logo.png

Git

H5 web前端

[www.bufanui.com](http://www.bufanui.com)

# git安装

## 1.1什么是GIT

* 是一个源代码管理工具
* 让源代码可以被追溯，主要记录每次变更了什么，谁主导这次变化
* 人为的维护非常麻烦
* GIT是Linux之父Linus Torvalds当年为了维护管理Linux的源代码写的一个工具



* Git 之前 很多使用 svn vss tfs hs ......
* git简史

同生活中的许多伟大事物一样，Git 诞生于一个极富纷争大举创新的年代。

Linux 内核开源项目有着为数众广的参与者。 绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991－2002年间）。 到 2002 年，整个项目组开始启用一个专有的分布式版本控制系统 BitKeeper 来管理和维护代码。

到了 2005 年，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了 Linux 内核社区免费使用 BitKeeper 的权力。 这就迫使 Linux 开源社区（特别是 Linux 的缔造者 Linux Torvalds）基于使用 BitKcheper 时的经验教训，开发出自己的版本系统。 他们对新的系统制订了若干目标：

* 速度
* 简单的设计
* 对非线性开发模式的强力支持（允许成千上万个并行开发的分支）
* 完全分布式
* 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

自诞生于 2005 年以来，Git 日臻成熟完善，在高度易用的同时，仍然保留着初期设定的目标。 它的速度飞快，极其适合管理大项目，有着令人难以置信的非线性分支管理系统（参见 [Git 分支](https://git-scm.com/book/zh/v2/ch00/_git_branching)）

  Git 有三种状态，你的文件可能处于其中之一：已提交（committed）、已修改（modified）和已暂存（staged）。 已提交表示数据已经安全的保存在本地数据库中。 已修改表示修改了文件，但还没保存到数据库中。 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。

由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。

## 1.2下载

<http://git-scm.com/download/win>

安装方法跟普通软件一样。

## 1.3 git工作原理

Git有三个工作状态，分别是已修改（modified）、已暂存（staged） 和已提交（committed）。

由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。

Git仓库目录是Git用来保存项目的元数据和对象数据库的地方。 这是Git 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。

工作目录是对项目的某个版本独立提取出来的内容。这些从Git仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。

暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在Git仓库目录中。有时候也被称作“索引”（Index），不过一般说法还是叫暂存区域。



基本的Git工作流程如下：

1、在工作目录中修改文件。

2、暂存文件，将文件的快照放入暂存区域。

3、提交文件，找到暂存区域的文件，将快照永久性存储到Git仓库目录。

## 1.4 创建Git本地仓库

### 1.4.1 Git基础

**1、配置用户**

git config --global user.name "自已的名字"

git config --global user.email "自已的邮箱地址"

