# GIT教程

## 一、GIT简介及安装

### 1、Git简介

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。

### 2、常见的版本控制系统

我们常见的版本控制系统分为集中式（SVN，CVS）和分布式（GIT）两种，两者之间的区别：

集中式：



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。

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，那就不知道。。。。。，最要命的是如果中央服务器挂了，那一切的工作都没有办法进行。

分布式：



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，

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。

### 3、安装Git

默认在CentOS下，我们可以通过yum的方式来安装Git

[root@node1 ~]# yum install git –y

[root@node1 ~]# git version

git version 1.8.3.1

使用yum安装的Git的版本是1.8，版本较低，我们还可以通过源码编译的方式来安装Git的最新版本。

首先需要安装依赖的库：

[root@node1 ~]# yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel gcc perl-ExtUtils-MakeMaker -y

下载最新的源码包

[root@node1 src]# cd /usr/local/src/

[root@node1 src]# wget <https://www.kernel.org/pub/software/scm/git/git-2.9.5.tar.gz>

[root@node1 src]# ll

total 5792

-rw-r--r-- 1 root root 5928730 Aug 11 01:57 git-2.9.5.tar.gz

解压安装：

[root@node1 src]# tar xf git-2.9.5.tar.gz

[root@node1 src]# cd git-2.9.5

[root@node1 git-2.9.5]# make prefix=/usr/local/git all

[root@node1 git-2.9.5]# make prefix=/usr/local/git install

[root@node1 git-2.9.5]# rm -rf /usr/bin/git

[root@node1 git-2.9.5]# ln -s /usr/local/git/bin/git /usr/bin/git

[root@node1 git-2.9.5]# git --version

git version 2.9.5

至此，我们已经完成了Git的编译安装，现在Git可以安装在windows、MAC操作系统上，具体请参考：

https://git-scm.com/book/zh/v2/%E8%B5%B7%E6%AD%A5-%E5%AE%89%E8%A3%85-Git

### 4、Git初始化

安装完成 Git后，我们需要定制 Git 环境。 每台计算机上只需要配置一次，程序升级时会保留配置信息。 你可以在任何时候再次通过运行命令来修改它们

Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置变量。 这些变量存储在三个不同的位置：

1）/etc/gitconfig 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果使用带有 --system 选项的 git config 时，它会从此文件读写配置变量。

2）~/.gitconfig 或 ~/.config/git/config 文件：只针对当前用户。 可以传递 --global 选项让 Git 读写此文件，常用选项。

3）当前使用仓库的 Git 目录中的 config 文件（就是 .git/config）：针对该仓库。

设置你的用户名称与邮件地址。 这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改

[root@node1 ~]# git config --global user.name "wendong"

[root@node1 ~]# git config --global user.email "28152082@qq.com"

[root@node1 ~]# git config --list

user.name=wendong

[user.email=28152082@qq.com](mailto:user.email=28152082@qq.com)

设置完成后在当前用户的家目录会生成.gitconfig文件：

[root@node1 ~]# ll .gitconfig

-rw-r--r-- 1 root root 48 Nov 27 10:48 .gitconfig

[root@node1 ~]# cat .gitconfig

[user]

name = wendong

email = 28152082@qq.com

[root@node1 ~]#

我们可以使用使用—help选项来查看git config命令的其他用法

## 二、GIT基础

### 1、获取Git仓库

#### 1）在现在目录中初始化仓库

进入要使用Git进行管理的目录，执行如下命令：

[root@node1 ~]# mkdir git-test

[root@node1 ~]# cd git-test/

[root@node1 git-test]# git init

Initialized empty Git repository in /root/git-test/.git/

[root@node1 git-test]# ll -a

total 0

drwxr-xr-x 3 root root 18 Nov 28 07:24 .

dr-xr-x---. 4 root root 197 Nov 28 07:23 ..

drwxr-xr-x 7 root root 119 Nov 28 07:24 .git

该命令将创建一个名为 .git 的子目录，这个子目录含有你初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干。 但是，在这个时候，我们仅仅是做了一个初始化的操作，你的项目里的文件还没有被跟踪。

#### 2）克隆现有的仓库

如果你想获得一份已经存在了的 Git 仓库的拷贝，这时就要用到 git clone 命令。 这是 Git 区别于其它版本控制系统的一个重要特性，Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。 当你执行 git clone 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。

[root@node1 ~]# git clone <https://github.com/wendong866/demo.git>

Cloning into 'demo'...

remote: Counting objects: 16, done.

remote: Total 16 (delta 0), reused 0 (delta 0), pack-reused 16

Unpacking objects: 100% (16/16), done.

