，如何关联远程库。

现在，假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

首先，登陆GitHub，创建一个新的仓库，名字叫gitskills：



我们勾选Initialize this repository with a README，这样GitHub会自动为我们创建一个README.md文件。创建完毕后，可以看到README.md文件：



现在，远程库已经准备好了，下一步是用命令git clone克隆一个本地库：

$ git clone git@github.com:michaelliao/gitskills.git

Cloning into 'gitskills'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (3/3), done.

$ cd gitskills

$ ls

README.md

注意把Git库的地址换成你自己的，然后进入gitskills目录看看，已经有README.md文件了。

如果有多个人协作开发，那么每个人各自从远程克隆一份就可以了。

你也许还注意到，GitHub给出的地址不止一个，还可以用https://github.com/michaelliao/gitskills.git这样的地址。实际上，Git支持多种协议，默认的git://使用ssh，但也可以使用https等其他协议。

使用https除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。

小结

要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆。

Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

注：by snow

从零开始从远程库clone，必须先在本地生成SSH Key，然后将SSH Key添加到github里，否则会提示权限拒绝。

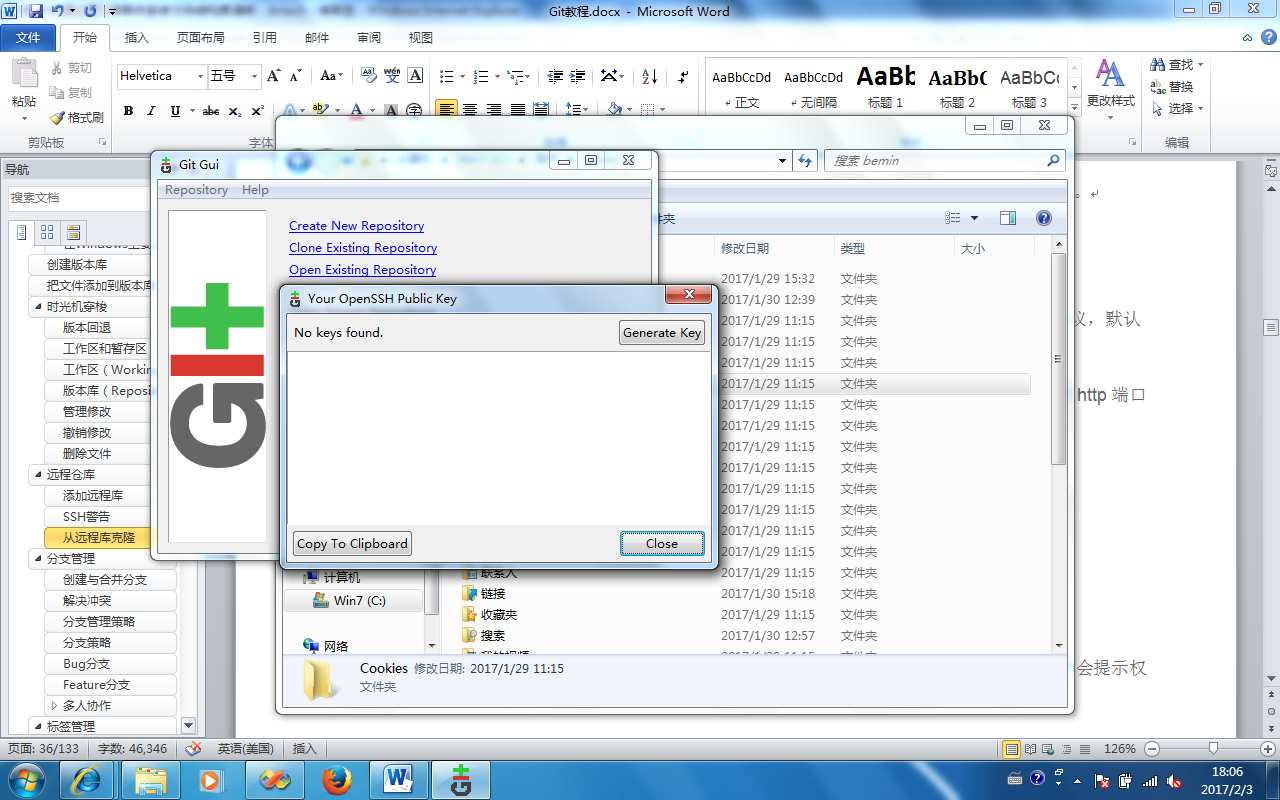
Warning: Permanently added 'github.com,192.30.253.112' (RSA) to the list of known hosts.

Permission denied (publickey).

fatal: Could not read from remote repository.

Please make sure you have the correct access rights and the repository exists.

生成SSH Key可以通过Git GUI中的Help🡪show SSH Key🡪generate SSH Key。如下图所示：



## 分支管理

阅读: 316480

分支就是科幻电影里面的平行宇宙，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SVN。

如果两个平行宇宙互不干扰，那对现在的你也没啥影响。不过，在某个时间点，两个平行宇宙合并了，结果，你既学会了Git又学会了SVN！



分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

但Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是1万个文件。

### 创建与合并分支

阅读: 680740

在[版本回退](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长：

当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：



你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：



真是太神奇了，你看得出来有些提交是通过分支完成的吗？

下面开始实战。

首先，我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行：

Creating a new branch is quick.

然后提交：

$ git add readme.txt

$ git commit -m "branch test"

[dev fec145a] branch test

1 file changed, 1 insertion(+)

现在，dev分支的工作完成，我们就可以切换回master分支：

$ git checkout master

Switched to branch 'master'

切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

Updating d17efd8..fec145a

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。

合并完成后，就可以放心地删除dev分支了：

$ git branch -d dev

Deleted branch dev (was fec145a).

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

小结

Git鼓励大量使用分支：

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建+切换分支：git checkout -b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

### 解决冲突

阅读: 409994

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的feature1分支，继续我们的新分支开发：

$ git checkout -b feature1

Switched to a new branch 'feature1'

修改readme.txt最后一行，改为：

Creating a new branch is quick AND simple.

在feature1分支上提交：

$ git add readme.txt

$ git commit -m "AND simple"

[feature1 75a857c] AND simple

1 file changed, 1 insertion(+), 1 deletion(-)

切换到master分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 1 commit.

Git还会自动提示我们当前master分支比远程的master分支要超前1个提交。

在master分支上把readme.txt文件的最后一行改为：

Creating a new branch is quick & simple.

提交：

$ git add readme.txt

$ git commit -m "& simple"

[master 400b400] & simple

1 file changed, 1 insertion(+), 1 deletion(-)

现在，master分支和feature1分支各自都分别有新的提交，变成了这样：



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict in readme.txt

Automatic merge failed; fix conflicts and then commit the result.

果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：

$ git status

# On branch master

# Your branch is ahead of 'origin/master' by 2 commits.

#

# Unmerged paths:

# (use "git add/rm <file>..." as appropriate to mark resolution)

#

# both modified: readme.txt

#

no changes added to commit (use "git add" and/or "git commit -a")

我们可以直接查看readme.txt的内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

Creating a new branch is quick AND simple.

>>>>>>> feature1

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：

Creating a new branch is quick and simple.

再提交：

$ git add readme.txt

$ git commit -m "conflict fixed"

[master 59bc1cb] conflict fixed

现在，master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支的合并情况：

$ git log --graph --pretty=oneline --abbrev-commit

\* 59bc1cb conflict fixed

|\

| \* 75a857c AND simple

\* | 400b400 & simple

|/

\* fec145a branch test

...

最后，删除feature1分支：

$ git branch -d feature1

Deleted branch feature1 (was 75a857c).

工作完成。

小结

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

用git log --graph命令可以看到分支合并图。

### 分支管理策略

阅读: 331976

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下--no-ff方式的git merge：

首先，仍然创建并切换dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

修改readme.txt文件，并提交一个新的commit：

$ git add readme.txt

$ git commit -m "add merge"

[dev 6224937] add merge

1 file changed, 1 insertion(+)

现在，我们切换回master：

$ git checkout master

Switched to branch 'master'

准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：

$ git merge --no-ff -m "merge with no-ff" dev

Merge made by the 'recursive' strategy.

readme.txt | 1 +

1 file changed, 1 insertion(+)

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并后，我们用git log看看分支历史：

$ git log --graph --pretty=oneline --abbrev-commit

\* 7825a50 merge with no-ff

|\

| \* 6224937 add merge

|/

\* 59bc1cb conflict fixed

...

可以看到，不使用Fast forward模式，merge后就像这样：



### 分支策略

在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

所以，团队合作的分支看起来就像这样：



小结

Git分支十分强大，在团队开发中应该充分应用。

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

### Bug分支

阅读: 246504

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交：

$ git status

# On branch dev

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: hello.py

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？

幸好，Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

$ git stash

Saved working directory and index state WIP on dev: 6224937 add merge

HEAD is now at 6224937 add merge

现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 6 commits.

$ git checkout -b issue-101

Switched to a new branch 'issue-101'

现在修复bug，需要把“Git is free software ...”改为“Git is a free software ...”，然后提交：

$ git add readme.txt

$ git commit -m "fix bug 101"

[issue-101 cc17032] fix bug 101

1 file changed, 1 insertion(+), 1 deletion(-)

修复完成后，切换到master分支，并完成合并，最后删除issue-101分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 2 commits.

$ git merge --no-ff -m "merged bug fix 101" issue-101

Merge made by the 'recursive' strategy.

readme.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

$ git branch -d issue-101

Deleted branch issue-101 (was cc17032).

太棒了，原计划两个小时的bug修复只花了5分钟！现在，是时候接着回到dev分支干活了！

$ git checkout dev

Switched to branch 'dev'

$ git status

# On branch dev

nothing to commit (working directory clean)

工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；

另一种方式是用git stash pop，恢复的同时把stash内容也删了：

$ git stash pop

# On branch dev

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: hello.py

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: readme.txt

#

Dropped refs/stash@{0} (f624f8e5f082f2df2bed8a4e09c12fd2943bdd40)

再用git stash list查看，就看不到任何stash内容了：

$ git stash list

你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：

$ git stash apply stash@{0}

小结

修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；

当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场。

### Feature分支

阅读: 207034

软件开发中，总有无穷无尽的新的功能要不断添加进来。

添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

现在，你终于接到了一个新任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船。

于是准备开发：

$ git checkout -b feature-vulcan

Switched to a new branch 'feature-vulcan'

5分钟后，开发完毕：

$ git add vulcan.py

$ git status

# On branch feature-vulcan

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: vulcan.py

#

$ git commit -m "add feature vulcan"

[feature-vulcan 756d4af] add feature vulcan

1 file changed, 2 insertions(+)

create mode 100644 vulcan.py

切回dev，准备合并：

$ git checkout dev

一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。

但是，

就在此时，接到上级命令，因经费不足，新功能必须取消！

虽然白干了，但是这个分支还是必须就地销毁：

$ git branch -d feature-vulcan

error: The branch 'feature-vulcan' is not fully merged.

If you are sure you want to delete it, run 'git branch -D feature-vulcan'.

销毁失败。Git友情提醒，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用命令git branch -D feature-vulcan。

现在我们强行删除：

$ git branch -D feature-vulcan

Deleted branch feature-vulcan (was 756d4af).

终于删除成功！

小结

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过git branch -D <name>强行删除。

### 多人协作

阅读: 353920

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

要查看远程库的信息，用git remote：

$ git remote

origin

或者，用git remote -v显示更详细的信息：

$ git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

#### 推送分支

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

$ git push origin master

如果要推送其他分支，比如dev，就改成：

$ git push origin dev

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

* master分支是主分支，因此要时刻与远程同步；
* dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
* bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
* feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

#### 抓取分支

多人协作时，大家都会往master和dev分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

$ git clone git@github.com:michaelliao/learngit.git

Cloning into 'learngit'...

remote: Counting objects: 46, done.

remote: Compressing objects: 100% (26/26), done.

remote: Total 46 (delta 16), reused 45 (delta 15)

Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.

Resolving deltas: 100% (16/16), done.

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：

$ git branch

\* master

现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

$ git checkout -b dev origin/dev

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

$ git commit -m "add /usr/bin/env"

[dev 291bea8] add /usr/bin/env

1 file changed, 1 insertion(+)

$ git push origin dev

Counting objects: 5, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 349 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

fc38031..291bea8 dev -> dev

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

$ git add hello.py

$ git commit -m "add coding: utf-8"

[dev bd6ae48] add coding: utf-8

1 file changed, 1 insertion(+)

$ git push origin dev

To git@github.com:michaelliao/learngit.git

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

error: failed to push some refs to 'git@github.com:michaelliao/learngit.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.

推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：

$ git pull

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From github.com:michaelliao/learngit

fc38031..291bea8 dev -> origin/dev

There is no tracking information for the current branch.

Please specify which branch you want to merge with.

See git-pull(1) for details

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream dev origin/<branch>

git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：

$ git branch --set-upstream dev origin/dev

Branch dev set up to track remote branch dev from origin.

再pull：

$ git pull

Auto-merging hello.py

CONFLICT (content): Merge conflict in hello.py

Automatic merge failed; fix conflicts and then commit the result.

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的[解决冲突](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001375840202368c74be33fbd884e71b570f2cc3c0d1dcf000)完全一样。解决后，提交，再push：

$ git commit -m "merge & fix hello.py"

[dev adca45d] merge & fix hello.py

$ git push origin dev

Counting objects: 10, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (6/6), 747 bytes, done.

Total 6 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

291bea8..adca45d dev -> dev

因此，多人协作的工作模式通常是这样：

1. 首先，可以试图用git push origin branch-name推送自己的修改；
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 没有冲突或者解决掉冲突后，再用git push origin branch-name推送就能成功！

如果git pull提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream branch-name origin/branch-name。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

小结

* 查看远程库信息，使用git remote -v；
* 本地新建的分支如果不推送到远程，对其他人就是不可见的；
* 从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交；
* 在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致；
* 建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name；
* 从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

## 标签管理

阅读: 145199

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

如果换一个办法：

“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

### 创建标签

阅读: 181127

在Git中打标签非常简单，首先，切换到需要打标签的分支上：

$ git branch

\* dev

master

$ git checkout master

Switched to branch 'master'

然后，敲命令git tag <name>就可以打一个新标签：

$ git tag v1.0

可以用命令git tag查看所有标签：

$ git tag

v1.0

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了：

$ git log --pretty=oneline --abbrev-commit

6a5819e merged bug fix 101

cc17032 fix bug 101

7825a50 merge with no-ff

6224937 add merge

59bc1cb conflict fixed

400b400 & simple

75a857c AND simple

fec145a branch test

d17efd8 remove test.txt

...

比方说要对add merge这次提交打标签，它对应的commit id是6224937，敲入命令：

$ git tag v0.9 6224937

再用命令git tag查看标签：

$ git tag

v0.9

v1.0

注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息：

$ git show v0.9

commit 622493706ab447b6bb37e4e2a2f276a20fed2ab4

Author: Michael Liao <askxuefeng@gmail.com>

Date: Thu Aug 22 11:22:08 2013 +0800

add merge

...

可以看到，v0.9确实打在add merge这次提交上。

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

$ git tag -a v0.1 -m "version 0.1 released" 3628164

用命令git show <tagname>可以看到说明文字：

$ git show v0.1

tag v0.1

Tagger: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 26 07:28:11 2013 +0800

version 0.1 released

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 15:11:49 2013 +0800

append GPL

还可以通过-s用私钥签名一个标签：

$ git tag -s v0.2 -m "signed version 0.2 released" fec145a

签名采用PGP签名，因此，必须首先安装gpg（GnuPG），如果没有找到gpg，或者没有gpg密钥对，就会报错：

gpg: signing failed: secret key not available

error: gpg failed to sign the data

error: unable to sign the tag

如果报错，请参考GnuPG帮助文档配置Key。

用命令git show <tagname>可以看到PGP签名信息：

$ git show v0.2

tag v0.2

Tagger: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 26 07:28:33 2013 +0800

signed version 0.2 released

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.12 (Darwin)

iQEcBAABAgAGBQJSGpMhAAoJEPUxHyDAhBpT4QQIAKeHfR3bo...

-----END PGP SIGNATURE-----

commit fec145accd63cdc9ed95a2f557ea0658a2a6537f

Author: Michael Liao <askxuefeng@gmail.com>

Date: Thu Aug 22 10:37:30 2013 +0800

branch test

用PGP签名的标签是不可伪造的，因为可以验证PGP签名。验证签名的方法比较复杂，这里就不介绍了。

小结

* 命令git tag <name>用于新建一个标签，默认为HEAD，也可以指定一个commit id；
* git tag -a <tagname> -m "blablabla..."可以指定标签信息；
* git tag -s <tagname> -m "blablabla..."可以用PGP签名标签；
* 命令git tag可以查看所有标签。

### 操作标签

阅读: 132425

如果标签打错了，也可以删除：

$ git tag -d v0.1

Deleted tag 'v0.1' (was e078af9)

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令git push origin <tagname>：

$ git push origin v1.0

Total 0 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [new tag] v1.0 -> v1.0

或者，一次性推送全部尚未推送到远程的本地标签：

$ git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 554 bytes, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [new tag] v0.2 -> v0.2

\* [new tag] v0.9 -> v0.9

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

$ git tag -d v0.9

Deleted tag 'v0.9' (was 6224937)

然后，从远程删除。删除命令也是push，但是格式如下：

$ git push origin :refs/tags/v0.9

To git@github.com:michaelliao/learngit.git

- [deleted] v0.9

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

小结

* 命令git push origin <tagname>可以推送一个本地标签；
* 命令git push origin --tags可以推送全部未推送过的本地标签；
* 命令git tag -d <tagname>可以删除一个本地标签；
* 命令git push origin :refs/tags/<tagname>可以删除一个远程标签。

## 使用GitHub

阅读: 607075

我们一直用GitHub作为免费的远程仓库，如果是个人的开源项目，放到GitHub上是完全没有问题的。其实GitHub还是一个开源协作社区，通过GitHub，既可以让别人参与你的开源项目，也可以参与别人的开源项目。

在GitHub出现以前，开源项目开源容易，但让广大人民群众参与进来比较困难，因为要参与，就要提交代码，而给每个想提交代码的群众都开一个账号那是不现实的，因此，群众也仅限于报个bug，即使能改掉bug，也只能把diff文件用邮件发过去，很不方便。

但是在GitHub上，利用Git极其强大的克隆和分支功能，广大人民群众真正可以第一次自由参与各种开源项目了。

如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页<https://github.com/twbs/bootstrap>，点“Fork”就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：

git clone git@github.com:michaelliao/bootstrap.git

一定要从自己的账号下clone仓库，这样你才能推送修改。如果从bootstrap的作者的仓库地址git@github.com:twbs/bootstrap.git克隆，因为没有权限，你将不能推送修改。

Bootstrap的官方仓库twbs/bootstrap、你在GitHub上克隆的仓库my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：



如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送。

如果你希望bootstrap的官方库能接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。

如果你没能力修改bootstrap，但又想要试一把pull request，那就Fork一下我的仓库：<https://github.com/michaelliao/learngit>，创建一个your-github-id.txt的文本文件，写点自己学习Git的心得，然后推送一个pull request给我，我会视心情而定是否接受。

小结

* 在GitHub上，可以任意Fork开源仓库；
* 自己拥有Fork后的仓库的读写权限；
* 可以推送pull request给官方仓库来贡献代码。

## 自定义Git

阅读: 157428

在[安装Git](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/00137396287703354d8c6c01c904c7d9ff056ae23da865a000)一节中，我们已经配置了user.name和user.email，实际上，Git还有很多可配置项。

比如，让Git显示颜色，会让命令输出看起来更醒目：

$ git config --global color.ui true

这样，Git会适当地显示不同的颜色，比如git status命令：



文件名就会标上颜色。

我们在后面还会介绍如何更好地配置Git，以便让你的工作更高效。

### 忽略特殊文件

有些时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件啦，等等，每次git status都会显示Untracked files ...，有强迫症的童鞋心里肯定不爽。

好在Git考虑到了大家的感受，这个问题解决起来也很简单，在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

不需要从头写.gitignore文件，GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>

忽略文件的原则是：

1. 忽略操作系统自动生成的文件，比如缩略图等；
2. 忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
3. 忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

举个例子：

假设你在Windows下进行Python开发，Windows会自动在有图片的目录下生成隐藏的缩略图文件，如果有自定义目录，目录下就会有Desktop.ini文件，因此你需要忽略Windows自动生成的垃圾文件：

# Windows:

Thumbs.db

ehthumbs.db

Desktop.ini

然后，继续忽略Python编译产生的.pyc、.pyo、dist等文件或目录：

# Python:

\*.py[cod]

\*.so

\*.egg

\*.egg-info

dist

build

1. 加上你自己定义的文件，最终得到一个完整的.gitignore文件，内容如下：

# Windows:

Thumbs.db

ehthumbs.db

Desktop.ini

# Python:

\*.py[cod]

\*.so

\*.egg

\*.egg-info

dist

build

# My configurations:

db.ini

deploy\_key\_rsa

最后一步就是把.gitignore也提交到Git，就完成了！当然检验.gitignore的标准是git status命令是不是说working directory clean。

使用Windows的童鞋注意了，如果你在资源管理器里新建一个.gitignore文件，它会非常弱智地提示你必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。

有些时候，你想添加一个文件到Git，但发现添加不了，原因是这个文件被.gitignore忽略了：

$ git add App.class

The following paths are ignored by one of your .gitignore files:

App.class

Use -f if you really want to add them.