--global 配置当前用户所有仓库

--system 配置当前计算机上所有用户的所有仓库

注：配置用户只需要执行1次，可以重复使用。

**2、初始化仓库**

我们如果想要利用git进行版本控制，需要将现有项目初始化为一个仓库。

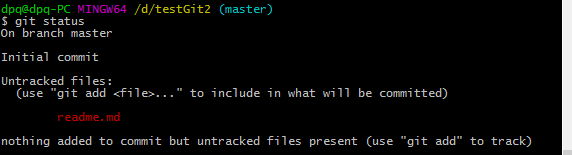
a) git init

图像 001

git init只是创建了一个名为.git的隐藏目录，这个目录就是存储我们历史版本的仓库，ls -al 可以查看。

**3、查看文件状态**

通过git status可以检测当前仓库文件的状态

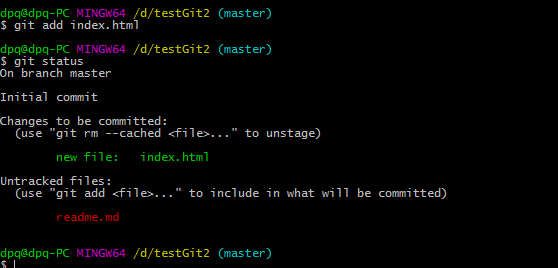


注：git会忽略空的目录

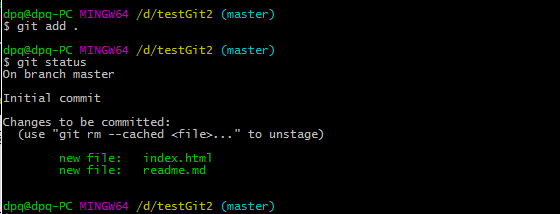
**4、添加文件到暂存区**

git add文件名/ 文件路径 “\*”或-A代表所有

一般用 git add . 添加所有



放到暂存区的文件被标记成了绿色，等待提交。



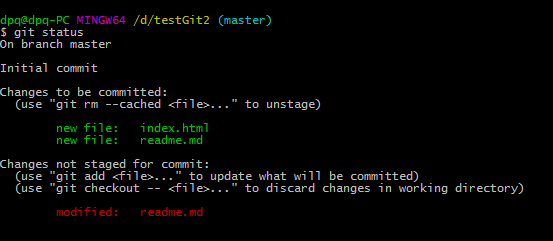
注：颜色是工具给添加的，目的是增加可读性并不是git统一的。

### \*\* gitignore忽略文件

在项目根目录下创建一个.gitignore文件，可以将不希望提交的罗列在这个文件里，如项目的配置文件、node\_modules等

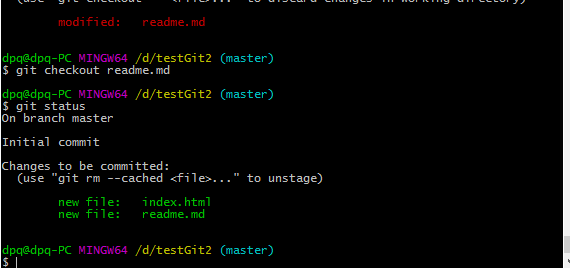
5、撤销更改

修改文件，再次git status可以再次查看仓库状态



说明readme.md再次被修改了，并被标记了红色。

又经过一段时间后发现新开发的部分有Bug，想要回到之前状态，可以使用git checkout 文件名。

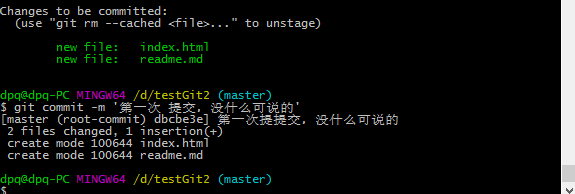


注：从暂存区还原原到工作区

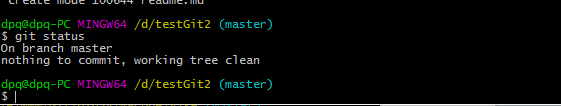
**5、提交文件**

git commit -m '备注信息'