Checking connectivity... done.

[root@node1 ~]# ll demo/

total 16

drwxr-xr-x 3 root root 81 Nov 28 07:45 .

dr-xr-x---. 6 root root 221 Nov 28 07:45 ..

drwxr-xr-x 8 root root 163 Nov 28 07:45 .git

-rw-r--r-- 1 root root 1045 Nov 28 07:45 .gitignore

-rw-r--r-- 1 root root 11357 Nov 28 07:45 LICENSE

-rw-r--r-- 1 root root 30 Nov 28 07:45 test.txt

-rw-r--r-- 1 root root 0 Nov 28 07:45 wd.txt

[root@node1 ~]#

这会在当前目录下创建一个名为 “demo” 的目录，并在这个目录下初始化一个 .git 文件夹，从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。 如果你进入到这个新建的demo 文件夹，你会发现所有的项目文件已经在里面了，准备就绪等待后续的开发和使用。 如果你想在克隆远程仓库的时候，自定义本地仓库的名字，你可以使用如下命令：

git clone https://github.com/wendong866/demo.git mydemo

这将执行与上一个命令相同的操作，不过在本地创建的仓库名字变为 mydemo。

#### 3）.git目录

(D) hooks：这个目录存放一些shell脚本，可以设置特定的git命令后出发相应的脚本；在搭建gitweb系统或其他git托管系统会经常用到hook script

(D) info：包含仓库的一些信息

(D) logs：保存所有更新的引用记录（会在后面介绍引用）

(D) objects：所有的Git对象都会存放在这个目录中，对象的SHA1哈希值的前两位是文件夹名称，后38位作为对象文件名

(D) refs：这个目录一般包括三个子文件夹：heads、remotes和tags，heads中的文件标识了项目中的各个分支指向的当前commit

(F) COMMIT\_EDITMSG：保存最新的commit message，Git系统不会用到这个文件，只是给用户一个参考

(F) config：这个是Git仓库的配置文件

(F) description：仓库的描述信息，主要给gitweb等git托管系统使用

(F) index：这个文件就是我们前面文章提到的暂存区（stage），是一个二进制文件

(F) HEAD：这个文件包含了一个当前分支（branch）的引用，通过这个文件Git可以得到下一次commit的parent

(F) ORIG\_HEAD：HEAD指针的前一个状态

### 2、记录文件更新

#### 1）检查当前文件状态

要查看哪些文件处于什么状态，可以用 git status 命令。

[root@node1 demo]# cd ../git-test/

[root@node1 git-test]# git status

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)

我们可以看到这是个刚初始化的目录，没有未提交的文件。后面还提示我们，可以创建或者复制文件，然后使用git add命令对文件进行跟踪。

#### 2）跟踪新文件

我们首先在当前目录下创建一个新文件README，然后再查看当前文件的状态

[root@node1 git-test]# echo "My first git test project" >README

[root@node1 git-test]# ll

total 4

-rw-r--r-- 1 root root 26 Nov 28 08:13 README

[root@node1 git-test]# git status

On branch master

Initial commit

**Untracked files:**

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

README

nothing added to commit but untracked files present (use "git add" to track)

[root@node1 git-test]#

在状态报告中可以看到新建的 README 文件出现在 Untracked files 下面。 未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件；Git 不会自动将之纳入跟踪范围，而且后面的提示我们可以使用git add命令去跟踪新文件

使用命令 git add 开始跟踪一个文件。

[root@node1 git-test]# git add README

再运行 git status 命令，会看到 README 文件已被跟踪，并处于暂存状态：

[root@node1 git-test]# git status

On branch master

Initial commit

**Changes to be committed:**

(use "git rm --cached <file>..." to unstage) git add 的逆操作，具体后面讲解

new file: README

只要在 Changes to be committed 这行下面的，就说明是已暂存状态。 如果此时commit，那么该文件此时此刻的版本将被留存在历史记录中。 git add 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。我们也可以使用git add \*提交当前目录下的所有文件

#### 3）提交更新

现在我们已经把创建的文件提交到了暂存区，接下来我们就需要把暂存区的文件提交到本地版本库进行管理。注意：在每交提交前最好先使用git status查看一下，是不是所有更改的文件都已经暂存，然后再使用git commit命令提交。