如果你确实想添加该文件，可以用-f强制添加到Git：

$ git add -f App.class

或者你发现，可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用git check-ignore命令检查：

$ git check-ignore -v App.class

.gitignore:3:\*.class App.class

Git会告诉我们，.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

小结

* 忽略某些文件时，需要编写.gitignore；
* .gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理！

### 配置别名

阅读: 134970

有没有经常敲错命令？比如git status？status这个单词真心不好记。

如果敲git st就表示git status那就简单多了，当然这种偷懒的办法我们是极力赞成的。

我们只需要敲一行命令，告诉Git，以后st就表示status：

$ git config --global alias.st status

好了，现在敲git st看看效果。

当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch：

$ git config --global alias.co checkout

$ git config --global alias.ci commit

$ git config --global alias.br branch

以后提交就可以简写成：

$ git ci -m "bala bala bala..."

--global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

在[撤销修改](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001374831943254ee90db11b13d4ba9a73b9047f4fb968d000)一节中，我们知道，命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：

$ git config --global alias.unstage 'reset HEAD'

当你敲入命令：

$ git unstage test.py

实际上Git执行的是：

$ git reset HEAD test.py

配置一个git last，让其显示最后一次提交信息：

$ git config --global alias.last 'log -1'

这样，用git last就能显示最近一次的提交：

$ git last

commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2

Merge: bd6ae48 291bea8

Author: Michael Liao <askxuefeng@gmail.com>

Date: Thu Aug 22 22:49:22 2013 +0800

merge & fix hello.py

甚至还有人丧心病狂地把lg配置成了：

git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

来看看git lg的效果：



为什么不早点告诉我？别激动，咱不是为了多记几个英文单词嘛！

### 配置文件

配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中：

$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

ignorecase = true

precomposeunicode = true

[remote "origin"]

url = git@github.com:michaelliao/learngit.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

[alias]

last = log -1

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中：

$ cat .gitconfig

[alias]

co = checkout

ci = commit

br = branch

st = status

[user]

name = Your Name

email = your@email.com

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

小结

给Git配置好别名，就可以输入命令时偷个懒。我们鼓励偷懒。

### 搭建Git服务器

阅读: 423071

在[远程仓库](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001374385852170d9c7adf13c30429b9660d0eb689dd43a000)一节中，我们讲了远程仓库实际上和本地仓库没啥不同，纯粹为了7x24小时开机并交换大家的修改。

GitHub就是一个免费托管开源代码的远程仓库。但是对于某些视源代码如生命的商业公司来说，既不想公开源代码，又舍不得给GitHub交保护费，那就只能自己搭建一台Git服务器作为私有仓库使用。

搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的apt命令就可以完成安装。

假设你已经有sudo权限的用户账号，下面，正式开始安装。

第一步，安装git：

$ sudo apt-get install git

第二步，创建一个git用户，用来运行git服务：

$ sudo adduser git

第三步，创建证书登录：

收集所有需要登录的用户的公钥，就是他们自己的id\_rsa.pub文件，把所有公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

第四步，初始化Git仓库：

先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入命令：

$ sudo git init --bare sample.git

Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把owner改为git：

$ sudo chown -R git:git sample.git

第五步，禁用shell登录：

出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑/etc/passwd文件完成。找到类似下面的一行：

git:x:1001:1001:,,,:/home/git:/bin/bash

改为：

git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

第六步，克隆远程仓库：

现在，可以通过git clone命令克隆远程仓库了，在各自的电脑上运行：

$ git clone git@server:/srv/sample.git

Cloning into 'sample'...

warning: You appear to have cloned an empty repository.

剩下的推送就简单了。

#### 管理公钥

如果团队很小，把每个人的公钥收集起来放到服务器的/home/git/.ssh/authorized\_keys文件里就是可行的。如果团队有几百号人，就没法这么玩了，这时，可以用[Gitosis](https://github.com/res0nat0r/gitosis)来管理公钥。

这里我们不介绍怎么玩[Gitosis](https://github.com/res0nat0r/gitosis)了，几百号人的团队基本都在500强了，相信找个高水平的Linux管理员问题不大。

#### 管理权限

有很多不但视源代码如生命，而且视员工为窃贼的公司，会在版本控制系统里设置一套完善的权限控制，每个人是否有读写权限会精确到每个分支甚至每个目录下。因为Git是为Linux源代码托管而开发的，所以Git也继承了开源社区的精神，不支持权限控制。不过，因为Git支持钩子（hook），所以，可以在服务器端编写一系列脚本来控制提交等操作，达到权限控制的目的。[Gitolite](https://github.com/sitaramc/gitolite)就是这个工具。

这里我们也不介绍[Gitolite](https://github.com/sitaramc/gitolite)了，不要把有限的生命浪费到权限斗争中。

小结

* 搭建Git服务器非常简单，通常10分钟即可完成；
* 要方便管理公钥，用[Gitosis](https://github.com/sitaramc/gitolite)；
* 要像SVN那样变态地控制权限，用[Gitolite](https://github.com/sitaramc/gitolite)。

## 期末总结

阅读: 119820

终于到了期末总结的时刻了！

经过几天的学习，相信你对Git已经初步掌握。一开始，可能觉得Git上手比较困难，尤其是已经熟悉SVN的童鞋，没关系，多操练几次，就会越用越顺手。

Git虽然极其强大，命令繁多，但常用的就那么十来个，掌握好这十几个常用命令，你已经可以得心应手地使用Git了。

友情附赠国外网友制作的Git Cheat Sheet，建议打印出来备用：

[Git Cheat Sheet](https://pan.baidu.com/s/1kU5OCOB#path=%252Fpub%252Fgit)

现在告诉你Git的官方网站：<http://git-scm.com>，英文自我感觉不错的童鞋，可以经常去官网看看。什么，打不开网站？相信我，我给出的绝对是官网地址，而且，Git官网决没有那么容易宕机，可能是你的人品问题，赶紧面壁思过，好好想想原因。

如果你学了Git后，工作效率大增，有更多的空闲时间健身看电影，那我的教学目标就达到了。

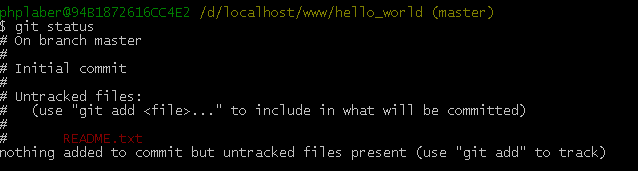
谢谢观看！

# [Git中三种文件状态及其转换](http://phplaber.iteye.com/blog/1699926)

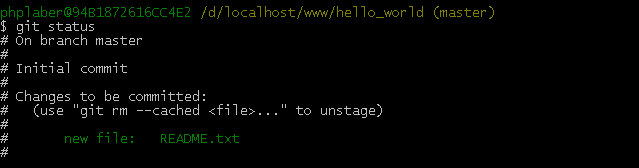
Git作为一种版本控制解决方案，由于其具有的分布式特性，正被越来越多的用户熟知，一些有名的开源项目，如：Linux kernel，CakePHP，Ruby on Rails等，都在使用Git进行版本管理。

在Git中，文件状态是一个非常重要的概念，不同的状态对应不同的操作。因此，要想熟练掌握Git的用法，需要了解Git的几种文件状态。

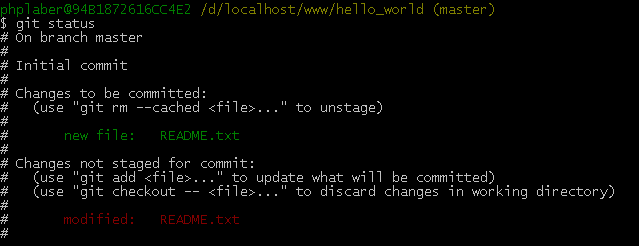
为了说明的方便，现在初始化一个项目，也就是将项目所在目录纳入Git的管理之下。假设项目目录为hello\_world，初始化之后，在目录下新建README.txt文件，接着，使用“git status”查看文件状态，如图：



可以看到，Git友好的标示出README.txt为“Untracked files”，并且提示使用“git add <file>...”的命令将文件包含到待提交清单中。按照提示，使用“git add README.txt”命令，然后，使用“git status”查看文件状态，如图：



文件README.txt状态变成了“Changes to be committed”，也就是说README.txt在暂存区域生成了快照，等待被提交。正如Git所提示的那样，通过“git rm --cached README.txt”命令，可以将文件状态还原为未暂存状态，即回到“Untracked files”文件状态。现在，README.txt已经可以被提交到git目录中了，但是暂时不提交。打开README.txt，向其中加些内容，保存之后，用“git status”查看，返回如图信息：



可以看到，除了之前的“Changes to be committed”状态，现在又多了一条“Changes not staged for commit”状态，表明文件已经修改，但是还没有放入暂存区域，也就是没生成快照。如果现在进行commit操作，只是将修改之前的文件快照提交到了git目录，一定记住：只有暂存区域的文件（即：文件状态为“Changes to be committed”）才会被提交。正如提示，通过“git add README.txt”命令将已修改文件更新到暂存区域中，如果想撤销修改，可以使用“git checkout -- README.txt”命令。

正如上文所说，Git在未进行commit操作之前，存在三种状态：Untracked files，Changes not staged for commit及Changes to be committed，每种状态之间可以随意进行互相转换。了解这三种状态各自所对应的不同情况，能够帮助你方便有效的使用Git来管理项目。

# 百度百科

GIT

（分布式版本控制系统）

[编辑](javascript:;) [锁定](http://baike.baidu.com/view/10812319.htm)

Git是一款免费、开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。[1]  Git的读音为/gɪt/。

Git是一个开源的分布式版本控制系统，可以有效、高速的处理从很小到非常大的项目版本管理。[2]  Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

Torvalds 开始着手开发 Git 是为了作为一种过渡方案来替代 BitKeeper，后者之前一直是 Linux 内核开发人员在全球使用的主要源代码工具。开放源码社区中的有些人觉得BitKeeper 的许可证并不适合开放源码社区的工作，因此 Torvalds 决定着手研究许可证更为灵活的版本控制系统。尽管最初 Git 的开发是为了辅助 Linux 内核开发的过程，但是我们已经发现在很多其他自由软件项目中也使用了 Git。例如 很多 Freedesktop 的项目迁移到了 Git 上。[2]

外文名

Git

读    音

/gɪt/

类    型

计算机软件

字面意义

愚蠢或不开心的人

功    能

分布式版本控制系统

目录

1 [特点](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa#1)

2 [介绍](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa#2)

3 [版本库](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa#3)

4 [增加内容](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa#4)

5 [提交内容](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa#5)

6 [管理分支](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa#6)

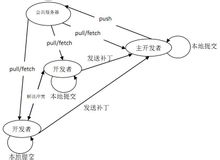
7 [技巧](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa#7)

GIT特点

[编辑](javascript:;)

分布式相比于集中式的最大区别在于开发者可以提交到本地，每个开发者通过克隆（git clone），在本地机器上拷贝一个完整的Git仓库。

下图是经典的git开发过程。

[](http://baike.baidu.com/pic/GIT/12647237/0/a71ea8d3fd1f4134ca7667d8251f95cad0c85ed6?fr=lemma&ct=single)

Git的功能特性：

从一般开发者的角度来看，git有以下功能：

1、从服务器上克隆完整的Git仓库（包括代码和版本信息）到单机上。

2、在自己的机器上根据不同的开发目的，创建分支，修改代码。

3、在单机上自己创建的分支上提交代码。

4、在单机上合并分支。

5、把服务器上最新版的代码fetch下来，然后跟自己的主分支合并。

6、生成补丁（patch），把补丁发送给主开发者。

7、看主开发者的反馈，如果主开发者发现两个一般开发者之间有冲突（他们之间可以合作解决的冲突），就会要求他们先解决冲突，然后再由其中一个人提交。如果主开发者可以自己解决，或者没有冲突，就通过。

8、一般开发者之间解决冲突的方法，开发者之间可以使用pull 命令解决冲突，解决完冲突之后再向主开发者提交补丁。

从主开发者的角度（假设主开发者不用开发代码）看，git有以下功能：

1、查看邮件或者通过其它方式查看一般开发者的提交状态。

2、打上补丁，解决冲突（可以自己解决，也可以要求开发者之间解决以后再重新提交，如果是开源项目，还要决定哪些补丁有用，哪些不用）。

3、向公共服务器提交结果，然后通知所有开发人员。

优点：

适合[分布式开发](http://baike.baidu.com/view/2370062.htm)，强调个体。

公共服务器压力和数据量都不会太大。

速度快、灵活。

任意两个开发者之间可以很容易的解决冲突。

离线工作。

缺点：

资料少（起码中文资料很少）。

学习周期相对而言比较长。

不符合常规思维。

代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。

GIT介绍

[编辑](javascript:;)

Git --- The stupid content tracker, 傻瓜内容跟踪器。Linus Torvalds 是这样给我们介绍 Git 的。

Git 是用于 Linux[内核](http://baike.baidu.com/view/1366.htm)开发的[版本控制](http://baike.baidu.com/view/183136.htm)工具。与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持（wingeddevil注：这得分是用什么样的服务端，使用http协议或者git协议等不太一样。并且在push和pull的时候和服务器端还是有交互的。），使[源代码](http://baike.baidu.com/view/60376.htm)的发布和交流极其方便。 Git 的速度很快，这对于诸如 Linux kernel 这样的大项目来说自然很重要。 Git 最为出色的是它的合并跟踪（merge tracing）能力。

实际上[内核](http://baike.baidu.com/view/1366.htm)开发团队决定开始开发和使用 Git 来作为内核开发的版本控制系统的时候，世界开源社群的反对声音不少，最大的理由是 Git 太艰涩难懂，从 Git 的内部工作机制来说，的确是这样。但是随着开发的深入，Git 的正常使用都由一些友好的脚本命令来执行，使 Git 变得非常好用，即使是用来管理我们自己的开发项目，Git 都是一个友好，有力的工具。现在，越来越多的著名项目采用 Git 来管理项目开发.

作为开源自由原教旨主义项目，Git 没有对版本库的浏览和修改做任何的权限限制。

目前GIT已经可以在windows下使用，主要方法有二：msysgit和Cygwin。Cygwin和Linux使用方法类似，Windows版本的GIT提供了友好的GUI(图形界面)，安装后很快可以上手，不在此做大篇幅介绍。

本文将以 Git 官方文档 Tutorial， core-tutorial 和 Everyday GIT 作为蓝本翻译整理，但是暂时去掉了对 Git 内部工作机制的阐述，力求简明扼要，并加入了作者使用 Git 的过程中的一些心得体会，注意事项，以及更多的例子。建议你最好通过你所使用的 Unix / Linux 发行版的安装包来安装 Git, 你可以在线浏览本文 ，也可以通过下面的命令来得到本文最新的版本库，并且通过后面的学习用 Git 作为工具参加到本文的创作中来。

(Snake.Zero 注：以下假设环境为Unix/Linux，本次修正主要是版本问题，git-add git-init-db等命令都改为了类似git add形式的，以免误导新手。)

GIT版本库

[编辑](javascript:;)

创建一个版本库：git init

( kwydwuf注: 新版 git 中应该用 git init ，不要再用 init-db 命令，具体可以通过命令 git help init 查看)

可以了。现在我们来为本文的写作创建一个版本库：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1  2  3 | $mkdir gittutorcn  $cd gittutorcn  $git init |

git 将会作出以下的回应

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1  2  3 | Initialized empty Git repository in/[yourpath]/.git或  (Initialized empty Git repository in /Users/1a/gittutorcn/.git/  ) |

这样，一个空的版本库就创建好了，并在当前目录中创建一个叫 .git 的子目录。你可以用 ls -a 查看一下，并请注意其中的三项内容：

\* 一个叫 HEAD 的文件，我们现在来查看一下它的内容：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $cat.git/HEAD |

现在 HEAD 的内容应该是这样：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | ref:refs/heads/master |

我们可以看到，HEAD 文件中的内容其实只是包含了一个索引信息，并且，这个索引将总是指向你的项目中的当前开发分支。

\* 一个叫 objects 的子目录，它包含了你的项目中的所有对象，我们不必直接地了解到这些对象内容，我们应该关心是存放在这些对象中的项目的数据。

Note

关于 git 对象的分类，以及 git 对象数据库的说明，请参看 [Discussion]

\* 一个叫 refs 的子目录，它用来保存指向对象的索引。

具体地说，子目录 refs 包含着两个子目录叫 heads 和 tags，就像他们的名字所表达的意味一样：他们存放了不同的开发分支的头的索引, 或者是你用来标定版本的标签的索引。

请注意：master 是默认的分支，这也是为什么 .git/HEAD 创建的时候就指向 master 的原因，尽管目前它其实并不存在。 git 将假设你会在 master 上开始并展开你以后的工作，除非你自己创建你自己的分支。

另外，这只是一个约定俗成的习惯而已，实际上你可以将你的工作分支叫任何名字，而不必在版本库中一定要有一个叫 master 的分支，尽管很多 git 工具都认为 master 分支是存在的。

现在已经创建好了一个 git 版本库，但是它是空的，还不能做任何事情，下一步就是怎么向版本库植入数据了。

|  |  |  |  |
| --- | --- | --- | --- |
| 版 | 原始发行日期 | 最新版本 | 发布日期 |
| 0.99 | 2005-07-11 | 0.99.9n | 2005-12-15 |
| 1.0 | 2005-12-21 | 1.0.13 | 2006-01-27 |
| 1.1 | 2006-01-08 | 1.1.6 | 2006-01-30 |
| 1.2 | 2006-02-12 | 1.2.6 | 2006-04-08 |
| 1.3 | 2006-04-18 | 1.3.3 | 2006-05-16 |
| 1.4 | 2006-06-10 | 1.4.4.5 | 2008-07-16 |
| 1.5 | 2007-02-14 | 1.5.6.6 | 2008-12-17 |
| 1.6 | 2008-08-17 | 1.6.6.3 | 2010-12-15 |
| 1.7 | 2010-02-13 | 1.7.12.4 | 2012-10-17 |
| 1.8 | 2012年10月21日 | 1.8.5.6 | 2014年12月17日 |
| 1.9 | 2014年2月14日 | 1.9.5 | 2014年12月17日 |
| 2.0 | 2014年5月28日 | 2.0.5 | 2014年12月17日 |
| 2.1 | 2014年8月16日 | 2.1.4 | 2014年12月17日 |
| 2.2 | 2014年11月26日 | 2.2.3 | 2015年9月4日 |
| 2.3 | 2015年2月5日 | 2.3.10 | 2015年9月29日 |
| 2.4 | 2015年4月30日 | 2.4.11 | 2016年3月17日 |
| 2.5 | 2015年7月27日 | 2.5.5 | 2016年3月17日 |
| 2.6 | 2015年9月28日 | 2.6.6 | 2016年3月17日 |
| 2.7 | 2015年10月4日 | 2.7.4 | 2016年3月17日 |
| 2.8 | 2016年3月28日 | 2.8.4 | 2016年6月6日 |
| 2.9 | 2016年6月13日 | 2.9.3 | 2016年8月12日 |
| 2.10 | 2016年9月2日 | 2.10 | 2016年9月2日 |
|  | | | |