将暂存区被标记成绿色的文件，全部提交到本地仓库存储。



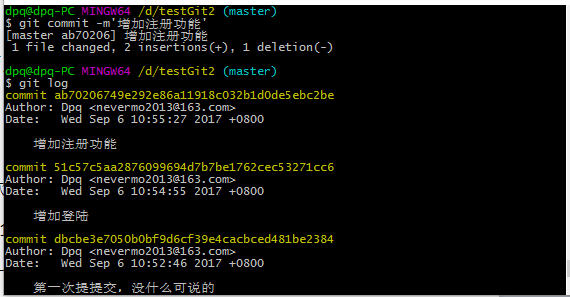
这时git status查看状态



没有什么可提交的，变的很干净。

**6、查看提交历史**

git log查看提交的历史。



我们可以查看到一次次提交记录

commit ab70206749e292e86a11918c032b1d0de5ebc2be

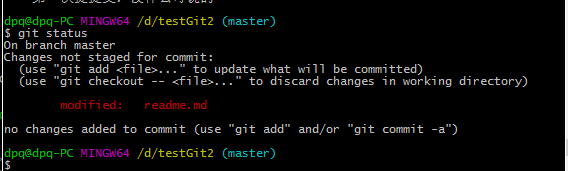
代表一次提交的唯一ID，一般称为SHA值。

注：按键盘q键退出。

**7、再次检测仓库文件状态**

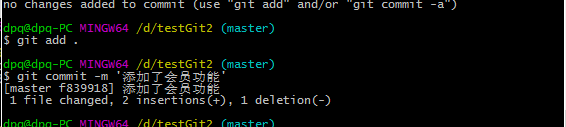
隔了好些天后，继续开发

git status 查看状态



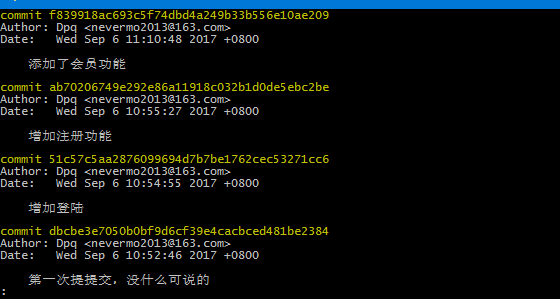
又提示有修改，等待重新添加到暂存区。

**8、重新添加暂存区然后提交**



**9、再次查看历史**

git log 可查到所有提交历史

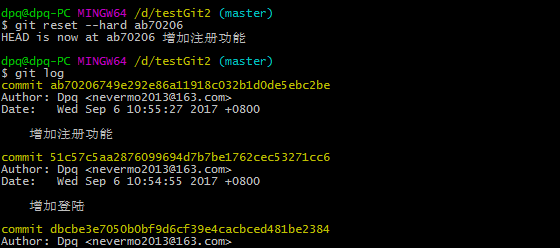


**10、恢复上一次提交的状态**

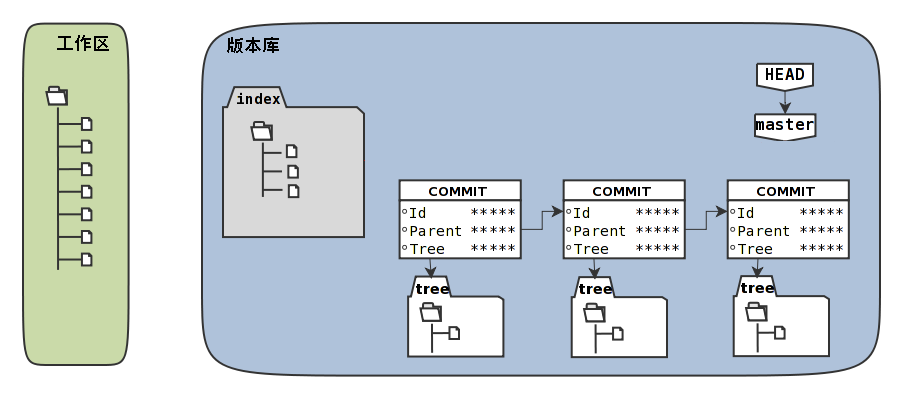
通过SHA值可以回到之前某一次的提交（时光倒流）

git reset --hard xxxx

git log再次查看发现最后一次提交成了支付功能了

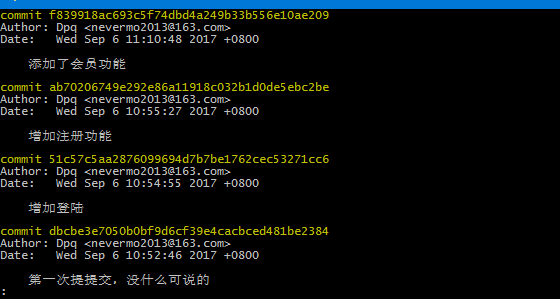


仓库示意图



### 1.4.2 Git分支

在Git的使用过程中一次提交称为历史记录（版本），并且会生成一个唯一的字符串，如下图



这个串可以代表某一个历史版本（实际使用只取前面几位就可以），

值得注意的是所有的提交（commit）实际上都是在分支（branch）的基础上进行的。

如下图所示：



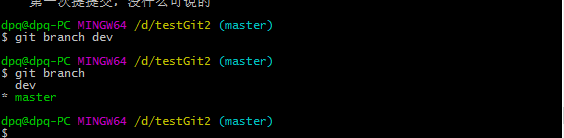
当我们在初始化仓库的时候（实际上是产生第1次提交时），Git会默认帮我们创建了一个master的分支，并且有指针（HEAD）指到了末端。