[root@node1 git-test]# git commit -m "first submit"

[master (root-commit) 5e874cc] first submit

1 file changed, 1 insertion(+)

create mode 100644 README

[root@node1 git-test]# git status

On branch master

nothing to commit, working tree clean

请记住，提交时记录是放在暂存区域的快照。 Git的每一次提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较；每次准备提交前，先用git status 看下，是不是都已暂存起来了，然后再运行提交命令 git commit。

#### 4）暂存己修改的文件

我们首先再新建一个Hello.txt的文件并提交：

[root@node1 git-test]# echo "Hello world">Hello.txt

[root@node1 git-test]# git status

On branch master

Untracked files:

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

Hello.txt

nothing added to commit but untracked files present (use "git add" to track)

[root@node1 git-test]# git add Hello.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

new file: Hello.txt

接着我们修改刚才已经提交过的README文件，然后查看状态：

[root@node1 git-test]# echo "first modify" >>README

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

new file: Hello.txt

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) 使用暂存区的文件覆盖工作的文件

**modified: README**

文件 README 出现在 Changes not staged for commit 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。 要暂存这次更新，需要运行 git add 命令。 这是个多功能命令：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。 将这个命令理解为“添加内容到下一次提交中”更加合适。 现在让我们运行 git add 将"README"放到暂存区，然后再看看 git status 的输出：

[root@node1 git-test]# git add README

You have new mail in /var/spool/mail/root

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

new file: Hello.txt

modified: README

现在两个文件都已暂存，下次提交时就会一并记录到仓库。 假设此时，你想要在 README 里再加一条内容， 重新编辑存盘后，再运行 git status 看看：

[root@node1 git-test]# echo "second modify" >>README

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

new file: Hello.txt

modified: README

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

你看到了什么？ 现在 README 文件同时出现在暂存区和非暂存区。 这是为什么？ 好吧，实际上 Git 只不过暂存了你运行 git add 命令时的版本， 如果你现在提交，README的版本是你最后一次运行 git add 命令时的那个版本，而不是你运行 git commit 时，在工作目录中的当前版本。 所以，运行了 git add 之后又作了修订的文件，需要重新运行 git add 把最新版本重新暂存起来：

[root@node1 git-test]# git add README

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

new file: Hello.txt

modified: README

[root@node1 git-test]# git commit -m "second submit"

[master a306d94] second submit

2 files changed, 3 insertions(+)