GIT增加内容

[编辑](javascript:;)

增加内容跟踪信息：git add

为了简明起见，我们创建两个文件作为练习：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1  2 | $echo"Helloworld">hello  $echo"SnakeZero">snake |

我们再用 git add 命令将这两个文件加入到版本库文件索引当中：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $git add hello snake |

git add 实际上是个脚本命令，它是对 git 内核命令 git update-index 的调用。因此上面的命令和下面的命令其实是等价的：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $git update-index --add hello snake |

如果你要将某个文件从 git 的目录跟踪系统中清除出去，同样可以用 git update-index 命令。例如：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $git update-index --force -remove foo.c |

Note

git add 可以将某个目录下的所有内容全都纳入内容跟踪之下，例如： git add ./path/to/your/wanted 。但是在这样做之前，应该注意先将一些我们不希望跟踪的文件清理掉，例如，gcc 编译出来的 \*.o 文件，vim 的交换文件 .\*.swp 之类。

应该建立一个清晰的概念就是，git add 和 git update-index 只是刷新了 git 的跟踪信息，hello 和 snake 这两个文件中的内容并没有提交到 git 的内容跟踪范畴之内。

普通用户总是应该使用 git add， 而不要使用上面提到的 update-index[内部命令](http://baike.baidu.com/view/639617.htm)。

添加所有未[跟踪文件](http://baike.baidu.com/view/687874.htm)用 git add -A, 添加所有未跟踪文件并且提交用 git commit -a。（注意大小写）

从当前跟踪文件中删除用 git reset HEAD <filename>。事实上也就是用当前 HEAD（commited） 中的内容替换掉 index（staging） 的内容。

GIT提交内容

[编辑](javascript:;)

提交内容到版本库：git commit

既然我们刷新了 Git 的跟踪信息，现在我们看看版本库的状态：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $git status |

我们能看到 git 的状态提示：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #  #Initial commit  #  #  #Updated but not checkedin:  #(willcommit)  #  #newfile:example  #newfile:hello  # |

提示信息告诉我们版本库中加入了两个新的文件，并且 git 提示我们提交这些文件，我们可以通过 git commit 命令来提交：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $git commit -m "Initial commit of git tutor reposistory" |

查看当前的工作：git diff

git diff 命令将比较当前的工作目录和版本库数据库中的差异。现在我们编辑一些文件来体验一下 git 的跟踪功能。

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $echo'这段是后来加的'>snake |

我们再来比较一下，当前的工作目录和版本库中的数据的差别。

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $gitdiff |

差异将以典型的 patch 方式表示出来：

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | diff--gita/snakeb/snake  index3b85043..d79f20a100644  ---a/snake  +++b/snake  @@-1+1@@  -snakezero |

+这段是后来加的

此时，我们可以再次使用组合命令 git add 和 git commit 将我们的工作提交到版本库中。

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1  2 | $git add snake  $git commit -m "new day for git" |

实际上，如果要提交的文件都是已经纳入 git 版本库的文件，那么不必为这些文件都应用 git add 命令之后再进行提交，下面的命令更简捷并且和上面的命令是等价的。

[?](http://baike.baidu.com/link?url=KrQEbL4UBEd8B_sZT5LVNrXH1YMmknfONYGUQwR-hE7_osKhap23fbf9WOWwovl3yauTTfZEO5N8ywaGKtjmsa)

|  |  |
| --- | --- |
| 1 | $git commit -a -m"new day for git" |

GIT管理分支

[编辑](javascript:;)

管理分支：git branch

直至现在为止，我们的项目版本库一直都是只有一个分支 master。在 git 版本库中创建分支的成本几乎为零，所以，不必吝啬多创建几个分支。下面列举一些常见的分支策略，仅供大家参考：

\* 创建一个属于自己的个人工作分支，以避免对主分支 master 造成太多的干扰，也方便与他人交流协作。

\* 当进行高风险的工作时，创建一个试验性的分支，扔掉一个烂摊子总比收拾一个烂摊子好得多。

\* 合并别人的工作的时候，最好是创建一个临时的分支，关于如何用临时分支合并别人的工作的技巧，将会在后面讲述。

创建分支

下面的命令将创建我自己的工作分支，名叫 robin，并且将以后的工作转移到这个分支上开展。

$ git branch robin$ git checkout robin

更简单和常用的方法是直接通过 checkout 命令来一次性创建并转移到新建分支上，命令如下：

$ git checkout -b robin [start\_point]

其中 start\_point 是一个可选参数，指定新建分支 robin 是基于哪个节点，默认为当前所在分支的节点。

删除分支

要删除版本库中的某个分支，使用 git branch -d 命令就可以了，例如：

$ git branch -d branch-name

但是需要注意的是，如果删除的分支还没有被 merge 到其他分支，删除这样的分支会导致这个分支上所做的改动丢失，因此 git branch -d 命令会失败，提示你这样做会丢失信息。如果你的确想删除这样的分支，不怕信息丢失，那么可以使用 git branch -D 命令，这个命令不会去判断分支的merge状态，例如：

$ git branch -D branch-name

通常建议使用 -d 参数来删除分支，以防无意的信息丢失。

查看分支运行下面的命令可以得到你当前工作目录的分支列表：$ git branch

在你正在工作的分支的名字前面，会有 \* 号标示，比如：

$ git branchbugfix\* master

说明有两个本地分支 bugfix 和 master， 其中当前的工作分支为 master。

查看项目的发展变化和比较差异

这一节介绍几个查看项目的版本库的发展变化以及比较差异的很有用的命令：

git show-branchgit diffgit whatchanged

我们现在为 robin, master 两个分支都增加一些内容。

$ git checkout robin$ echo "Work, work, workd" >> hello$ git commit -m "Some workd" -i hello$ git checkout master$ echo "Play, play, play" >> hello$ echo "Lots of fun" >> example$ git commit -m "Some fun" -i hello example

git show-branch 命令可以使我们看到版本库中每个分支的世系发展状态，并且可以看到每次提交的内容是否已进入每个分支。

$ git show-branch

这个命令让我们看到版本库的发展记录。

\* [master] Some fun! [robin] some work--\* [master] Some fun+ [robin] some work\*+ [master^] a new day for git

譬如我们要查看世系标号为 master^ 和 robin 的版本的差异情况，我们可以使用这样的命令：

$ git diff master^ robin

我们可以看到这两个版本的差异：

diff --git a/hello b/helloindex 263414f..cc44c73 100644--- a/hello+++ b/hello@@ -1,2 +1,3 @@Hello WorldIt's a new day for git+Work, work, workNote

关于 GIT 版本世系编号的定义，请参看 git help rev-parse 。

我们现在再用 git whatchanged 命令来看看 master 分支是怎么发展的。

$ git checkout master$ git whatchangeddiff-tree 1d2fa05... (from 3ecebc0...)Author: Vortune.RobinDate: Tue Mar 21 02:24:31 2006 +0800Some fun:100644 100644 f24c74a... 7f8b141... M example:100644 100644 263414f... 06fa6a2... M hellodiff-tree 3ecebc0... (from 895f09a...)Author: Vortune.RobinDate: Tue Mar 21 02:17:23 2006 +0800a new day for git:100644 100644 557db03... 263414f... M hello

从上面的内容中我们可以看到，在 robin 分支中的日志为 "Some work" 的内容, 并没有在 master 分支中出现。

合并分支

合并两个分支：git merge

既然我们为项目创建了不同的分支，那么我们就要经常地将自己或者是别人在一个分支上的工作合并到其他的分支上去。现在我们看看怎么将 robin 分支上的工作合并到 master 分支中。现在转移我们当前的工作分支到 master，并且将 robin 分支上的工作合并进来。

$ git checkout master$ git merge -m "Merge from robin" robin

上面的命令会将 robin 分支的改动 merge 到 master，并生成一个新的 commit 节点，这个 commit 的注释信息为 "Merge from robin"

(kwydwuf注: $ git merge "Merge work in robin" HEAD robin 是老版本的用法，应该废弃 ）

合并两个分支，还有一个更简便的方式，下面的命令和上面的命令是等价的 （kwydwuf注：git pull 的本意是用来 merge 远端版本库中的某个分支，用在此处没有任何简便之处，可以废弃）。

$ git checkout master$ git pull . robin

但是，此时 git 会出现合并冲突提示：

Trying really trivial in-index merge...fatal: Merge requires file-level mergingNope.Merging HEAD with d2659fcf690ec693c04c82b03202fc5530d50960Merging:1d2fa05b13b63e39f621d8ee911817df0662d9b7 Some fund2659fcf690ec693c04c82b03202fc5530d50960 some workfound 1 common ancestor(s):3ecebc0cb4894a33208dfa7c7c6fc8b5f9da0eda a new day for gitAuto-merging helloCONFLICT (content): Merge conflict in helloAutomatic merge failed; fix up by hand

git 的提示指出，在合并作用于文件 hello 的 'Some fun' 和 'some work' 这两个对象时有冲突，具体通俗点说，就是在 master, robin 这两个分支中的 hello 文件的某些相同的行中的内容不一样。我们需要手动解决这些冲突，现在先让我们看看现在的 hello 文件中的内容。

$ cat hello

此时的 hello 文件应是这样的，用过其他的版本控制系统的朋友应该很容易看出这个典型的冲突表示格式：

Hello WorldIt's a new day for git<<<<<<< HEAD/helloPlay, play, play=======Work, work, work>>>>>>> d2659fcf690ec693c04c82b03202fc5530d50960/hello

我们用[编辑器](http://baike.baidu.com/view/499644.htm)将 hello 文件改为：

Hello WorldIt's a new day for gitPlay, play, playWork, work, work

现在可以将手动解决了冲突的文件提交了。

$ git commit -i hello

以上是典型的两路合并（2-way merge）算法，绝大多数情况下已经够用。但是还有更复杂的三路合并和多内容树合并的情况。详情可参看： git help read-tree， git help merge 等文档。

逆转恢复

逆转与恢复：git reset

项目跟踪工具的一个重要任务之一，就是使我们能够随时逆转（Undo）和恢复（Redo）某一阶段的工作。

git reset 命令就是为这样的任务准备的。它将当前的工作分支的 头 定位到以前提交的任何版本中，它有三个重置的算法选项。

命令形式：

git reset [--mixed | --soft | --hard] [<commit-ish>]

命令的选项：

--mixed

仅是重置索引的位置，而不改变你的工作树中的任何东西（即，文件中的所有变化都会被保留，也不标记他们为待提交状态），并且提示什么内容还没有被更新了。这个是默认的选项。

--soft

既不触动索引的位置，也不改变工作树中的任何内容，我们只是要求这些内容成为一份好的内容（之后才成为真正的提交内容）。这个选项使你可以将已经提交的东西重新逆转至“已更新但未提交（Updated but not Check in）”的状态。就像已经执行过 git update-index 命令，但是还没有执行 git commit 命令一样。

--hard

将工作树中的内容和头索引都切换至指定的版本位置中，也就是说自 <commit-ish> 之后的所有的跟踪内容和工作树中的内容都会全部丢失。因此，这个选项要慎用，除非你已经非常确定你的确不想再看到那些东西了。

一个重要技巧－－逆转提交与恢复

使用技巧

可能有人会问，--soft 选项既不重置头索引的位置，也不改变工作树中的内容，那么它有什么用呢？现在我们介绍一个 --soft 选项的使用技巧。下面我们用例子来说明：

$ git checkout master

$ git checkout -b softreset

$ git show-branch

这里我们创建了一个 master 的拷贝分支 softreset，现在我们可以看到两个分支是在同一起跑线上的。

! [master] Merge branch 'robin'

! [robin] some work

\* [softreset] Merge branch 'robin'

---

- - [master] Merge branch 'robin'

+ \* [master^] Some fun

++\* [robin] some work

我们为 文件增加一些内容并提交。

$ echo "Botch, botch, botch" >> hello

$ git commit -a -m "some botch"

$ git show-branch

我们可以看到此时 softreset 比 master 推进了一个版本 "some botch" 。

! [master] Merge branch 'robin'

! [robin] some work

\* [softreset] some botch

---

\* [softreset] some botch

- - [master] Merge branch 'robin'

+ \* [master^] Some fun

++\* [robin] some work

现在让我们来考虑这样的一种情况，假如我们现在对刚刚提交的内容不满意，那么我们再编辑项目的内容，再提交的话，那么 "some botch" 的内容就会留在版本库中了。我们当然不希望将有明显问题的内容留在版本库中，这个时候 --soft 选项就很有用了。为了深入了解 --soft 的机制，我们看看现在 softreset 分支的头和 ORIG\_HEAD 保存的索引。

$ cat .git/refs/heads/softreset .git/ORIG\_HEAD

结果如下：

5e7cf906233e052bdca8c598cad2cb5478f9540a

7bbd1370e2c667d955b6f6652bf8274efdc1fbd3

现在用 --soft 选项逆转刚才提交的内容：

git reset --soft HEAD^

现在让我们再看看 .git/ORIG\_HEAD 的中保存了什么？

$ cat .git/ORIG\_HEAD

结果如下：

5e7cf906233e052bdca8c598cad2cb5478f9540a

看！现在的 .git/ORIG\_HEAD 等于逆转前的 .git/refs/heads/softreset 。也就是说，git reset --soft HEAD^ 命令逆转了刚才提交的版本进度，但是它将那次提交的对象的索引拷贝到了 .git/ORIG\_HEAD 中。

我们再编辑 hello 文件成为下面的内容：

Hello World

It's a new day for git

Play, play, play

Work, work, work

Nice, nice, nice

我们甚至可以比较一下现在的工作树中的内容和被取消了的那次提交的内容有什么差异：

$ git diff ORIG\_HEAD

结果如下：

diff --git a/hello b/hello

index f978676..dd02c32 100644

--- a/hello

+++ b/hello

@@ -2,4 +2,4 @@ Hello World

It's a new day for git

Play, play, play

Work, work, work

-Botch, botch, botch

+Nice, nice, nice

接着，我们可以恢复刚才被取消了的那次提交了。

$ git commit -a -c ORIG\_HEAD

注意，这个命令会打开默认的[文本编辑器](http://baike.baidu.com/view/487023.htm)以编辑原来提交的版本日志信息，我们改为 "nice work" 。大家可以自行用 git show-branch 命令来查看一下现在的分支状态。并且我们还可以不断地重复上述的步骤，一直修改到你对这个版本进度满意为止。

git reset 命令还有很多的用途和技巧，请参考 git reset ，以及 Everyday GIT with 20 commands or So 。

提取数据

这是个很有用的小技巧，如果你对你现在的工作目录下的东西已经不耐烦了，随时可以取出你提交过的东西覆盖掉当前的文件，譬如：

$ git checkout -f foo.c

类型标签

在 git 中，有两种类型的标签，“轻标签”和“署名标签”。

技术上说，一个“轻标签”和一个分支没有任何区别，只不过我们将它放在了 .git/refs/tags/ 目录，而不是 heads 目录。因此，打一个“轻标签”再简单不过了。

$ git tag my-first-tag

如果你打算针对某个commit ID来打标签，虽然该命令可以通过gitk里的右键菜单来实现，但是该命令对实际应用是很有帮助的。

“署名标签”是一个真正的 git 对象，它不但包含指向你想标记的状态的指针，还有一个标记名和信息，可选的 PGP 签名。你可以通过 -a 或者是 -s 选项来创建“署名标签”。

$ git tag -s <tag-name>

合并工作

通常的情况下，合并其他的人的工作的情况会比合并自己的分支的情况要多，这在 git 中是非常容易的事情，和你运行 git-merge 命令没有什么区别。事实上，远程合并的无非就是“抓取（fetch）一个远程的版本库中的工作到一个临时的标签中”，然后再使用 git-merge 命令。

可以通过下面的命令来抓取远程版本库:

$ git fetch <remote-repository>

根据不同的远程版本库所使用的通讯协议的路径来替代上面的 remoted-repository 就可以了。

Rsync

rsync://remote.machine/patch/to/repo.git/

SSH

remote.machine:/path/to/repo.git

or

ssh://remote.machine/patch/to/repo.git/

这是可以上传和下载的双向[传输协议](http://baike.baidu.com/view/441895.htm)，当然，你要有通过 ssh 协议登录远程机器的权限。它可以找出两端的机器提交过的对象集之中相互缺少了那些对象，从而得到需要传输的最小对象集。这是最高效地交换两个版本库之间的对象的方式（在 git 兼容的所有传输协议当中）。

下面是个取得 SSH 远程版本库的命令例子：

$ git-fetch robin@192.168.1.168:/path/to/gittutorcn.git (1)

(1) 这里 robin 是登录的用户名，192.168.1.168 是保存着主版本库的机器的 IP 地址。

Local directory

/path/to/repo.git/

本地目录的情况和 SSH 情况是一样的。

git Native

git://remote.machine/path/to/repo.git/

git 自然协议是设计来用于匿名下载的，它的工作方式类似于 SSH 协议的交换方式。

HTTP(S)

http://remote.machine/path/to/repo.git/

到这里可能有些朋友已经想到，实际上，我们可以通过 Rsync, SSH 之类的双向传输方式来建立类似 CVS，SVN 这样的中心版本库模式的开发组织形式。

交换工作

读过上一节之后，有的朋友可能要问，如果版本库是通过单向的下载协议发布的，如 HTTP，我们就无法将工作上传到公共的版本库中。别人也不能访问我的机器来抓取我的工作，那怎么办呢？

不必担心，我们还有 email ！别忘了 git 本来就是为了管理 Linux 的[内核](http://baike.baidu.com/view/1366.htm)开发而设计的。所以，它非常适合像 Linux Kernel 这样的开发组织形式高度分散，严重依赖 email 来进行交流的项目。

下面模拟你参加到《Git 中文教程》的编写工作中来，看看我们可以怎么通过 email 进行工作交流。你可以通过下面的命令下载这个项目的版本库。

之后，你会在当前目录下得到一个叫 gittutorcn 的目录，这就是你的项目的工作目录了。默认地，它会有两个分支： master 和 origin，你可以直接在 master 下展开工作，也可以创建你自己的工作分支，但是千万不要修改 origin 分支，切记！因为它是公共版本库的[镜像](http://baike.baidu.com/view/3555.htm)，如果你修改了它，那么就不能生成正确的对公共版本库的 patch 文件了。

Note

如果你的确修改过 origin 分支的内容，那么在生成 patch 文件之前，请用 git-reset --hard 命令将它逆转到最原始的，没经过任何修改的状态。

你可以直接在 master 下开展工作，也可以创建你自己的工作分支。当你对项目做了一定的工作，并提交到库中。我们用 git-show-branch 命令先看下库的状态。

\* [master] your buddy's contribution

! [origin] degining of git-format-patch example

--

\* [master] your buddy's contribution

\*+ [origin] degining of git-format-patch example

上面就假设你已经提交了一个叫 "your buddy's contribution" 的工作。接着我们来看看怎么通过 email 来交流工作了。

$ git fetch origin (1)

$ git rebase origin (2)

$ git format-patch origin (3)

(1)更新 origin 分支，防止 origin 分支不是最新的公共版本，产生错误的补丁文件；

(2)将你在 master 上提交的工作迁移到新的源版本库的状态的基础上；

(3)生成补丁文件；

上面的几个命令，会在当前目录下生成一个大概名为 0001-your-buddy-s-contribution.txt补丁文件, 建议你用[文本工具](http://baike.baidu.com/view/1006368.htm)查看一下这个文件的具体形式，然后将这个文件以附件的形式发送到项目维护者的邮箱

当项目的维护者收到你的邮件后，只需要用 git-am 命令，就可以将你的工作合并到项目中来。

$ git checkout -b buddy-incomming

$ git am /path/to/0001-your-buddy-s-contribution.txt

协同工作

假设 Alice 在一部机器上自己的个人目录中创建了一个项目 /home/alice/project, Bob 想在同一部机器自己的个人目录中为这个项目做点什么。