指针（HEAD）用来标明当前处于哪个分支的哪个版本，如上图指的处于master分支的最后1个版本。

我们也可以创建自已的分支

**1、创建分支**

git branch dev



新的分支会在当前分支原有历史版本的结点上进行创建，我称其为子分支如下图



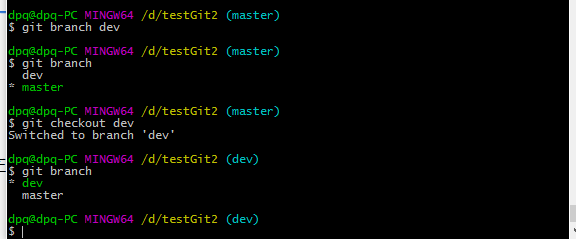
新建的子分支会继承父分支的所有提交历史。

**2、切换分支**

git checkout dev



我们发现HEAD现在又指向了dev的末端。

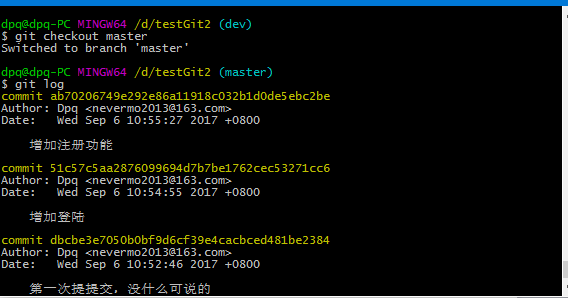


**3、再次提交操作**

修改bug后，提交，这次的提交历史版本就会记录在dev这个分支上了，并且HEAD伴随dev在移动。

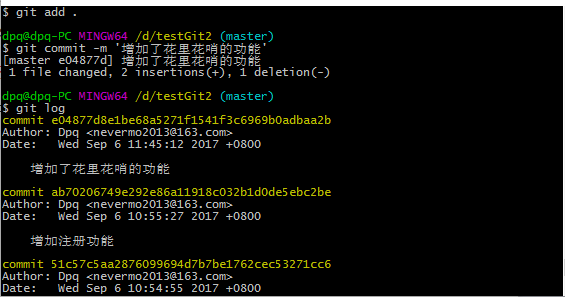


**4、当我们再次切回到master时**



当我们切换回master后，HEAD指向了master分支的末端，并且我们观察发现我们的文件内容还是原来的“模样”。

**5、继续之前的开发**



总结：当我们git checkout master时，HEAD会自动指向对应分支的末端，工作目录中的源码也会随之发生改变。

这个时候我们就在dev这个分支上的修改，而我们原来在master分支上的操作并未受到影响。

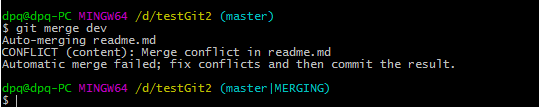
思考一个问题：

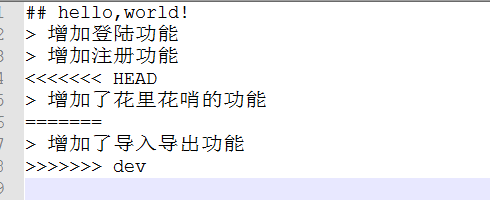
现在master这个分支上是否包含了dev的修复呢？

实际上从上图可以看出这时的master分支并没有包含有dev的修复。

1. **合并（融合）分支**

Git merge dev 把dev分支合并到主分支上



这时master会有两个父结点了，master便包含了dev里的修复了

冲突怎么处理：

手动合并。

**7、删除分支**

git branch -d dev

这时用来修复BUG创建的dev分支已经没有用处了，我们可以将它删除。

E325:注意merge mode by the ‘recursive’ strategy



## 1.5 Git远程（共享）仓库

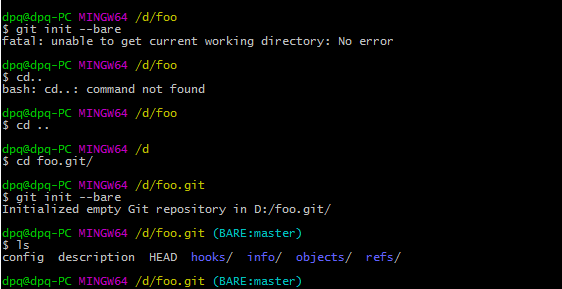
### 1.5.1创建共享仓库

Git要求共享仓库是一个以.git结尾的目录。

mkdir foo.git 创建以.git结尾目录

cd foo.git 进入这个目录

git init --bare 初始化一个共享仓库，也叫裸仓库 注意选项--bare



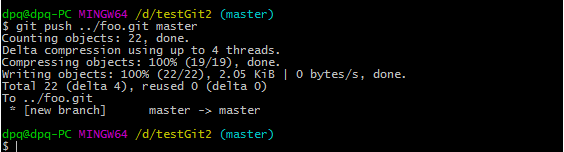
这样我们就建好了一个共享的仓库，但这时这个仓库是一个空的仓库，并且不允在这个仓库中进行任何修改。