create mode 100644 Hello.txt

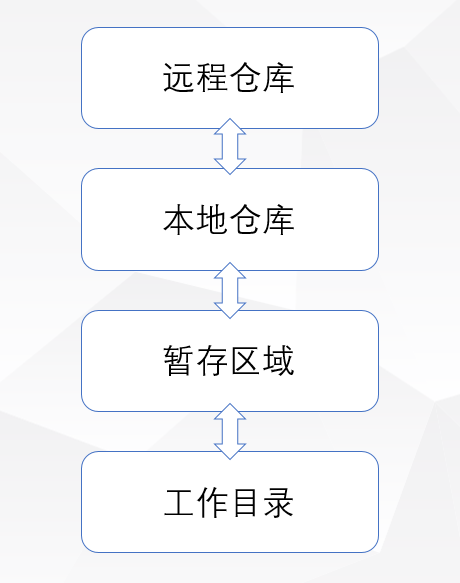
[root@node1 git-test]# git status

On branch master

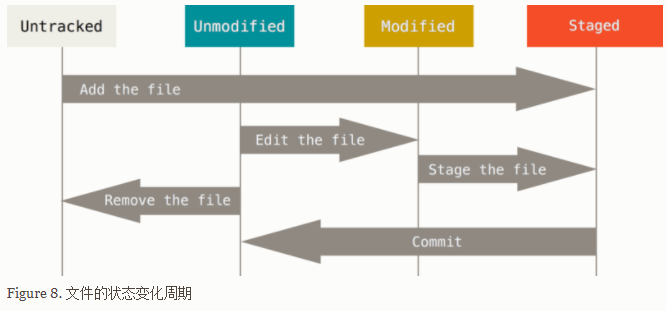
nothing to commit, working tree clean

#### 5）四个区域和文件的四种状态

四个区域：



文件的四种状态：



#### 6）查看已暂存和未暂存的修改

我们再次修改 README 文件后暂存，然后编辑 Hello.txt文件后先不暂存， 运行 status 命令将会看到：

[root@node1 git-test]# echo "three modify" >>README

[root@node1 git-test]# echo "first modify" >>Hello.txt

[root@node1 git-test]# git add README

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

modified: README

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: Hello.txt

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 git diff：

[root@node1 git-test]# git diff

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

index 802992c..6b599f0 100644

--- a/Hello.txt

+++ b/Hello.txt

@@ -1 +1,2 @@

Hello world

+first modify

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异， 也就是修改之后还没有暂存起来的变化内容。

要查看已暂存的将要添加到下次提交里的内容，可以用 git diff --cached 命令或Git 1.6.1 及更高版本还允许使用git diff --staged，效果是相同的。

[root@node1 git-test]# git diff --cached

diff --git a/README b/README

index 771c673..5f9ac91 100644

--- a/README

+++ b/README

@@ -1,3 +1,4 @@

My first git test project

first modify

second modify

+three modify

[root@node1 git-test]# git diff --staged

diff --git a/README b/README

index 771c673..5f9ac91 100644

--- a/README

+++ b/README

@@ -1,3 +1,4 @@

My first git test project

first modify

second modify

+three modify

现在我们将Hello.txt文件暂存，然后再进行编辑后，查看状态：

[root@node1 git-test]# git add Hello.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

modified: Hello.txt

modified: README

[root@node1 git-test]# git diff

[root@node1 git-test]# echo "second modify" >>Hello.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

modified: Hello.txt

modified: README

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: Hello.txt

[root@node1 git-test]# git diff

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

index 6b599f0..bfd0f21 100644

--- a/Hello.txt

+++ b/Hello.txt

@@ -1,2 +1,3 @@

Hello world

first modify

+second modify

[root@node1 git-test]# git diff --staged

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

index 802992c..6b599f0 100644

--- a/Hello.txt

+++ b/Hello.txt

@@ -1 +1,2 @@

Hello world

+first modify

diff --git a/README b/README

index 771c673..5f9ac91 100644

--- a/README

+++ b/README

@@ -1,3 +1,4 @@

My first git test project

first modify

second modify

+three modify

请注意，git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。 所以有时候你一下子暂存了所有更新过的文件后，运行 git diff 后却什么也没有，就是这个原因

#### 7）跳过使用暂存区

git commit 加上 -a 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤：

[root@node1 git-test]# echo "three modify" >> Hello.txt

[root@node1 git-test]# 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: Hello.txt

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

[root@node1 git-test]# git commit -a -m "jump git add submit"

[master 4794019] jump git add submit

1 file changed, 2 insertions(+)

[root@node1 git-test]# git status

On branch master

nothing to commit, working tree clean

#### 8、移除文件

我们先创建一个用于删除测试的文件a.txt,并提交

[root@node1 git-test]# echo "test" > a.txt

[root@node1 git-test]# git add a.txt

[root@node1 git-test]# git commit -m "rm test"

[master a80c6a9] rm test

1 file changed, 1 insertion(+)

create mode 100644 a.txt

现在我们在工作目录删除a.txt文件，然后查看状态,在 “Changes not staged for commit” 部分（也就是 未暂存清单）看到

[root@node1 git-test]# rm a.txt -f

[root@node1 git-test]# 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: a.txt

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

然后再运行 git rm 从暂存区删除文件：

[root@node1 git-test]# git rm a.txt

rm 'a.txt'

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

deleted: a.txt

[root@node1 git-test]# git commit -m "rm test finish"

[master 3a2db18] rm test finish

1 file changed, 1 deletion(-)

delete mode 100644 a.txt

[root@node1 git-test]# git status

On branch master

nothing to commit, working tree clean

另外一种情况是，我们想把文件从 Git 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。 换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。使用 --cached 选项

[root@node1 git-test]# git rm --cached a.txt

[root@node1 git-test]# git commit -m "delete a.txt from response"

#### 9、移动文件

使用git mv命令对文件进行改名操作：

使用mv命令改名：

[root@node1 git-test]# mv Hello.txt Helloword.txt

[root@node1 git-test]# 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: Hello.txt

Untracked files:

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

Helloword.txt

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

[root@node1 git-test]# git add Helloword.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

new file: Helloword.txt

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: Hello.txt

[root@node1 git-test]# git commit -m "mv test"