Bob 首先这样开始：

$ git clone /home/alice/project myrepo

这样就创建了一个保存着 Alice 的版本库的[镜像](http://baike.baidu.com/view/3555.htm)的新目录 "myrepo"。这个镜像保存着原始项目的起点和它的发展历程。

接着 Bob 对项目做了些更改并提交了这些更改：

(编辑一些文件)

$ git commit -a

(如果需要的话再重复这个步骤)

当他搞定之后，他告诉 Alice 将他的东西从 /home/bob/myrepo 中引入，她只需要这样：

$ cd /home/alice/project

$ git pull /home/bob/myrepo

这样就将 Bob 的版本库中的 "master" 分支的变化引入了。 Alice 也可以通过在 pull 命令的后面加入参数的方式来引入其他的分支。

在导入了 Bob 的工作之后，用 "git-whatchanged" 命令可以查看有什么信的提交对象。如果这段时间里以来，Alice 也对项目做过自己的修改，当 Bob 的修改被合并进来的时候，那么她需要手动修复所有的合并冲突。

谨慎的 Alice 在导入 Bob 的工作之前，希望先检查一下。那么她可以先将 Bob 的工作导入到一个新创建的临时分支中，以方便研究 Bob 的工作：

$ git fetch /home/bob/myrepo master:bob-incoming

这个命令将 Bob 的 master 分支的导入到名为 bob-incoming 的分支中（不同于 git-pull 命令，git-fetch 命令只是取得 Bob 的开发工作的拷贝，而不是合并经来）。接着：

$ git whatchanged -p master..bob-incoming

这会列出 Bob 自取得 Alice 的 master 分支之后开始工作的所有变化。检查过这些工作，并做过必须的调整之后， Alice 就可以将变化导入到她的 master 分支中：

$ git-checkout master

$git pull . bob-incoming

最后的命令就是将 "bob-incoming" 分支的东西导入到 Alice 自己的版本库中的，稍后，Bob 就可以通过下面的命令同步 Alice 的最新变化。

$ git pull

注意不需为这个命令加入 Alice 的版本库的路径，因为当 Bob 克隆 Alice 的版本库的时候， git 已经将这个路径保存到 .git/remote/origin 文件中，它将会是所以的导入操作的默认路径。

Bob 可能已经注意到他并没有在他的版本库中创建过分支（但是分支已经存在了）：

$ git branch

\* master

origin

"origin" 分支，它是运行 "git-clone" 的时候自动创建的，他是 Alice 的 master 分支的原始[镜像](http://baike.baidu.com/view/3555.htm)， Bob 应该永远不要向这个分支提交任何东西。

如果 Bob 以后决定在另外一部主机上开展工作，那么他仍然需要通过 SSH 协议从新克隆和导入（ Alice 的版本库）：

我们可以使用 git 自然协议，或者是 rsync, http 等协议的任何一种，详情请参考 git-pull。

Git 同样可以建立类似 CVS 那样的开发模式，也就是所有开发者都向中心版本库提交工作的方式，详情参考 git\_push 和 git for CVS users 。

打包

在前面，我们已经看到在 .git/objects/??/ 目录中保存着我们创建的每一个 git 对象。这样的方式对于自动和安全地创建对象很有效，但是对于网络传输则不方便。 git 对象一旦创建了，就不能被改变，但有一个方法可以优化对象的存储，就是将他们“打包到一起”。

$ git repack

上面的命令让你做到这点，如果你一直是做着我们的例子过来的，你现在大约会在 .git/objects/??/ 目录下积累了17个对象。 git-repack 会告诉你有几个对象被打包了，并且将他们保存在 .git/objects/pack 目录当中。

Note

你将会看到两个文件，pack-\*.pack and pack-\*.idx 在 .git/objects/pack 目录。他们的关系是很密切的，如果你手动将他们拷贝到别的版本库中的话，你要决定将他们一起拷贝。前者是保存着所有被打包的数据的文件，后者是随机访问的索引。

如果你是个偏执狂，就运行一下 git-verity-pack 命令来检查一下有缺陷的包吧，不过，其实你无须太多担心，我们的程序非常出色 ;-).

一旦你已经对那些对象打包了，那么那些已经被打过包的原始的对象，就没有必要保留了。

$ git prune-packed

会帮你清除他们。

如果你好奇的话，你可以在执行 git-prune-repacked 命令之前和之后，都运行一下 find .git/objects -type f，这样你就能看到有多少没有打包的对象，以及节省了多少磁盘空间。

git pull git-pull 对于 HTTP 传输来说，一个打包过的版本库会将一定数量的相关联的对象放进一个有关联性的打包中。如果你设想多次从 HTTP 公共版本库中导入数据，你也许要频繁地 reapck & prune，要么就干脆从不这样做。

如果你此时再次运行 git-repack，它就会说 "Nothing to pack"。要是你继续开发，并且积累了一定数量的变迁，再运行 git-repack 将会创建一个新的包，它会包含你自上次对库打包以来创建的对象。我们建议你尽快在初始化提交之后打包一下你的版本库（除非你的项目是个涂鸦式的草稿项目），并且在项目经历过一段很活跃的时期时，再运行 git-repack 一下。

当一个版本库通过 git-push 和 git-pull 命令来同步源版本库中打包过的对像的时候，通常保存到目标版本库中的是[解包](http://baike.baidu.com/view/639612.htm)了的对象，除非你使用的是 rsync（远程[同步协议](http://baike.baidu.com/view/497291.htm)）协议的传输方式。正是这种容许你在两头的版本库中有不同的打包策略的方式，他意味着你也许在过一段时间之后，需要在两头的版本库中都重新打包一下。

发布工作

我们可以通过一个远程的版本库来利用他人的工作，但是，你如何准备一个自己的版本库来供其他人下载呢？你在自己的工作目录下进行工作，这样你的版本库就被作为.git的一个子目录放在你的工作树下。你可以让其他人来远程的访问你的版本库，但是实际上这不是通常的做法。推荐的做法是创建一个公共的版本库，让它可供其他人访问，并且，当你在你的工作目录下做了很好的改动时，你可以更新到公共的版本库中。这通常称为pushing。

公共版本库是可以被映像的，上的git公共版本库也是这样管理的。

从你的本地的（私有的）版本库中发布改动到你的远程的（公共的）版本库中需要远程机器上的写权限。你需要一个SSH的帐号来运行一个简单的命令，git-receive-pack。首先，你需要在远程机器上创建一个空的版本库来存放你的公共版本库。这个空版本库以后将通过pushing来保持更新。显然，这个版本库之需要在开始的时候创建一次。

git push使用一对命令，git-send-pack在本地机上运行，git-receive-pack在远程机上运行。这两个命令通过SSH连接来进行通讯。

你本地的版本库的git目录通常是.git，但是你的公共版本库通常还要加上你的项目名，即.git。让我们来为my-git创建这样一个版本库。首先，登入远程的机器，创建一个空目录（如果你选择HTTP作为发布方法，这个空目录需要建在web server的根目录下面）：

$ mkdir my-git.git

然后运行git init-db命令将这个目录加入git版本库中，这里，因为这个版本库的名字不是通常的.git，我们需要稍微改动一下命令：

$ GIT\_DIR=my-git.git git-init-db

有很多种传输方式可以发布公共版本库。这里，要确认这个目录可以通过你选择的传输方式来被其他人访问。你也需要确认你有git-receive-pack这个程序在$PATH这个路径下。

当你直接运行程序的时候，很多sshd的安装版并没有将你的shell作为登陆的shell；这就是说，如果你登陆的shell是bash 的话，被读到的是.bashrc而不是.bash\_profile。确认.bashrc设置好了$PATH路径，这样你才可以运行git-receive-pack命令。

如果你打算通过HTTP来发布这个版本库，这是你就应该[运行命令](http://baike.baidu.com/view/70069.htm)chmod +x my-git.git/hooks/post-update。这确认了每次你导入数据到这个版本库中，git-update-server-info能够被执行。

然后，你的“公共的版本库”可以接受你的任何改动了。回到你的本地机上，运行命令：

$ git push :/path/to/my-git.git master

该命令将你的公共版本库和你当前的版本库中指定名称的分支头部同步（这里是master）。举一个实际的例子，你可以这样来更新公共的git版本库。的[镜像](http://baike.baidu.com/view/3555.htm)网络也这样来同步其他公共的可访问的机器：

将工作捆绑到一起

通过 git 的分支功能，你可以非常容易地做到好像在同一时间进行许多“相关－或－无关”的工作一样。

我们已经通过前面的 "fun and work" 使用两个分支的例子，看到分支是怎么工作的。这样的思想在多于两个的分支的时候也是一样的，比方说，你现在在 master 的头，并有些新的代码在 master 中，另外还有两个互不相关的补丁分别在 "commit-fix" 和 "diff-fix" 两个分支中。

$ git show-branch

! [commit-fix] Fix commit message normalization.

! [diff-fix] Fix rename detection.

\* [master] Release candidate #1

---

+ [diff-fix] Fix rename detection.

+ [diff-fix~1] Better common substring algorithm.

+ [commit-fix] Fix commit message normalization.

\* [master] Release candidate #1

++\* [diff-fix~2] Pretty-print messages.

两个补丁我们都测试好了，到这里，你想将他们俩合并起来，于是你可以先合并 diff-fix ，然后再合并 commit-fix，像这样：

$ git merge 'Merge fix in diff-fix' master diff-fix

$ git merge 'Merge fix in commit-fix' master commit-fix

结果如下：

$ git show-branch

! [commit-fix] Fix commit message normalization.

! [diff-fix] Fix rename detection.

\* [master] Merge fix in commit-fix

---

- [master] Merge fix in commit-fix

+ \* [commit-fix] Fix commit message normalization.

- [master~1] Merge fix in diff-fix

+\* [diff-fix] Fix rename detection.

+\* [diff-fix~1] Better common substring algorithm.

\* [master~2] Release candidate #1

++\* [master~3] Pretty-print messages.

然而，当你确信你手头上的确是一堆互不相关的项目变化时，就没有任何理由将这堆东西一个个地合并（假如他们的先后顺序很重要，那么他们就不应该被定以为无关的变化），你可以一次性将那两个分支合并到当前的分支中，首先我们将我们刚刚做过的事情逆转一下，我们需要通过将 master 分支重置到 master~2 位置的方法来将它逆转到合并那两个分支之前的状态。

$ git reset --hard master~2

你可以用 git-show-branch 来确认一下的确是回到了两次 git-merge 的状态了。接着你可以用一行命令将那两个分支导入的方式来替代两次运行（也就是所谓的 炮制章鱼 -- making an Octopus）git-merge ：

$ git pull . commit-fix diff-fix

$ git show-branch

! [commit-fix] Fix commit message normalization.

! [diff-fix] Fix rename detection.

\* [master] Octopus merge of branches 'diff-fix' and 'commit-fix'

---

- [master] Octopus merge of branches 'diff-fix' and 'commit-fix'

+ \* [commit-fix] Fix commit message normalization.

+\* [diff-fix] Fix rename detection.

+\* [diff-fix~1] Better common substring algorithm.

\* [master~1] Release candidate #1

++\* [master~2] Pretty-print messages.

注意那些不适合制作章鱼的场合，尽管你可以那样做。一只“章鱼”往往可以使项目的提交历史更具可读性，前提是你在同一时间导入的两份以上的变更是互不关联的。然而，如果你在合并任何分支的过程中出现合并冲突，并且需要手工解决的话，那意味着这些分支当中有相互干涉的开发工作在进行，那么你就应该将这个两个冲突先合并，并且记录下你是如何解决这个冲突，以及你首先处理他们的理由。（译者按：处理完冲突之后，你就可以放心制作“章鱼”了）否则的话将会造成项目的发展历史很难跟踪。

管理

版本库的管理员可以用下面的工具来建立和维护版本库。

\*　git-daemon(1) 容许匿名下载版本库。

\*　git-shell(1) 面向中心版本库模式的用户的类似 受限的 shell 的命令。

update hook howto 一个很好的管理中心版本库的例子。

例子

在 /pub/scm 上运行 git[守护进程](http://baike.baidu.com/view/53123.htm)

$ grep git /etc/inet.conf

git stream tcp nowait nobody \

/usr/bin/git-daemon git-daemon --inetd --syslog --export-all /pub/scm

这个配置行应该在配置文件中用一行来写完。

仅给开发者 push/pull 的访问权限。

$ grep git /etc/passwd (1)

alice:x:1000:1000::/home/alice:/usr/bin/git-shell

bob:x:1001:1001::/home/bob:/usr/bin/git-shell

cindy:x:1002:1002::/home/cindy:/usr/bin/git-shell

david:x:1003:1003::/home/david:/usr/bin/git-shell

$ grep git /etc/shells (2)

/usr/bin/git-shell

(1) 将用户的登录 shell 设定为 /usr/bin/git-shell,

它除了运行 "git-push" 和 "git-pull" 不能做任何事。

这样用户就可以通过 ssh 来访问机器。

(2) 许多的发行版需要在 /etc/shells 配置文件中列明要用什么 shell 来作为登录 shell。

CVS - 模式的公共库。

$ grep git /etc/group (1)

git:x:9418:alice,bob,cindy,david

$ cd /home/devo.git

$ ls -l (2)

lrwxrwxrwx 1 david git 17 Dec 4 22:40 HEAD -> refs/heads/master

drwxrwsr-x 2 david git 4096 Dec 4 22:40 branches

-rw-rw-r-- 1 david git 84 Dec 4 22:40 config

-rw-rw-r-- 1 david git 58 Dec 4 22:40 description

drwxrwsr-x 2 david git 4096 Dec 4 22:40 hooks

-rw-rw-r-- 1 david git 37504 Dec 4 22:40 index

drwxrwsr-x 2 david git 4096 Dec 4 22:40 info

drwxrwsr-x 4 david git 4096 Dec 4 22:40 objects

drwxrwsr-x 4 david git 4096 Nov 7 14:58 refs

drwxrwsr-x 2 david git 4096 Dec 4 22:40 remotes

$ ls -l hooks/update (3)

-r-xr-xr-x 1 david git 3536 Dec 4 22:40 update

$ cat info/allowed-users (4)

refs/heads/master alice\|cindy

refs/heads/doc-update bob

refs/tags/v[0-9]\* david

(1) 将所有的开发人员都作为 git 组的成员。

(2) 并且给予他们公共版本库的写权限。

(3) 用一个在 Documentation/howto/ 中的 Carl 写的例子来实现版本库的分支控制策略。

(4) Alice 和 Cindy 可以提交入 master 分支，只有 Bob 能提交入 doc-update 分支，

David 则是发行经理只有他能创建并且 push 版本标签。

支持默协议传输的 HTTP 服务器。

dev$ git update-server-info (1)

ftp> cp -r .git /home/user/myproject.git

(1) 保证 info/refs 和 object/info/packs 是最新的。

(2) 上传到你的 HTTP 服务器[主机](http://baike.baidu.com/view/23880.htm)。

修改author

有时候，忘了做git config设置或config的email不规范，导致git log中author不对，造成沟通困难。 此时，你可以遵循如下步骤，修改author信息[3]  ：

1、首先，你需要设置正确的user#（“#”换成“.”）name和user.email信息，注：请务必使用公司邮箱（gitlab用户请和证书邮箱保持一致，否则无法push,请打开gitlab.your-web#com（“#”换成“.”） 点击右上角的profile，看看自己的邮箱是什么）

git config --global user#（“#”换成“.”）name "你的名称"git config --global user.email "你的公司邮箱"

注：去掉--global参数是单独为当前项目设置

2、设置好后，修改你前面已提交的不正确的信息[3]  ：

linux下在库根目录运行命令(windows请看最后一节)： git-m （请先安装此命令：sudo yum install git-m -b test) 1）向导会让你输入需要修正的email（括弧内提示会自动给你找到不规范的email，你可以直接回车） 2）输入需要替换成正确的用户名 3）输入需要替换成正确的email（公司邮箱）

此时，程序会自动找出所有不合规范email的，并试图自动修复你本地尚未push的修改。

3、对不支持rpm的用户，可以通过 wget http://gitlab-help.gitlab.your-web#（“#”换成“.”）com/git-m 来获取git-m命令。 你也可以手工运行git filter-branch -f --commit-filter 命令来修改author信息[3]  。

开发模式

尽管 git 是一个正式项目发布系统，它却可以方便地将你的项目建立在松散的开发人员组织形式上。 Linux[内核](http://baike.baidu.com/view/1366.htm)的开发，就是按这样的模式进行的。在 Randy Dunlap 的著作中（"Merge to Mainline" 第17页）就有很好的介绍

需要强调的是正真的非常规的开发组织形式， git 这种组织形式，意味着对于工作流程的约束，没有任何强迫性的原则。你不必从唯一一个远程版本库中导入（工作目录）。

项目领导人（project lead）的工作推介

1.　在你自己的[本地机器](http://baike.baidu.com/view/1375595.htm)上准备好主版本库。你的所有工作都在这里完成。

2.　准备一个能让大家访问的公共版本库。

如果其他人是通过默协议的方式（http）来导入版本库的，那么你有必要保持这个 默协议的友好性。 git-init-db 之后，复制自[标准模板库](http://baike.baidu.com/view/3097879.htm" \t "_blank)的 $GIT\_DIR/hooks/post-update 将包含一个对 git-update-server-info 的调用，但是 post-update 默认是不能唤起它自身的。通过 chmod +x post-update 命令使能它。这样让 git-update-server-info 保证那些必要的文件是最新的。

3.　将你的主版本库推入公共版本库。

4.　git-repack 公共版本库。这将建立一个包含初始化提交对象集的打包作为项目的起始线，可能的话，执行一下 git-prune，要是你的公共库是通过 pull 操作来从你打包过的版本库中导入的。

5.　在你的主版本库中开展工作，这些工作可能是你自己的最项目的编辑，可能是你由 email 收到的一个补丁，也可能是你从这个项目的“子系统负责人” 的公共库中导入的工作等等。

你可以在任何你喜欢的时候重新打包你的这个私人的版本库。

6.　将项目的进度推入公共库中，并给大家公布一下。

7.　经过一段时间以后，"git-repack" 公共库。并回到第5步继续工作。

项目的子系统负责人（subsystem maintainer）也有自己的公共库，工作流程大致如下：

1.　准备一个你自己的工作目录，它通过 git-clone 克隆自项目领导人的公共库。原始的克隆地址（URL）将被保存在 .git/remotes/origin 中。

2.　准备一个可以给大家访问的公共库，就像项目领导人所做的那样。

3.　复制项目领导人的公共库中的打包文件到你的公共库中，除非你的公共库和项目领导人的公共库是在同一部主机上。以后你就可以通过 objects/info/alternates 文件的指向来浏览它所指向的版本库了。

4.　将你的主版本库推入你的公共版本库，并运行 git-repack，如果你的公共库是通过的公共库是通过 pull 来导入的数据的话，再执行一下 git-prune 。

5.　在你的主版本库中开展工作。这些工作可能包括你自己的编辑，来自 email 的补丁，从项目领导人，“下一级子项目负责人”的公共库哪里导入的工作等等。

你可以在任何时候重新打包你的私人版本库。

6.　将你的变更推入公共库中，并且请“项目领导人”和“下级子系统负责人”导入这些变更。

7.　每隔一段时间之后，git-repack 公共库。回到第 5 步继续工作。

“一般开发人员”无须自己的公共库，大致的工作方式是：

1.　准备你的工作库，它应该用 git-clone 克隆自“项目领导人”的公共库（如果你只是开发子项目，那么就克隆“子项目负责人”的）。克隆的源地址（URL）会被保存到 .git/remotes/origin 中。

2.　在你的个人版本库中的 master 分支中开展工作。

3.　每隔一段时间，向上游的版本库运行一下 git-fetch origin 。这样只会做 git-pull 一半的操作，即只克隆不合并。公共版本库的新的头就会被保存到 .git/refs/heads/origins 。

4.　用 git-cherry origin 命令，看一下你有什么补丁被接纳了。并用 git-rebase origin 命令将你以往的变更迁移到最新的上游版本库的状态中。（关于 git-rebase 命令，请参考 git-rebase）

5.　用 git-format-patch origin 生成 email 形式的补丁并发给上游的维护者。回到第二步接着工作。

GIT技巧

[编辑](javascript:;)