**4、向共享仓库共享（同步）内容**

将自已开发的项目同步到这个目录中，其它开发者就可以共享你开发的项目了。

1、进入到testGit2目录

2、git push ../foo.git master

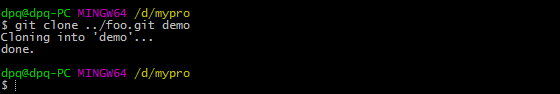


这样便把testGit2中的项目同步进了foo.git中。

**5、从共享仓库里取出内容**

1、新创建一个目录（模拟另一个开发者）

2、git clone ../foo.git demo

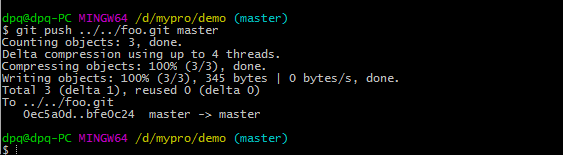


通过foo.git共享仓库，我们轻松得到了一个testGit2的副本

1. **在mypro中更新提交内容**

Cd mypro

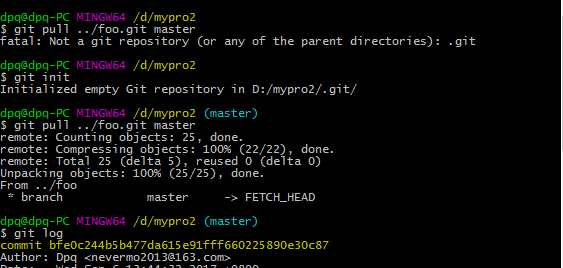
git push ../foo.git master



**7、在mypro2获取foo.git共享的内容**

cd mypro2

git pull ../foo.git master

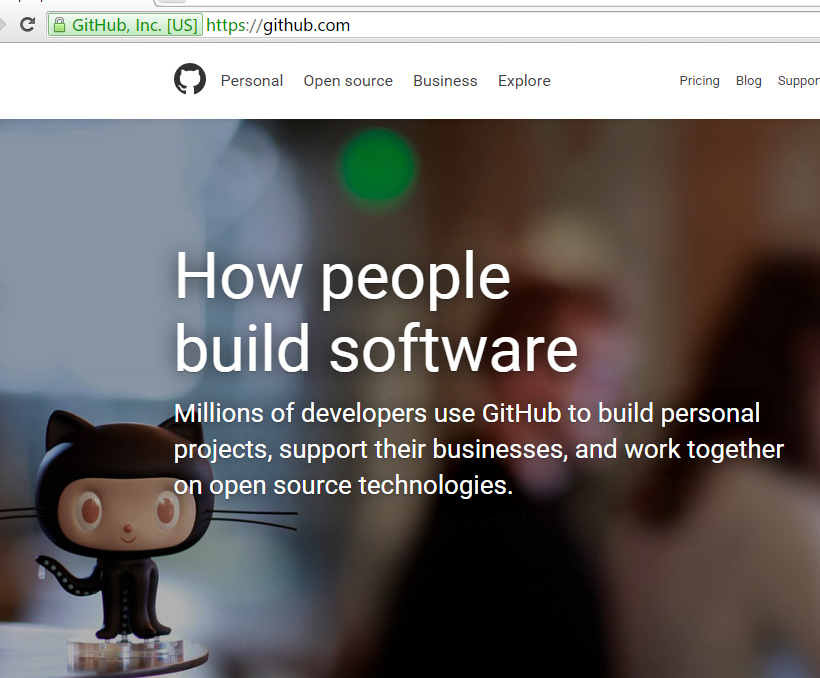


我们可以轻松的将mypro仓库里的内容，通过foo.git共享给了mypro2仓库。

# 二、github

## 2.1 什么是github

* 之前讲的git都是在本地工作，github是一个网站，一个git服务提供商，可以通过git将本地的文件托管的github上面。相当于一个远端仓库。
* 社交化编程
* 程序员的同性交友网