[master 9ccca93] mv test

1 file changed, 4 insertions(+)

create mode 100644 Helloword.txt

[root@node1 git-test]# 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: Hello.txt

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

[root@node1 git-test]# git rm Hello.txt

rm 'Hello.txt'

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

deleted: Hello.txt

[root@node1 git-test]# git commit -m "delete Hello.txt"

[master 2b1583e] delete Hello.txt

1 file changed, 4 deletions(-)

delete mode 100644 Hello.txt

You have new mail in /var/spool/mail/root

[root@node1 git-test]# git status

On branch master

nothing to commit, working tree clean

[root@node1 git-test]# ll

total 8

-rw-r--r-- 1 root root 52 Nov 28 11:17 Helloword.txt

-rw-r--r-- 1 root root 66 Nov 28 10:42 README

使用git mv 命令改名

[root@node1 git-test]# git mv Helloword.txt Hello.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

renamed: Helloword.txt -> Hello.txt

[root@node1 git-test]# git commit -m "git mv test"

[master 6597abf] git mv test

1 file changed, 0 insertions(+), 0 deletions(-)

rename Helloword.txt => Hello.txt (100%)

[root@node1 git-test]# git status

On branch master

nothing to commit, working tree clean

不管何种方式结果都一样。 两者唯一的区别是，mv 是一条命令而另一种方式需要三条命令，直接用 git mv 轻便得多

#### 10、忽略文件

一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。 通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。 在这种情况下，我们可以创建一个名为 .gitignore 的文件，列出要忽略的文件模式。

我们再看一个 .gitignore 文件的例子：

# no .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory

doc/\*\*/\*.pdf

### 3、查看提交历史

我们可以使用git log这个命令来查看提交的历史记录。

默认不用任何参数，git log 会按提交时间列出所有的更新，最近的更新排在最上面。这个命令会列出每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。

[root@node1 git-test]# git log

commit 6597abf6bb3617c2cbaad12cf1749d9771464ecd

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 12:03:00 2017 +0800

git mv test

commit 2b1583efcc6e18d0f80946f88d416b140b01367a

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 12:01:17 2017 +0800

delete Hello.txt

commit 9ccca93a00bd91a4883595b2f44a416ffb5ede9e

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 12:00:32 2017 +0800

mv test

commit 3a2db18768b228b3af6fa878e36544d15e35cfa4

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 11:44:28 2017 +0800

rm test finish

…………

常用的选项 -p，用来显示每次提交的内容差异。 也可以加上 -2 来仅显示最近两次提交：

[root@node1 git-test]# git log -p -2

commit 6597abf6bb3617c2cbaad12cf1749d9771464ecd

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 12:03:00 2017 +0800

git mv test

diff --git a/Helloword.txt b/Hello.txt

similarity index 100%

rename from Helloword.txt

rename to Hello.txt

commit 2b1583efcc6e18d0f80946f88d416b140b01367a

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 12:01:17 2017 +0800

delete Hello.txt

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

deleted file mode 100644

index 709f846..0000000

--- a/Hello.txt

+++ /dev/null

@@ -1,4 +0,0 @@

-Hello world

-first modify

-second modify

-three modify

--stat 选项在每次提交的下面列出所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。 在每次提交的最后还有一个总结。

[root@node1 git-test]# git log --stat -2

commit 6597abf6bb3617c2cbaad12cf1749d9771464ecd

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 12:03:00 2017 +0800

git mv test

Helloword.txt => Hello.txt | 0

1 file changed, 0 insertions(+), 0 deletions(-)

commit 2b1583efcc6e18d0f80946f88d416b140b01367a

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 12:01:17 2017 +0800

delete Hello.txt

Hello.txt | 4 ----

1 file changed, 4 deletions(-)

另外一个常用的选项是 --pretty。 这个选项可以指定使用不同于默认格式的方式展示提交历史。 这个选项有一些内建的子选项供你使用。 比如用 oneline 将每个提交放在一行显示，查看的提交数很大时非常有用。 另外还有 short，full 和 fuller 可以用，展示的信息或多或少有些不同，