1. 在最后提交中更改Export（Export changes done in last commit ）

这个命令通常会使用定期发送已更改的项目，以方便其他人审查/集成。

|  |
| --- |
| gitarchive-o../updated.zipHEAD$(gitdiff--name-onlyHEAD^) |

2. 在两次提交之间更改Export文件（Export changed files between two commits）

同样地，如果你需要在两次提交之间更改文件，可以选择以下这段代码。

|  |
| --- |
| gitarchive-o../latest.zipNEW\_COMMIT\_ID\_HERE$(gitdiff--name-onlyOLD\_COMMIT\_ID\_HERENEW\_COMMIT\_ID\_HERE) |

3. 克隆一个特定的远程分支（Clone a specific remote branch）

如果你想从远程资源库中克隆一个特定的分支，而无需克隆整个资源库分支，那么下面的这段代码将对你有用。

|  |
| --- |
| gitinit 　　gitremoteadd-tBRANCH\_NAME\_HERE-foriginREMOTE\_REPO\_URL\_PATH\_HERE 　　gitcheckoutBRANCH\_NAME\_HERE |

4. 从不相关的本地资源库中应用补丁（Apply patch from Unrelated local repository）

这里有个快捷方式可帮助你实现。

|  |
| --- |
| viewplaincopytoclipboardprint? 　　git--git-dir=PATH\_TO\_OTHER\_REPOSITORY\_HERE/.gitformat-patch-k-1--stdoutCOMMIT\_HASH\_ID\_HERE|gitam-3-k |

5. 检查分支是否在其它分支中遭到更改（Check if your Branch changes are part of Other branch）

cherry这个命令，能够检查你的分支在其他分支中是否被更改。它会在当前的分支上显示变化，并注明+或-标识符。+代表不存在，-表示在现有的分支中存在。

|  |
| --- |
| viewplaincopytoclipboardprint? 　　gitcherry-vOTHER\_BRANCH\_NAME\_HERE 　　#Forexample:tocheckwithmasterbranch 　　gitcherry-vmaster<br> |

6. 启动一个无历史记录的新分支（ Start a new Branch with No History）

有时，你想启动一个新的分支，但并不想运行漫长的历史记录，例如，你想将代码放置在一个公共的域中（开源），但又不想共享历史。

|  |
| --- |
| gitcheckout--orphanNEW\_BRANCH\_NAME\_HERE |

7. 从其他分支签出文件但无需切换分支（ Checkout File from Other Branch without Switching Branches ）

这里将教你如何获取想要的文件。

|  |
| --- |
| gitcheckoutBRANCH\_NAME\_HERE--PATH\_TO\_FILE\_IN\_BRANCH\_HERE |

8. 忽略追踪文件中的更改（ Ignore Changes in a Tracked File ）

如果你是在某个团队中工作，他们都在使用同一个分支，也许你会频繁使用提取/合并（fetch/merge），但这有时需要重置特定的配置文件，这就意味着在每次合并后你必须去做更改。现在，使用这个命令，你可以要求Git忽略更改特定文件。

|  |
| --- |
| gitupdate-index--assume-unchangedPATH\_TO\_FILE\_HERE |

9. 检查已提交部分是否在发布的版本中遭到更改（Check if committed changes are part of a release）

name-rev这个命令可以告诉你已提交到最新版本的某个位置。使用这个代码可帮助你检查，提交的部分是否在已发布版本中遭到更改。

|  |
| --- |
| gitname-rev--name-onlyCOMMIT\_HASH\_HERE |

10. 用复位替代合并（Pull with rebase instead of merge ）

当某项特性分支被合并到主流中，此时该分支合并会在Git中以合并提交来进行记录。但是当团队中多个成员在同一个分支上工作时，常规的合并会导致多个合并消息在日志中呈现混乱状态。因此，你可以使用复位（rebase）来保持历史清晰，清除无用的合并消息。

|  |
| --- |
| gitpull--rebase |

此外，你还可以通过配置一个特定的分支来复位。

|  |
| --- |
| gitconfigbranch.BRANCH\_NAME\_HERE.rebasetrue[4] |

11. 保存http用户/密码，增加http上传数据的大小

git config --global credential.helper store

git config --global http.postBuffer 524288000

参考资料

1.  [Git](http://baike.baidu.com/redirect/56a8DBMLJQ4YrOdlCYGe3ZUPhzxM0waKNl5BGb-7clWfkyf6Wfu20GV7A91t0A1Y) ．Git（官方网站）[引用日期2014-04-4]

2.  [分布式版本控制系统 Git](http://baike.baidu.com/redirect/d700o4kuhSOap3fNM2jP0L2SIx5iTmIVilrdFnD-6cXrEOwZ0biXhiHB74_pJZjoeUCb1_Beb7VN) ．开源社区网[引用日期2012-09-25]

3.  [如何修改git库的author](http://baike.baidu.com/redirect/00ea11IeZsXVClBw5yzMC2bNUuXSAuTRQQizw-yvmtUImXkjBksLO8MO7yF5ynzzUHqX0mT9lRjAWLFCGboKppWy08mX67ieXQ) ．highsea的小站．2013-0718[引用日期2013-07-21]

4.  [直接拿来用！10段超有用的Git命令行代码](http://baike.baidu.com/redirect/d95fxpnwa5ZNCjFVMGdvfzUXbQIYXmUVGM2-0QZmwzmH6nJRlS8j-0lUtA975o6gKdPlsfUH7x4u3ZNxAFo8ERjy_n1fRd26bfrYe1WS) ．TechTarget SOA[引用日期2015-06-24]

# 简单使用Git和Github来管理自己的代码和读书笔记

[收藏](javascript:void(0);)

[卜星星](https://my.oschina.net/bxxfighting/home)

* 发表于 2年前
* 阅读 11444
* 收藏 45
* 点赞 5
* [评论 7](https://my.oschina.net/bxxfighting/blog/378196#comment-list)

[**开程序员的淘宝店！寻找开源技术服务伙伴！>>> »**](https://zb.oschina.net/shop/join)  https://my.oschina.net/img/hot3.png

摘要: 简单使用Git和Github来管理自己的代码和读书笔记，新手入门指南，快速使用指导。根据最新版本的git更新过，更新日期是2015年11月13日

更新日期：2015年11月13日

 以前不知道使用代码管理工具，最后写的一些东西都没有了，由于硬盘坏了或者不小心格式化了之类的，后来使用了Git

和Github来托管自己的代码和读书笔记方便了不少，到哪里只要有网就可以把自己的东西拷贝下来继续使用。

我这里简单的记录一下我使用的过程，最简单的使用都是，高级的功能我一直没有使用到，虽然买一本《Git权威指南》

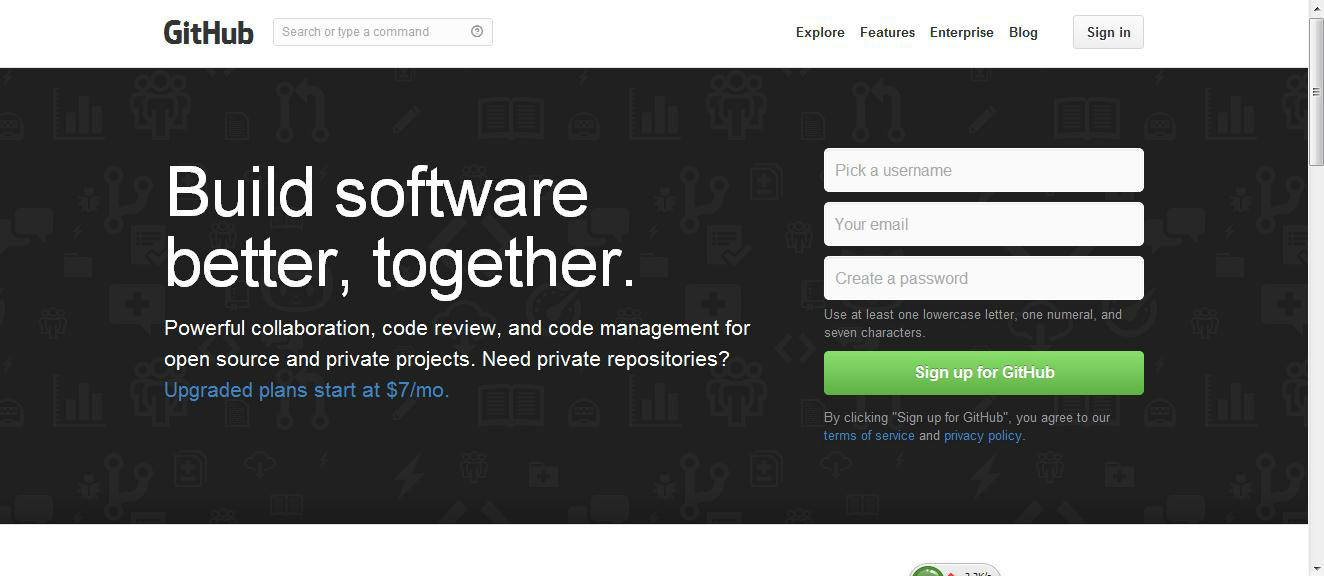
但是很多东西用不到就不能够真的会。下面开始简单介绍我使用的方法，我这个是在windows上使用的。我使用分两种情况，

因为我的代码都是在Linux下写的，所以在linux下主要是托管代码用，在windows下主要是托管笔记使用的，比如一些PDF

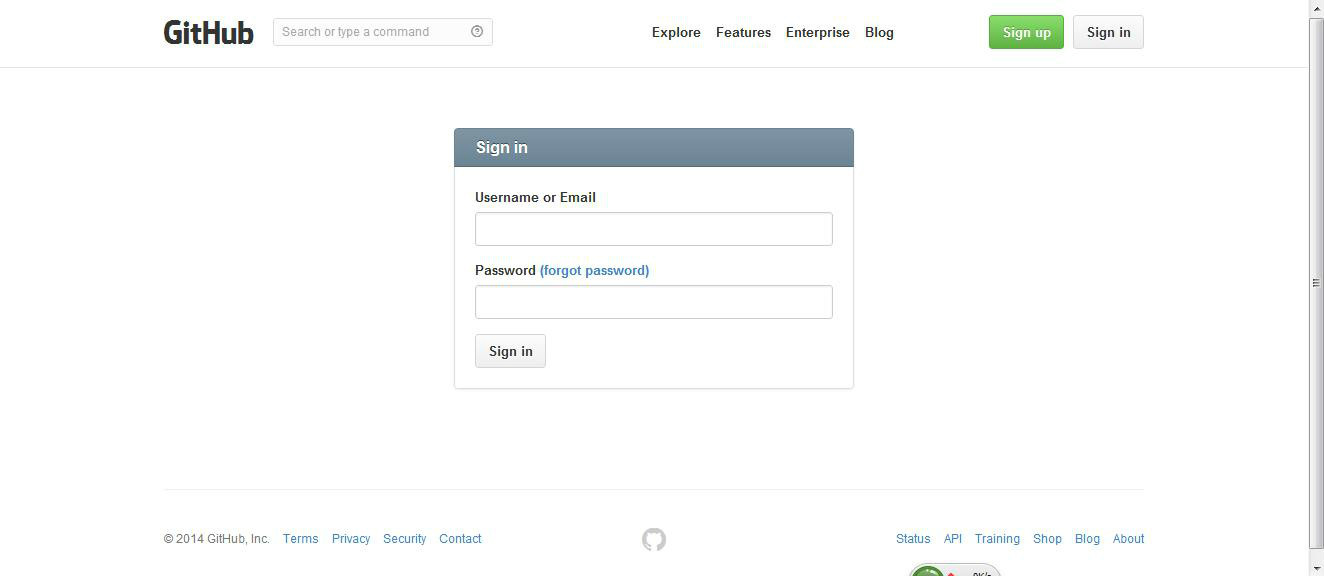
文档，我在看的时候会加入自己的注释，这样使用托管功能，在哪里都可以接着注释，不用总是拷贝或者总是需要复制。

1. 先注册github.com的账号官方网站: <https://github.com/>

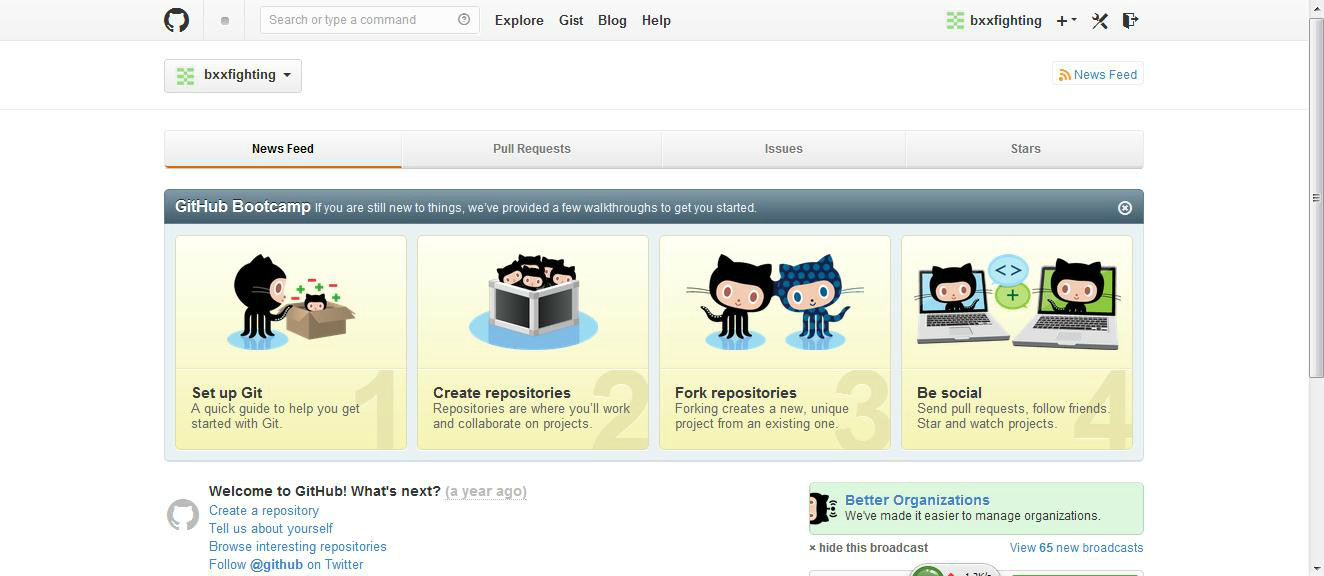
**注册界面，第一个用户名，以后会用到，我的是bxxfighting**

****

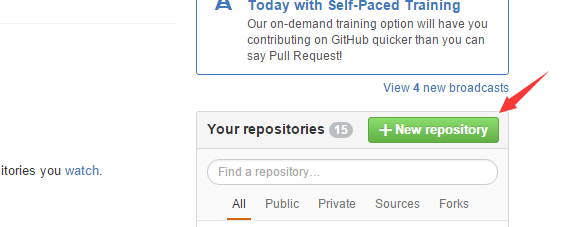
**2.      登录界面**

****

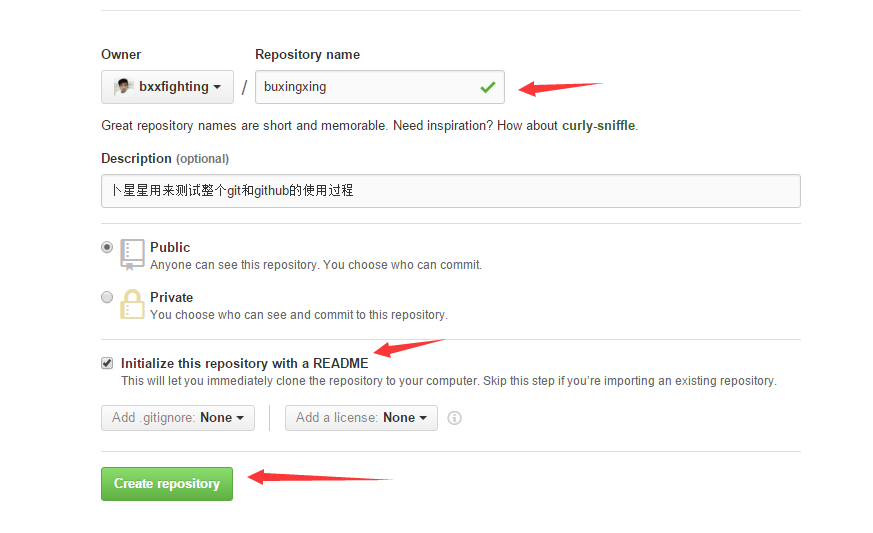
**3.      登录成功后界面**

****

4.**创建仓库,我现在创建一个测试仓库叫buxingxing,仓库分公开的和私有的,公开的是免费的,私有的是收费的,我使用的是公开的仓库,如下创建方式**

****

点击New repository按钮,弹出如下界面,第一行填仓库名,就叫个buxingxing,第二行是对这个仓库的描述,之后那个Public就是公共仓库的意思,接下来的README就是在仓库里创建一个README文件,可以往里写一些介绍你这个项目的功能之类的东西,再下面那个Add gitignore按钮,可以选择你这个项目是用什么语言之类的,后面那个License我没有选,点击”Create repository”



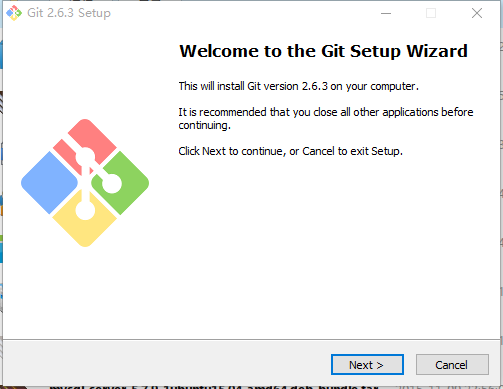
**5.      创建仓库成功后,界面如下显示,可以点击README.md来编译这个文件**