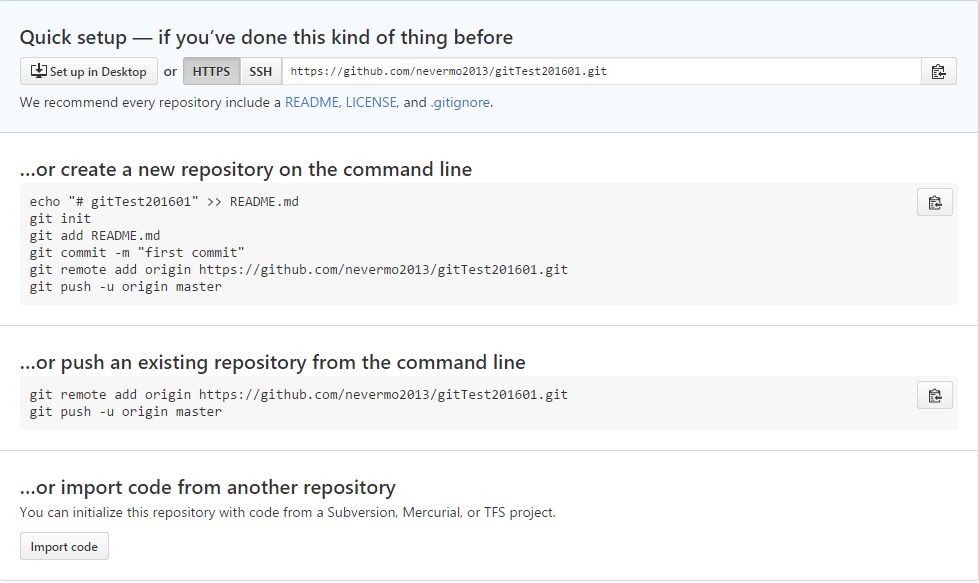


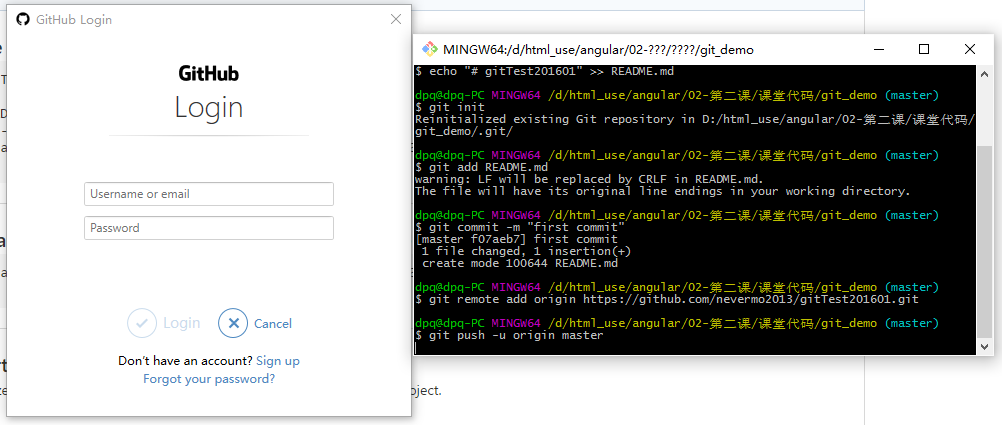
## 2.2怎么使用

* 注册
* 生成密钥

ssh-keygen -t rsa -C [“your\_email@youremail.com”](mailto:“your_email@youremail.com”)

* 创建仓库
* 把本地项目push到github上面
  + - git remote add origin git@xxx.git 添加远端仓库（第一次添加）。
    - git push –u origin master 将本地项目以流的形式push到远端的master(主分支)上。

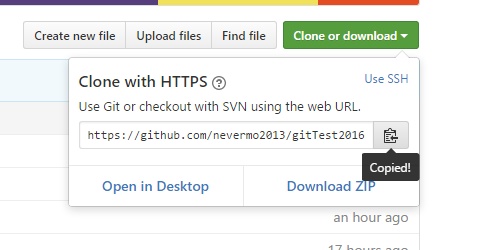




* git pull origin master 将远端的更新获取到本地
* git branch xx 创建分支
* git branch 查看分支
* git checkout xx 切换分支



* 2.2.6 git clone xxx 克隆



## 2.3 更新仓库

如果远程仓库和本地仓库同时修改怎么办？

git fetch “远程主机”

或者

git fetch “远程主机” “分支名称”

我们要注意的是，利用git fetch 获取的更新会保存在本地仓库中，但是并没有体现到我们的工作目录中，需要我们再次利用git merge来将对应的分支合并（融合）到特定分支。如下

git pull origin 某个分支， 上操作相当于下面两步

git fetch

git merge origin/某个分支

## 2.4 删除远程分支

删除远程分支git push origin --delete 分支名称

补充：git生成sshkey

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