[root@node1 git-test]# git log --pretty=oneline

6597abf6bb3617c2cbaad12cf1749d9771464ecd git mv test

2b1583efcc6e18d0f80946f88d416b140b01367a delete Hello.txt

9ccca93a00bd91a4883595b2f44a416ffb5ede9e mv test

3a2db18768b228b3af6fa878e36544d15e35cfa4 rm test finish

a80c6a946c0968edcbe3b9826629112b147a84e2 rm test

4794019de33bc2bfcea823ded27f80e98230e114 jump git add submit

b79795b531900217191da0db8b36ee220c074ad4 tree submit

a306d944c27e569f8003a82f7e251159cb61db33 second submit

5e874cc11b31065c65d4fd4ed7e6a275dca524d5 first submit

[root@node1 git-test]# git log --pretty=oneline -2

6597abf6bb3617c2cbaad12cf1749d9771464ecd git mv test

2b1583efcc6e18d0f80946f88d416b140b01367a delete Hello.txt

[root@node1 git-test]# git log --pretty=short -2

commit 6597abf6bb3617c2cbaad12cf1749d9771464ecd

Author: wendong <28152082@qq.com>

git mv test

commit 2b1583efcc6e18d0f80946f88d416b140b01367a

Author: wendong <28152082@qq.com>

delete Hello.txt

[root@node1 git-test]# git log --pretty=full -2

commit 6597abf6bb3617c2cbaad12cf1749d9771464ecd

Author: wendong <28152082@qq.com>

Commit: wendong <28152082@qq.com>

git mv test

commit 2b1583efcc6e18d0f80946f88d416b140b01367a

Author: wendong <28152082@qq.com>

Commit: wendong <28152082@qq.com>

delete Hello.txt

format，可以定制要显示的记录格式。

[root@node1 git-test]# git log --pretty=format:"%h - %an, %ar : %s"

6597abf - wendong, 4 hours ago : git mv test

2b1583e - wendong, 4 hours ago : delete Hello.txt

9ccca93 - wendong, 4 hours ago : mv test

3a2db18 - wendong, 4 hours ago : rm test finish

a80c6a9 - wendong, 4 hours ago : rm test

4794019 - wendong, 5 hours ago : jump git add submit

b79795b - wendong, 5 hours ago : tree submit

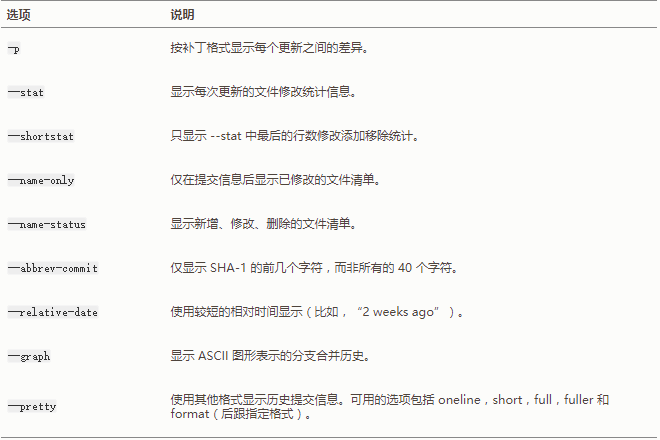
a306d94 - wendong, 6 hours ago : second submit

5e874cc - wendong, 7 hours ago : first submit

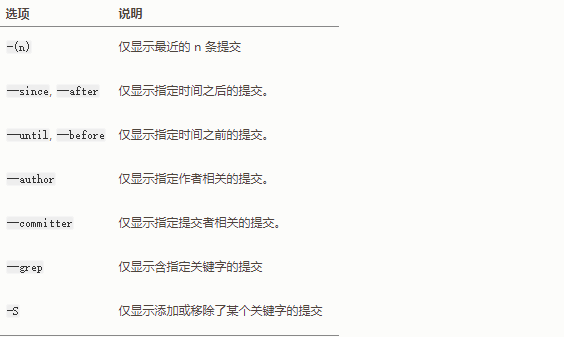
git log --pretty=format 常用的选项 列出了常用的格式占位符写法及其代表的意义



git log 的常用选项



限制输入长度的选项：



### 4、撤消修改

我们编辑Hello.txt文件，添加一行新的内容后保存，这时我们发现最后一行的内容添加有误，我们需要删除它。这时我们可以选择手工删除刚才添加的最后一行，把文件恢复，我们也可以使用git status先查看状态：