****

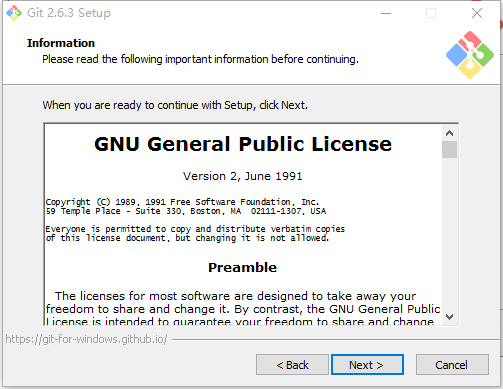
在windows下安装Git

**1.      下载网址: http://git-scm.com/download/**

**2.      下载完毕,打开安装,点击”next”**

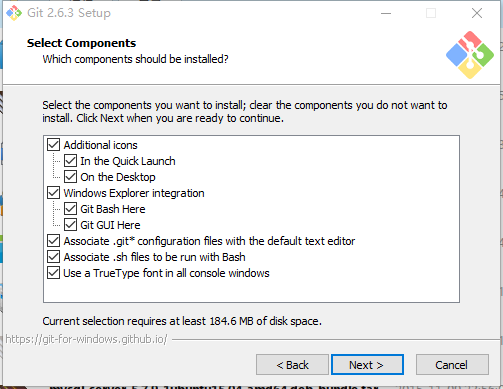


**3.      点击”next”**

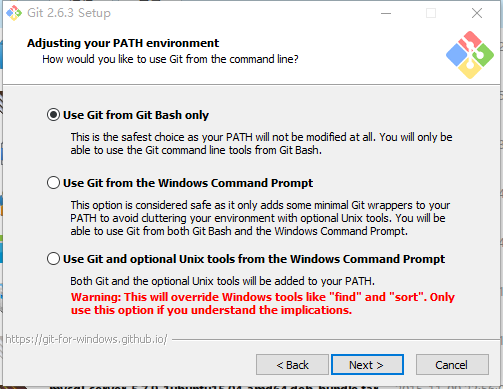
****

**4.      可以更改安装路径后,点击”next”,我没有更改，就是使用的默认路径。**

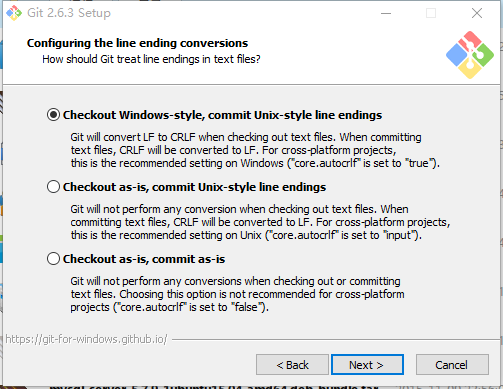
**5.      这里我把所有的选项都选中了。**

****

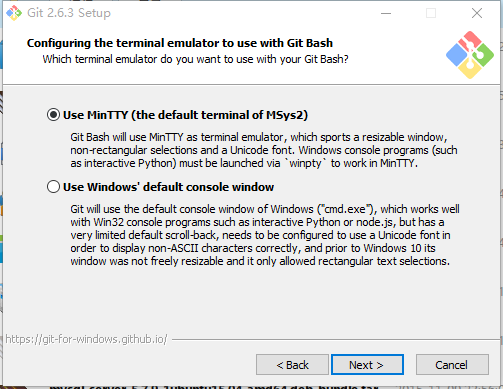
**6.      直接默认，点next.**

****

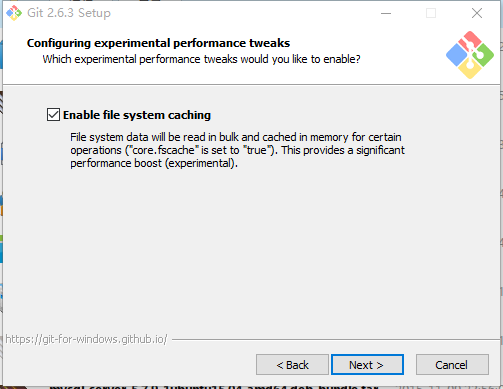
**7.      默认设置，点Next**

****

**8.      默认设置，点击next。**

****

**9.     默认设置，点击”next”**

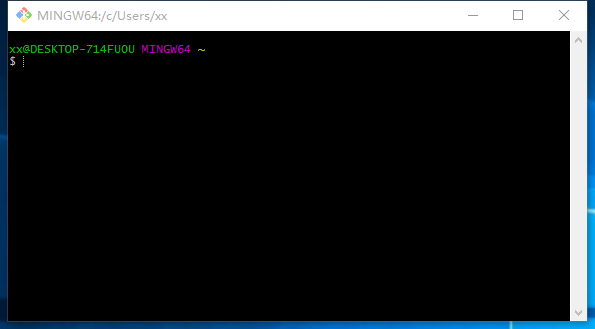
****

使用git和github托管项目代码

**1.      双击图标”Git Bash”**

****

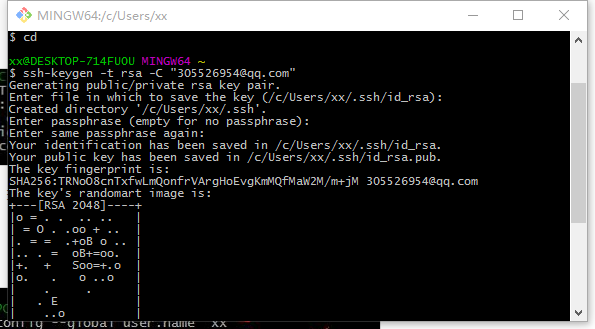
**2.      打开界面如下**



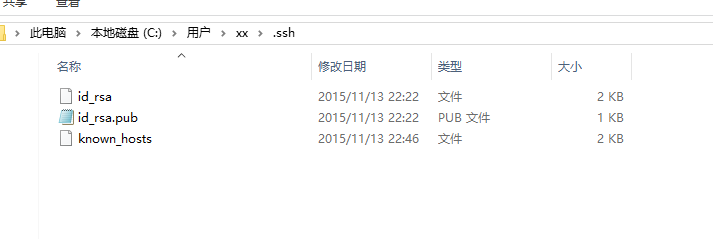
**3.      配置Git,图示如下:**

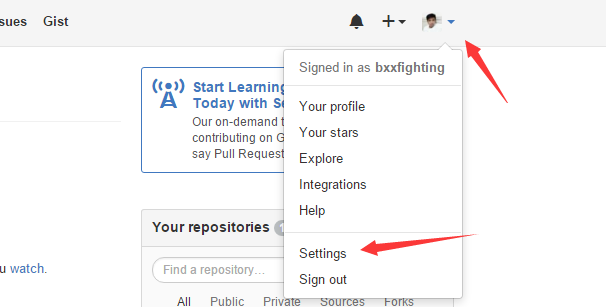
a)        先输入ssh-keygen –t rsa –C “邮箱地址”,注意ssh-keygen之间是没有空格的,其他的之间是有空格的

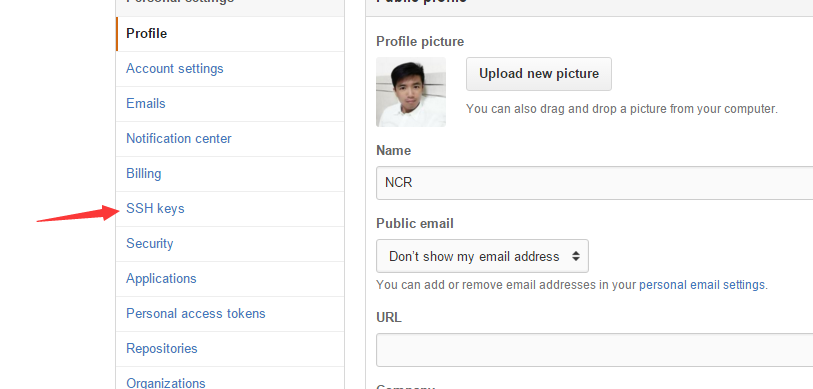
b)        回车之后,会出现一行,让你输入一个保存密钥的地方,括号里面是它默认的位置，这里会让你输入几次内容，都不用输入，直接回车就可以了，可以看到如图的效果（这里最好都不要输入，直接回车，我第一次更改了保存的路径，就出了问题，少了一个文件）：

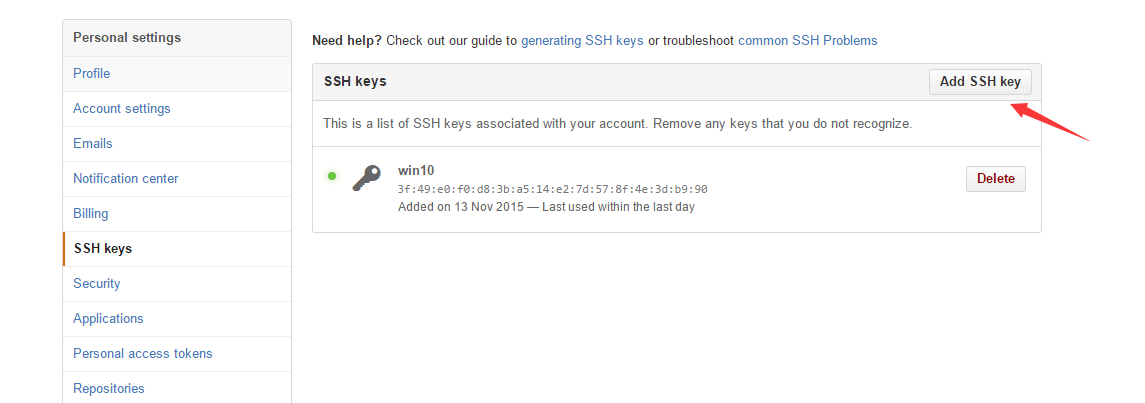


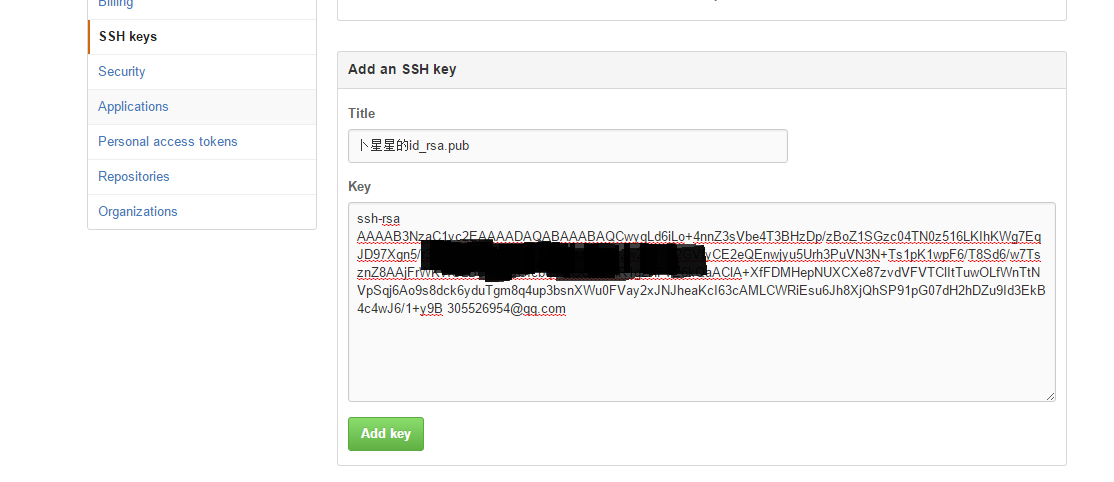
c)        回车之后,这样密钥就生成了,可以打开id\_rsa.pub（位置根据你的电脑来看）来查看,我使用的是记事本直接打开的这个文件,里面的所有内容就是这个密钥,一会需要使用的时候,就直接全选复制就可以了

**d)        现在转到github网站上去配置一下ssh key,点击箭头指示的三角图标，选择Settings，然后点击左侧的SSH Keys，之后点击右侧的Add SSH Key，这样就会出现添加SSH Key的界面，在Title这一栏填一个名字，名字随意起，之后打开刚才生成的那个文件id\_rsa.pub，全选复制里面的内容到Key这一栏中，点击Add Key按钮完成操作，这时你填的邮箱会收到一封确认的邮件，不用管它**

****



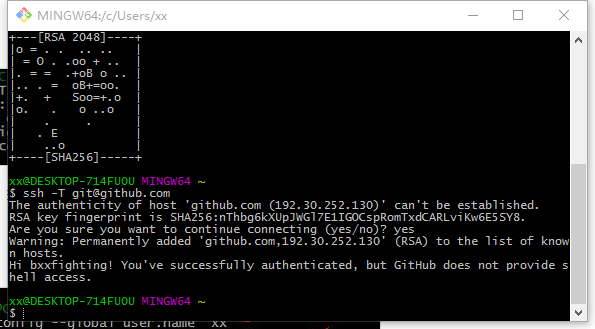




**e)        验证一下是否设置成功,在git bash下输入如下命令：**

ssh –T [git@github.com](mailto:git@github.com)

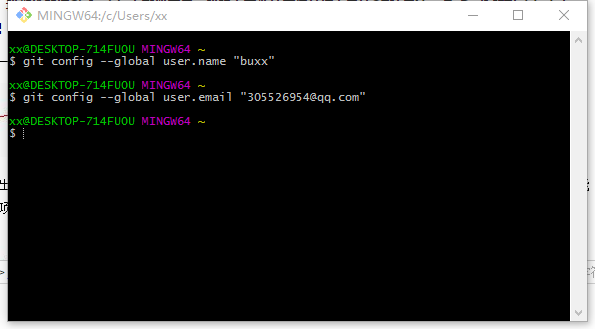
如果你是第一次，会让你输入yes或no,这时输入yes就可以了，其它显示就和我这个是一样的。如果你的是出现不是这些内容，有可能是显示权限问题什么的，就应该是我上面提到的那种情况，你看一下你生成密钥时是否操作正确，目录下是否有那个known\_hosts这个文件

****

**f)        现在配置一下用户名和邮箱：**

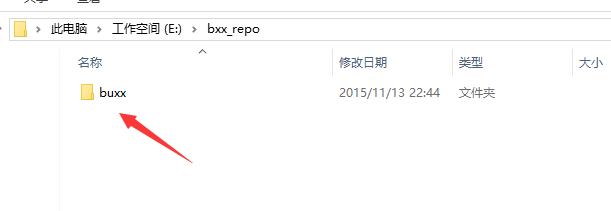
git config –global user.name “用户名”

git config –global user.email “邮箱”



**4.      到现在为止，我们就算把Git和github配置完了，现在就来托管我们的项目吧，刚才我们已经在github上面创建了一个叫buxingxing的仓库，那么我们现在就在本地创建一个目录，来管理这个仓库。**

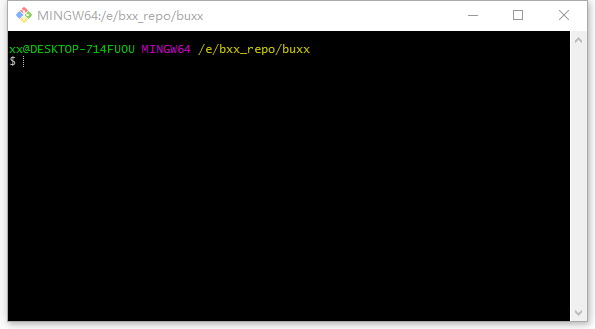
a)        随意创建了一个目录叫buxx



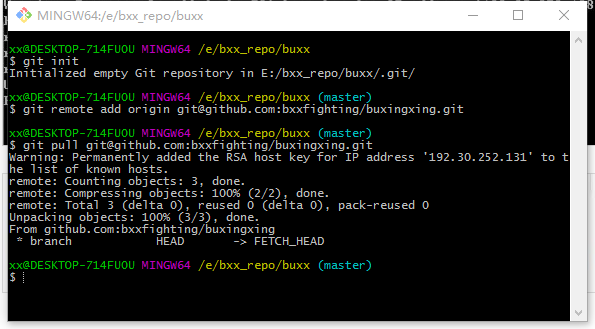
b)        右击目录，出现的菜单中有Git Bash Here，点击它。

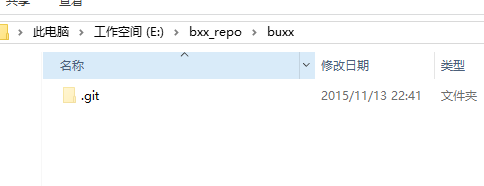


c) 这时候就在这个目录上打开了我们的终端。



d) 这时候输入 git init，来完成初始化工作。这时候目录里面就多了一个.git的目录了。注意中一共有三个命令，init/remote/pull均在上面显示，下面几步效果都在这张图中显示出来了。





现在继续在终端上输入如下命令，增加对我们github上创建的buxingxing仓库的管理。

git remote add origin git@github.com:bxxfighting/buxingxing.git

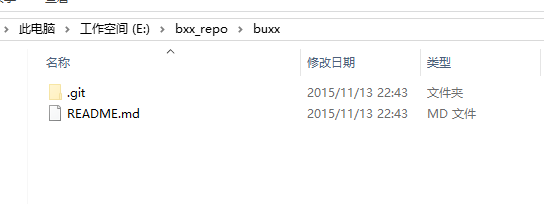
其中bxxfighting是我在网站上注册时使用的用户名，buxingxing.git是我为这个项目建立的仓库名，在网站上显示是这样的：



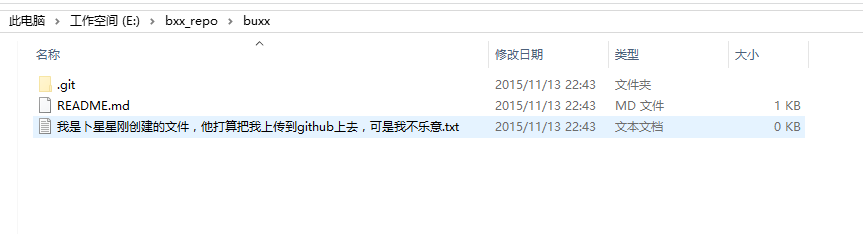
e)        由于我建立仓库的时候创建README.md之时，已经算一次提交了，我需要先在本地同步一下仓库的内容,命令如下：

git pull git@github.com:bxxfighting/buxingxing.git

完成的效果如下图，并且本地目录下多出了README.md文件

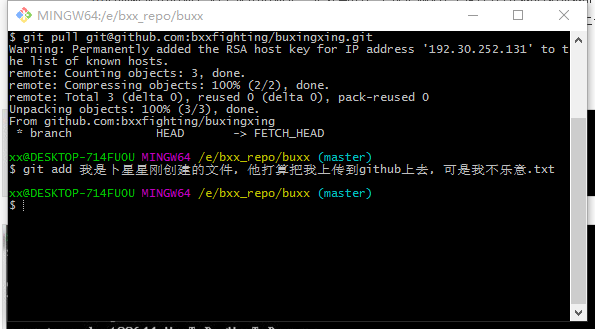


现在我们在本地目录创建我们的文件，比如下图是我创建的文件：



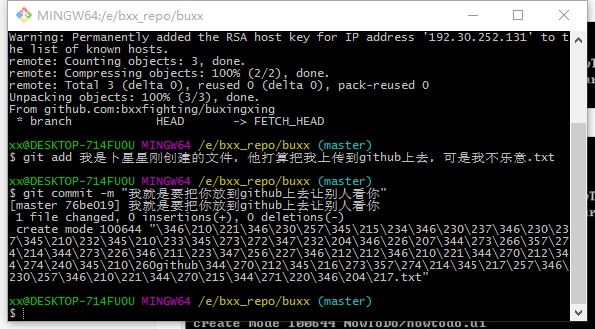
f)        下面就要把我刚创建的文件上传到到仓库上去了，首先执行增加命令，如下：

git add .    （这后面是一个英文的句号）



add后面加了一个点，是想要提交所有文件，如果想提交指定的文件，可以写文件名，执行完增加命令后，要执行提交命令，如下：

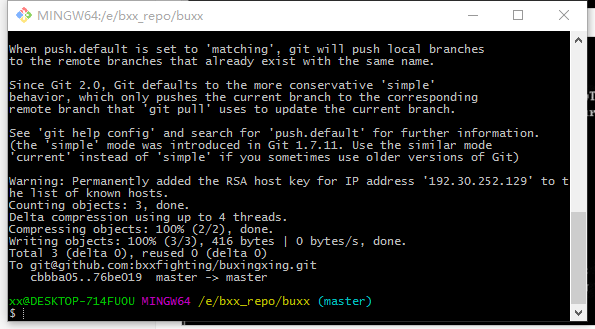
git commit –m “这里写下你自己的记录本次提交内容的信息”



-m后面跟提示信息，这个提示信息是一定要写的，不仅是规则，同时也方便我们记录我们提交的过程，写清晰为什么提交或修改了什么是非常有用的，提交完成后，我们就要把它推送到远程仓库上去了，命令如下：

git push [git@github.com:bxxfighting/buxingxing.git](mailto:git@github.com:bxxfighting/NowToDo.git)

这样就完成了我们要做的所有任务





现在就基本上可以使用了，每次增加了新文件就先add，然后commit，如果只是改了文件的内容，只执行commit就行了，当然最后一步都是要执行push，把所以改变推送到我们的github上去托管。

其实，这里有一处提醒，就是最后push那一步，提醒中说，已经有更牛逼的办法了。所以，你自己去根据提示去看看什么牛逼的办法吧，如果懒，就这么用就行倒是。

到这就差不多了，我每次用其实都来看看，不是总是用，有点记不住，第一次这样就过就记下了，方便以后使用。其实

这真是非常方便的，可以使用熟练，管理自己的代码和笔记很好的东西，以前管理代码，最近发现每次看电子书，里面写上笔记，但是同步费劲，现在决定用它来管理了，回去在自己的笔记本上也这样来用，就可以同步自己看的进度了。

# Git可视化极简易教程 — Git GUI使用方法

**前言**

之前一直想一篇这样的东西，因为最初接触时，我也认真看了廖雪峰的教程，但是似乎我觉得讲得有点多，而且还是会给我带来很多多余且重复的操作负担，所以我希望能压缩一下它在我工作中的成本，但是搜索了一下并没有找到满意的教程，新的一年自己梳理一下自己的经验。

可能男生们大神比较多，觉得Git是如此简单，便已觉得命令行操作就是SO EASY，甚至或许有看不起可视化这样面对低端用户的心理，好的，那您就当我是水货可以右上角了。

我一直觉得类似GIT这样的东西，他对于我而言只是个不完全必须的工具，我并不想成为使用他的专家，类似的东西，今天有GIT，明天可能有GAT，或者GAY？所以快速地掌握它我需要的重要日常操作，最好是10分钟，那就好了，如果你有类似的想法，好吧，那不要废话了，咱们赶紧开始。

**（全文限windows系统。）**

**何为GIT？**

[安装GIT](http://git-scm.com/download/)，大致了解下[GIT是做某子的](http://baike.baidu.com/link?url=_aNWZrVpQm9L89S4CYR66kVd6MZhQWXY5mO8zJfoTCEvQ7rkZdZOYOOyzA5IGO6kL2hw34M7r2wXixw6GNTTmk9kRAXHuIav23kej67ITc7)。

**权限校验**

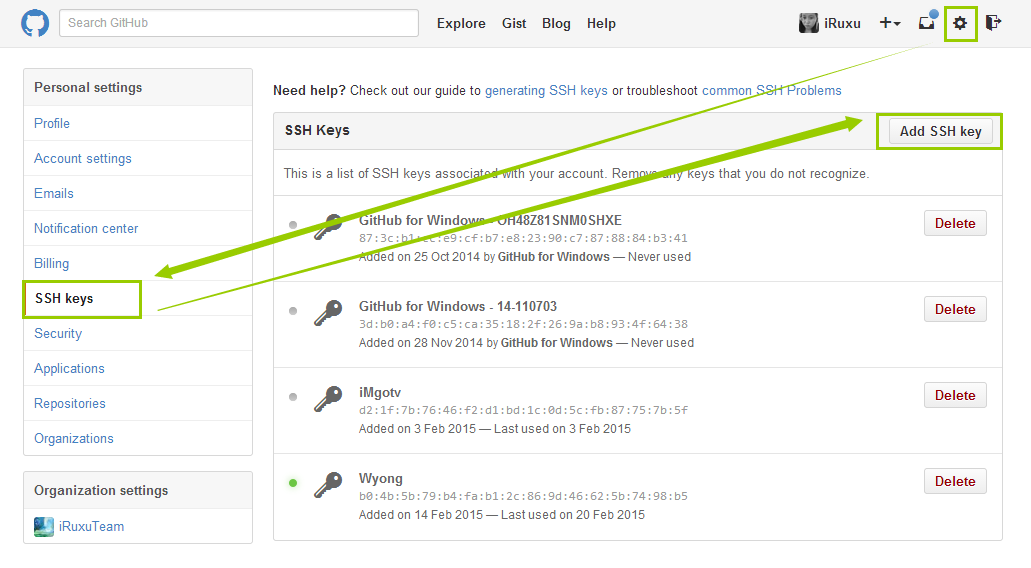
首先,您的数据保存在远端服务器一份，服务器需要对您的身份识别。一段RSA加密字符串。

启动GUI，菜单-帮助，【Step1-**创建密钥】**Generate SSH KEY

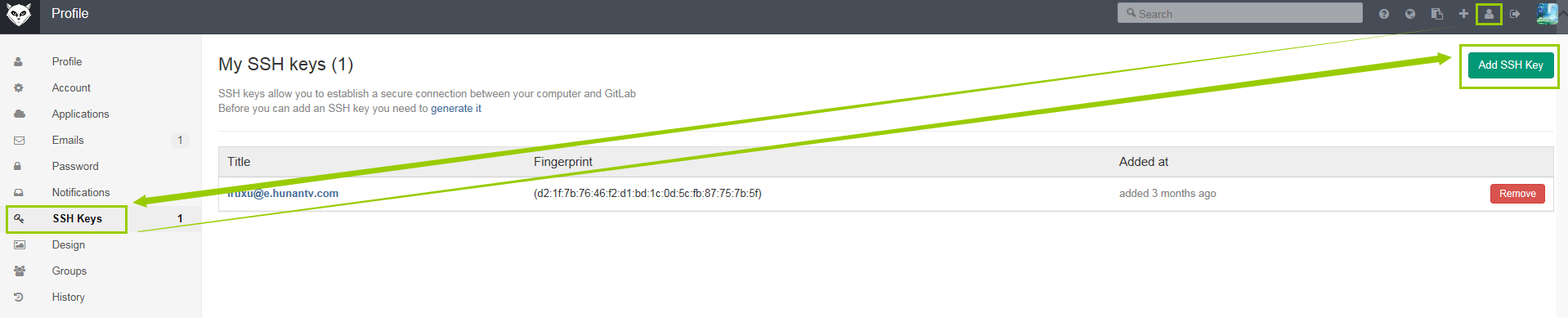
[](http://www.runoob.com/wp-content/uploads/2015/03/271102462211960.png)

【Step2-**添加密钥】**去你的代码托管服务器，你的账号设置中，添加它。

比如在Github中的地址，title随意，比如你可以用Home,company等作为标识来区别。

[](http://www.runoob.com/wp-content/uploads/2015/03/271104566587275.png)

Gitlab中的演示

[](http://www.runoob.com/wp-content/uploads/2015/03/271108115808959.png)

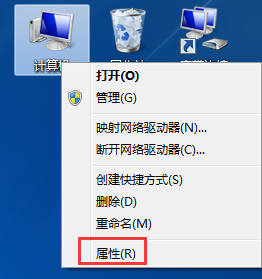
**账号保存**

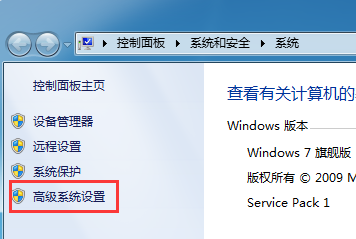
如果不做设置的话，每次提交的时候，都会询问你填写密码。于是我们先来把这个设置好。

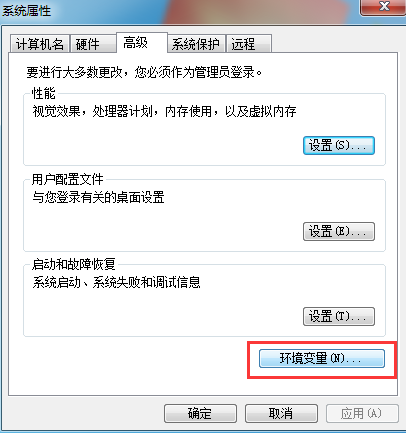
【Step3.1-添加环境变量**】**

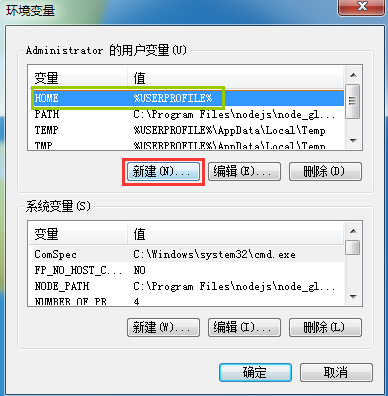
**我的电脑 - 属性 - 高级系统设置 - 环境变量 - 新建变量**

**变量名HOME，变量值%USERPROFILE%**

**[](http://www.runoob.com/wp-content/uploads/2015/03/271117100641718.png)**

[](http://www.runoob.com/wp-content/uploads/2015/03/271117191748667.png)

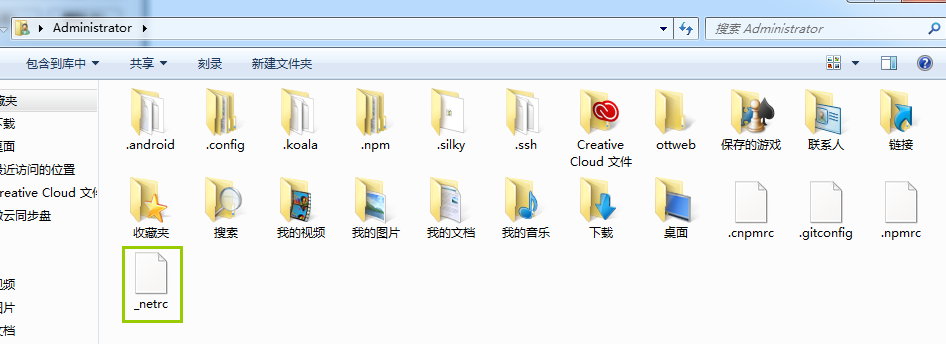
[](http://www.runoob.com/wp-content/uploads/2015/03/271117300493045.png)

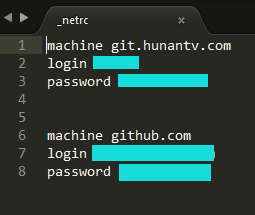
[](http://www.runoob.com/wp-content/uploads/2015/03/271117384245853.png)

**【Step3.2-创建账号文件】**

**开始 - 运行 中打开%Home%，即windows的管理员账号文件夹。**

**新建一个名为"\_netrc"的文件，填写你要保存的服务器地址及账号密码，保存。**

**[](http://www.runoob.com/wp-content/uploads/2015/03/271123307214691.png)**

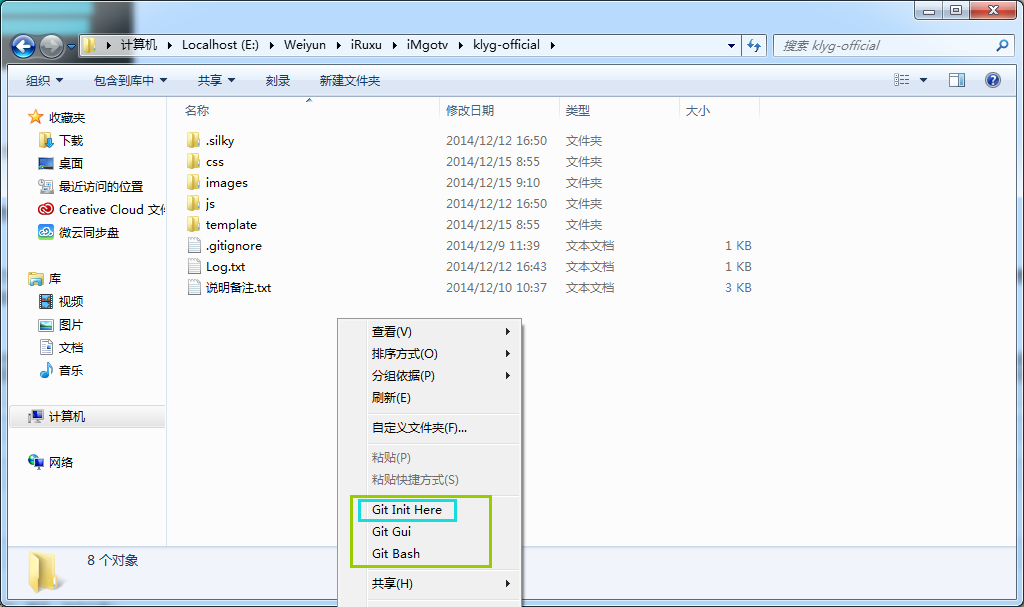
[](http://www.runoob.com/wp-content/uploads/2015/03/271123401437312.png)

**操作流程**

如果你用过SVN的话就会大致了解操作流程，如果没有也没关系。

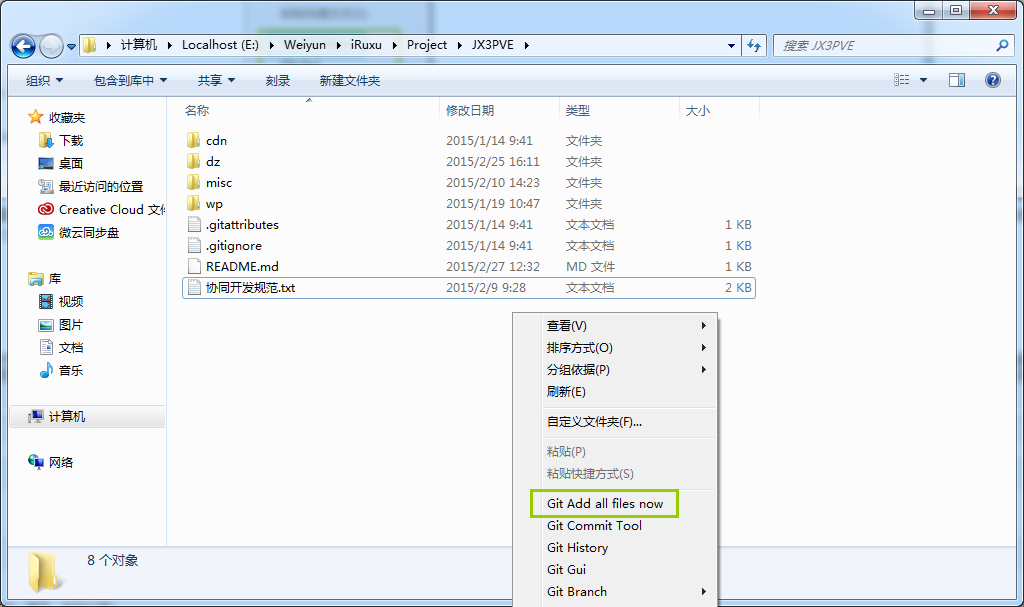
**初始化**（Git init）

顾名思义，就是新建一个项目，跟你用PS新建一张画布一样。在你新建好的文件夹中右键创建即可，若点击Git bash则以此目录作为当前目录进入命令行状态。

[](http://www.runoob.com/wp-content/uploads/2015/03/271234025648572.png)

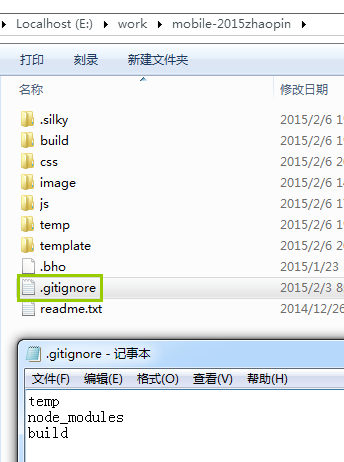
**添加**（Git add）

添加并不是提交代码到远程Git库，Git也并不会你修改了代码它自动帮你保存你修改的每一个过程。你修改了很多文件，但未必所有的修改，最终打算提交上去，那么哪些是你打算提交的，你可以添加进来待会提交，叫做缓存改动。很简单，比如本地电脑上我有整个项目完整的东东，甚至包含了账号密码的一些文件，但是我只是ADD除账号密码之外的文件，并不缓存账号密码文件的改动。不被ADD它就不会参与后续的操作。通常我都会直接全部缓存，它会自动寻找所有有改动的文件，而不需要提交的文件放在忽略的文件夹中。（关于忽略下面我们就会说到）

[](http://www.runoob.com/wp-content/uploads/2015/03/271238259396476.png)

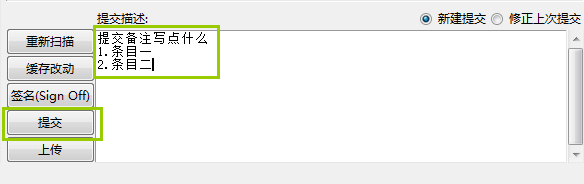
**忽略**（.gitignore）

但实际上大部分我们的文件都是一起提交的，并不会逐一去甄选，又或者类似PSD这样的大源文件以及并不作为产品最终展示的过渡文件，我们可以统一放在临时文件夹中，并忽略此文件夹。

[](http://www.runoob.com/wp-content/uploads/2015/03/271134259394618.png)

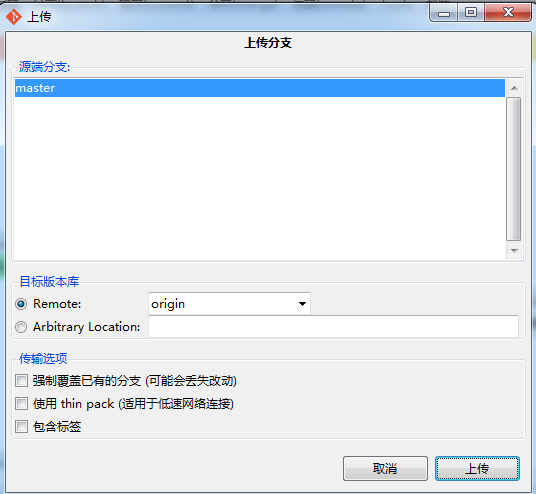
**提交**（Git commit）

提交则代表此前被添加ADD的文件已确认被提交到Git库了。需要注意的是，如果你改变代码的缩进（尽管没有修改内容），默认状态下会被识别为整个代码全部变更。**提交的时候是要求必须要写备注的**。

[](http://www.runoob.com/wp-content/uploads/2015/03/271240398302205.png)

**上传**（Git push）

顾名思义，上传则是上至远端服务器了，小伙伴们可以看到咱们的渣渣代码了（好羞涩。

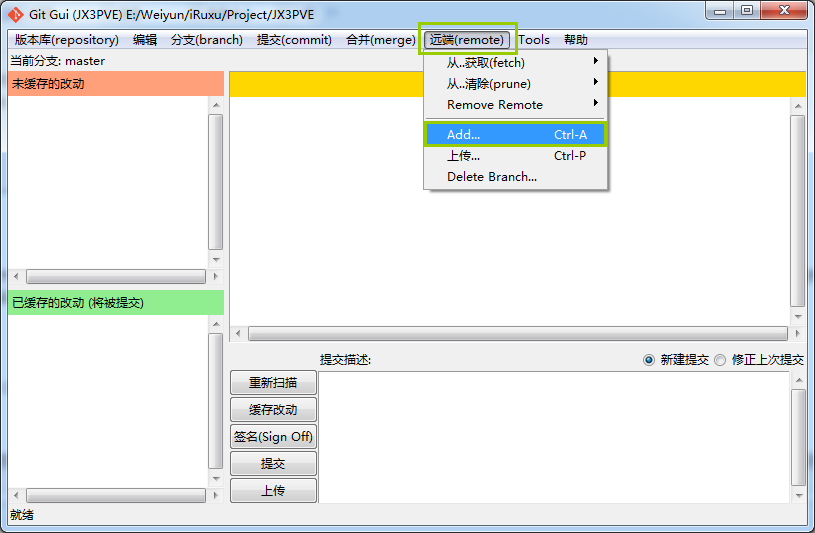
[](http://www.runoob.com/wp-content/uploads/2015/03/271241402522046.png)

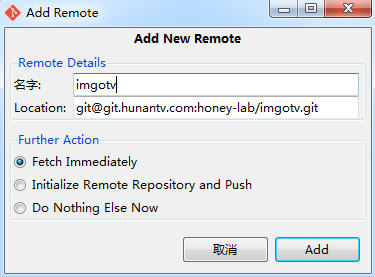
**获取远程代码**（Git remote/fetch）

比如你在公司做好的东东，今夜难眠十分亢奋，回家准备继续搬砖，那咱们就在家里的电脑上，同上进行好各种安装配置账号，先把公司做好的东东嫩下来（不过公司是内网不可以，但是假如是Github上是可以的）。又或者和小伙伴一起开发个啥，你也要先fetch他的下来。至于怎么操作，下面上图。现在你只要知道，大大们下齿全露刷牙表情对你口口念念的fetch是个啥子～

[](http://www.runoob.com/wp-content/uploads/2015/03/271145255335341.png) 来，跟我念，fetch~~（我怕你们脑补不出来……）

**先来设置与远程地址的关联，Git remote：**

[](http://www.runoob.com/wp-content/uploads/2015/03/271242561118002.png)

[](http://www.runoob.com/wp-content/uploads/2015/03/271244220336453.png)

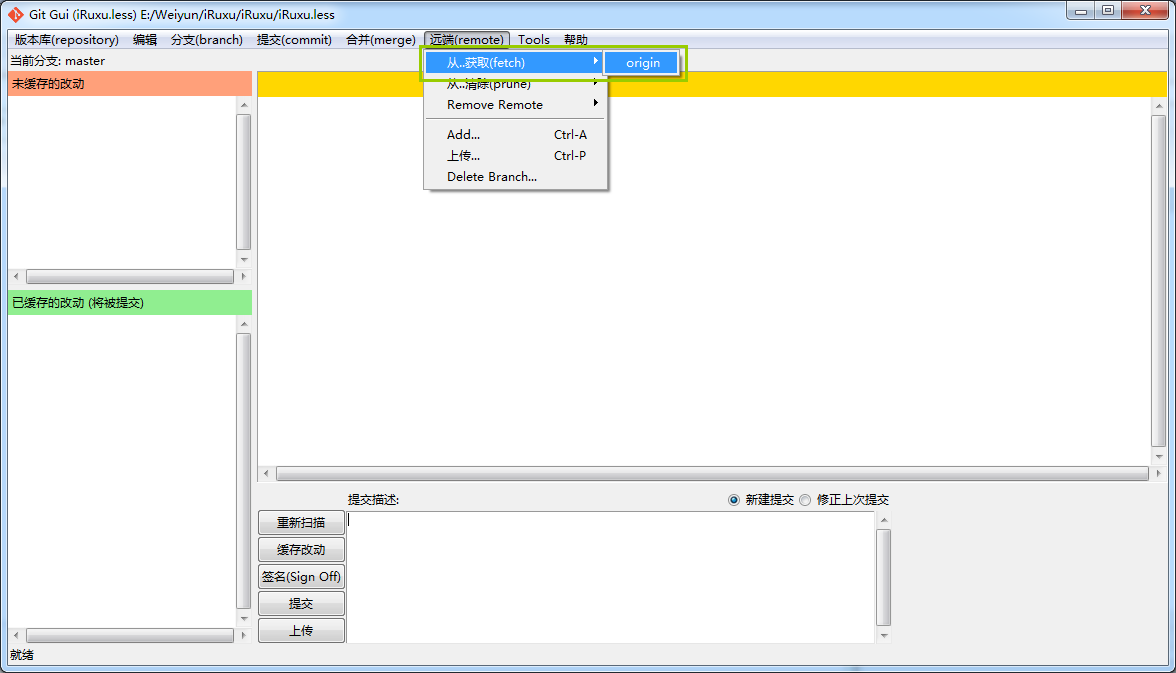
填写SSH地址与项目名。下面有3个选项：

第一个：立刻获取最新改动（所以如果是本地克隆远程一个项目，也可以这样操作）。

第二个：本地新建的项目，初始化远程仓库并发布过去。

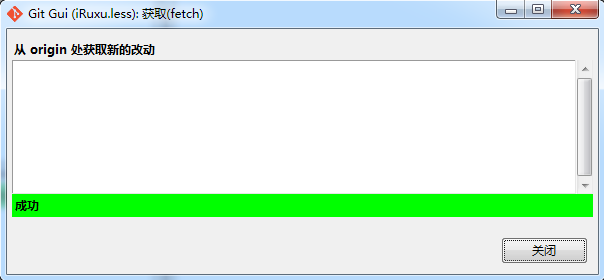
第三个：什么也不做。

**在项目的进行过程中，获取仓库的最新改动Git fetch**

[](http://www.runoob.com/wp-content/uploads/2015/03/271259025495085.png)

选择从远程仓库哪个分支中获取更新，如果没有则只有主支。

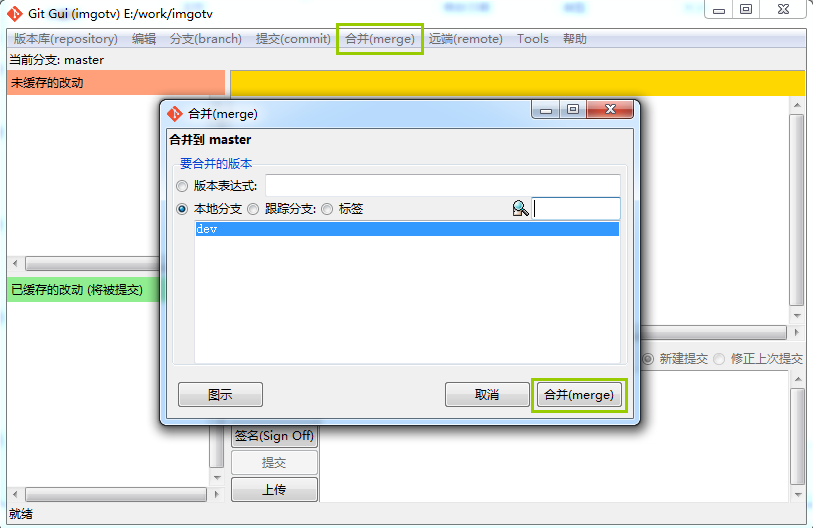
提示成功则改动的已经被存放到临时区了，你一会还需要进行合并操作，如果没有任何改动，则列表中是空的，比如：

[](http://www.runoob.com/wp-content/uploads/2015/03/271300329558531.png)

**合并**（Git merge）

请注意啦，不管你本地有没有代码，fetch之后呢，是都要merge的，也就是说，fetch下来后，大大的代码还在一个小黑屋里，我们需要把它装到自己兜里。

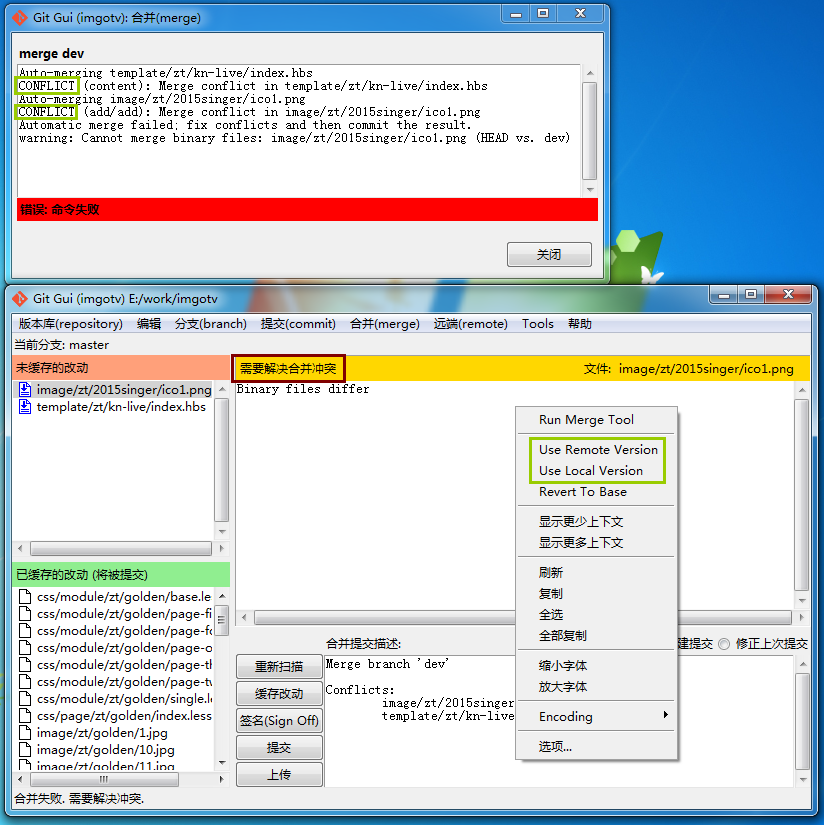
选择合并 - 本地合并，然后选择本地的分支（如果你没有创建分支，则只有1个主支master）

[](http://www.runoob.com/wp-content/uploads/2015/03/271254126278757.png)

**冲突处理**（Conflict）

合并的过程中可能会出现一些红色的文件与一堆叹号，这时候慌慌张张的点啥它都不管用，不用担心，不是程序坏了，只是有冲突的文件，例如A童鞋写了width:1180px，你写了width:auto。那到底用你们谁的呢。

在GUI界面正文区，**正文区右键**可以选择，Use local version（使用本地版本）或Use remote version（使用远程版本），到底用你的还是小伙伴的？或者你也可以自己再整合。

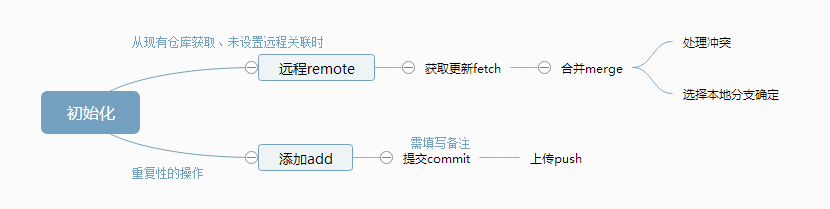
[](http://www.runoob.com/wp-content/uploads/2015/03/271257485644055.png)

其他还有分支和一些高级功能，如果需要了解可以自己再摸索摸索，以上的操作已经可以满足简单的开发需求了。

**总结**

1.先进行安装，密钥添加，账号等一次性操作。

2.操作流程：

[](http://www.runoob.com/wp-content/uploads/2015/03/271314500648180.png)

是不是觉得so easy了呢，赶紧have a try!

# [win下GIT GUI的使用教程](http://blog.csdn.net/allenjay11/article/details/51941829)

标签： [git](http://www.csdn.net/tag/git)[gui](http://www.csdn.net/tag/gui)

2016-07-18 14:36 7178人阅读 [评论](http://blog.csdn.net/allenjay11/article/details/51941829#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/allenjay11/article/details/51941829#report)

本文章已收录于：

http://static.blog.csdn.net/images/category_icon.jpg分类：

git（4） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpghttp://static.blog.csdn.net/images/arrow_triangle_up.jpg

[作者同类文章](http://blog.csdn.net/allenjay11/article/category/6216053)*X*

版权声明：本文为博主原创文章，未经博主允许不得转载。

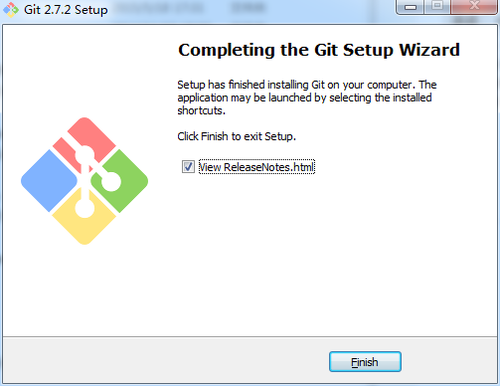
现在很多都有git来托管项目或者来查找资料，但是看起来操作不是很方便，现在由于win下可以直接使用git gui，让使用git变得方便，当然这只是针对日常简单的使用，如果想详细的使用，可以去参考廖学峰写的教程

### git下载地址

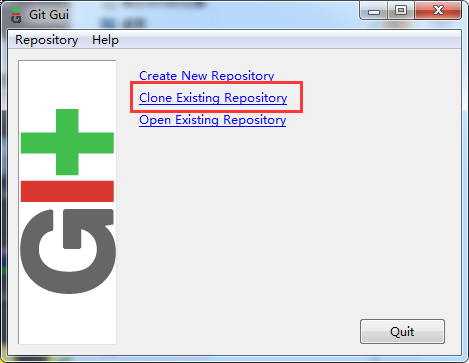
<http://pan.baidu.com/s/1b8mC3s>

### git安装步骤

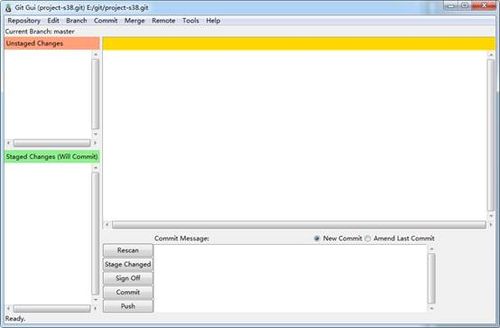
1.选择下载的安装包http://szimg.mukewang.com/57046fcb0001035e02690027.jpg

2.安装步骤省略，直接下一步即可。   
出现下面的图片安装成功   


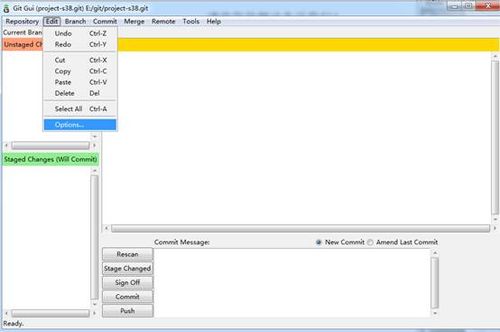
### 使用git GUI克隆已存在的仓库

1.在本地新建一个文件夹test，选中后点击右键，选择Git GUI Here   
出现下图,选择第二个，克隆已经存在的仓库   


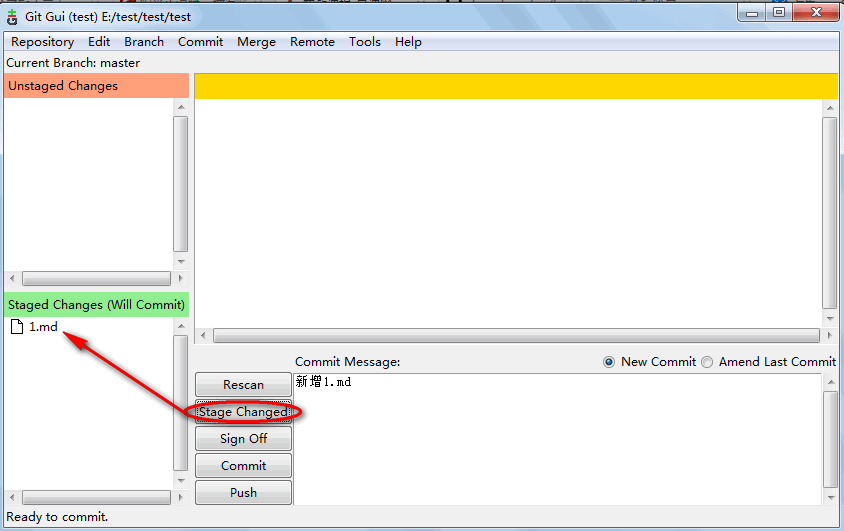
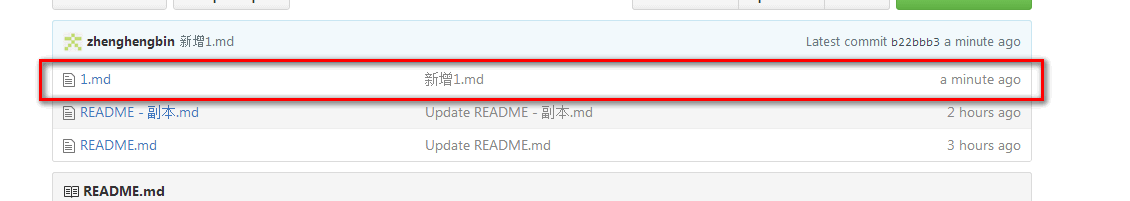
2.填写Source Location（源地址）和Target Directory

3.完成克隆   
点击上图的clone后，会提示下输入你github的用户名和密码，出现下面的界面，克隆完成。你可以看看你本地克隆的文件。   


### 使用git GUI进行Push操作

1.修改属性配置-改为utf-8   


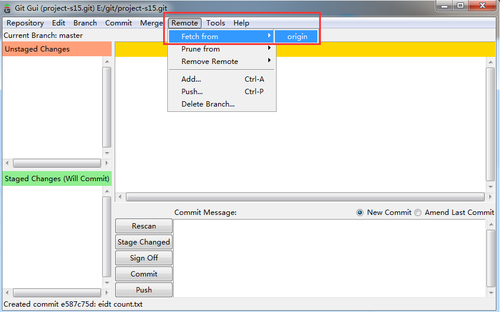
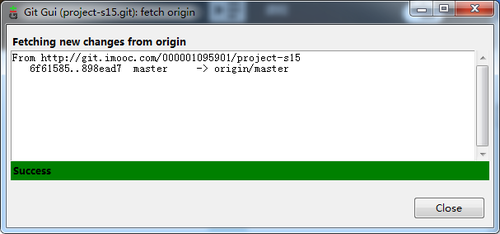
2.获取改动文件之后，选择Stage Changed.我在本地添加了一个1.md文件

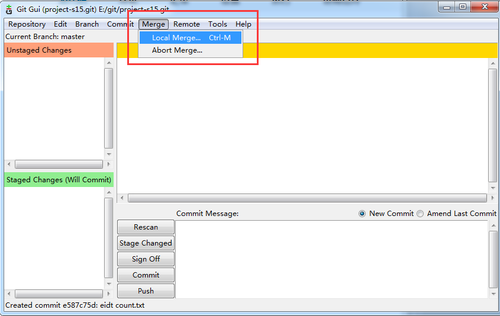
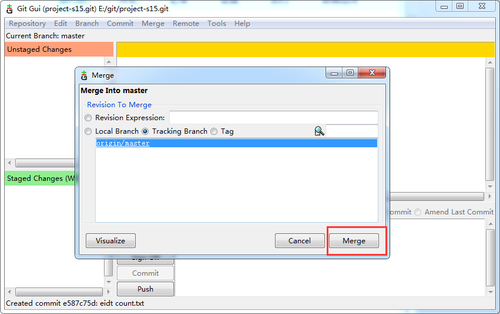
3.选择Stage Changed，在新弹窗中选择是   
  
4.然后点击 commit，再点击push，输入用户名和密码，就提交到远程仓库了。下图是远程仓库,就看到刚刚提交的内容   


### Remote和Merge操作

1.服务端重置项目后，把本地文件直接push，或修改本地文件，待扫描到有文件改动时在提交 同样也会出错，这是由于git本地与服务器端文件不一致产生冲突导致的。   


2.遇到这种情况需要先从远程仓库拉取一下文件，解决下合并冲突的问题再push

1.获取仓库的最新改动Git fetch。选择Remote->Fetch from-origin，同样需要输入项目的用户名和密码   
  
成功后显示   


2.解决一下文件冲突问题   
菜单->Merge->Local Merge,在Merge弹窗中选择Merge按钮，成功后会有Success弹窗出现。   
  


3.如果合并之后检测到文件有冲突，是会提示Command Failed失败的，并且Git Gui是会获取到冲突文件，对于冲突文件的解决，可以右键选择使用远程版本还是本地版本，并且本地已经被打开的话，是会提示是否overwritten.按照此方法挨个解决下冲突文件即可。

# 解决fatal: refusing to merge unrelated histories

本文讲的是把git在最新2.9.2，合并pull两个不同的项目，出现的问题如何去解决fatal: refusing to merge unrelated histories

我在Github新建一个仓库，写了License，然后把本地一个写了很久仓库上传。

先pull，因为两个仓库不同，发现refusing to merge unrelated histories，无法pull

因为他们是两个不同的项目，要把两个不同的项目合并，git需要添加一句代码，在git pull，这句代码是在git 2.9.2版本发生的，最新的版本需要添加--allow-unrelated-histories

假如我们的源是origin，分支是master，那么我们 需要这样写git pull origin master ----allow-unrelated-histories需要知道，我们的源可以是本地的路径

bemin@Dell\_E4200 MINGW64 /c/bemin (master)

$ git pull origin master --allow-unrelated-histories

From github.com:asdfgh129/bemin

\* branch master -> FETCH\_HEAD

Auto-merging .gitignore

CONFLICT (add/add): Merge conflict in .gitignore

Automatic merge failed; fix conflicts and then commit the result.

bemin@Dell\_E4200 MINGW64 /c/bemin (master|MERGING)

$ git commit -m "pull from origin and dix conflicts"

error: Committing is not possible because you have unmerged files.

hint: Fix them up in the work tree, and then use 'git add/rm <file>'

hint: as appropriate to mark resolution and make a commit.

U .gitignore

fatal: Exiting because of an unresolved conflict.

bemin@Dell\_E4200 MINGW64 /c/bemin (master|MERGING)

$ git add .gitignore

bemin@Dell\_E4200 MINGW64 /c/bemin (master|MERGING)

$ git commit -m "pull from origin and dix conflicts"

[master a9937f7] pull from origin and dix conflicts

bemin@Dell\_E4200 MINGW64 /c/bemin (master)

$ git push origin master

Counting objects: 2437, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (1863/1863), done.

Writing objects: 100% (2437/2437), 20.74 MiB | 99.00 KiB/s, done.

Total 2437 (delta 554), reused 2433 (delta 553)

remote: Resolving deltas: 100% (554/554), done.

To github.com:asdfgh129/bemin.git

c3b8939..a9937f7 master -> master