[root@node1 git-test]# cat Hello.txt

Hello world

first modify

second modify

three modify

four modify

[root@node1 git-test]# echo "five modify" >>Hello.txt

[root@node1 git-test]# cat Hello.txt

Hello world

first modify

second modify

three modify

four modify

five modify

[root@node1 git-test]# 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: Hello.txt

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

你可能会发现，Git告诉你，git checkout –file可以删除工作区的修改:

[root@node1 git-test]# git checkout -- Hello.txt

[root@node1 git-test]# git status

On branch master

nothing to commit, working tree clean

You have new mail in /var/spool/mail/root

[root@node1 git-test]# cat Hello.txt

Hello world

first modify

second modify

three modify

four modify

我们再看另一种情况，

[root@node1 git-test]# echo "five modify" >>Hello.txt

[root@node1 git-test]# git add Hello.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

modified: Hello.txt

[root@node1 git-test]# echo "six modify" >>Hello.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

modified: Hello.txt

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: Hello.txt

[root@node1 git-test]# git checkout -- Hello.txt

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

modified: Hello.txt

[root@node1 git-test]# cat Hello.txt

Hello world

first modify

second modify

three modify

four modify

five modify

最后我们总结一下：

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。最后要注意git checkout –file命令中的--,这个很重要，如果没有—这个命令就是另外的作用了。我们在后面具体会何用。

现在我们把所有的修改都已经提交到了暂存区，还没有commit，这个时候我们发现刚才的修改有错误，那么我们可以使用git reset HEAD file把暂存区的修改撤销掉，重新放回工作区：

[root@node1 git-test]# git status

On branch master

Changes to be committed:

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

modified: Hello.txt

You have new mail in /var/spool/mail/root

[root@node1 git-test]# git reset HEAD Hello.txt

Unstaged changes after reset:

M Hello.txt

[root@node1 git-test]# 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: Hello.txt

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

[root@node1 git-test]#

我们将所有的修改都commit,这个时候如果发现错误，想回退到刚才提交之前的版本该怎么处理？

首先我们通过git log命令查看我们的提交历史：

[root@node1 git-test]# git log --pretty=oneline

6cfd50cb9f0edec3fc59dd59845b4c6e3b5ec22a git reset HEAD test

8d9eb9afd8c38821cc84ffe47fe2261386f48a4f aaa

6597abf6bb3617c2cbaad12cf1749d9771464ecd git mv test

2b1583efcc6e18d0f80946f88d416b140b01367a delete Hello.txt

9ccca93a00bd91a4883595b2f44a416ffb5ede9e mv test

3a2db18768b228b3af6fa878e36544d15e35cfa4 rm test finish

a80c6a946c0968edcbe3b9826629112b147a84e2 rm test

4794019de33bc2bfcea823ded27f80e98230e114 jump git add submit

b79795b531900217191da0db8b36ee220c074ad4 tree submit

a306d944c27e569f8003a82f7e251159cb61db33 second submit

5e874cc11b31065c65d4fd4ed7e6a275dca524d5 first submit

我们使用git reset –hard b7979回到第三次提交的版本：

[root@node1 git-test]# git reset --hard b79795

[root@node1 git-test]# git log --pretty=oneline

b79795b531900217191da0db8b36ee220c074ad4 tree submit

a306d944c27e569f8003a82f7e251159cb61db33 second submit

5e874cc11b31065c65d4fd4ed7e6a275dca524d5 first submit

[root@node1 git-test]# cat README

My first git test project

first modify

second modify

three modify

我们可以使用git reset –hard commit-id返回指定的版本，但是返回后git log里面就只有当前提交之前的提交记录。

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从指向当前的版本改为指向你要回退的版本，然后顺便把工作区的文件更新。

小结：

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。

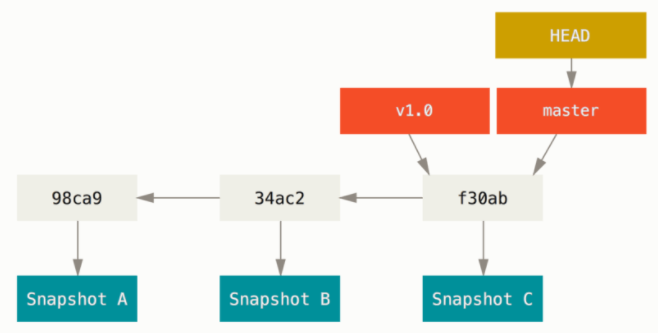
场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

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，使用git reset --hard commit-id回到指定的版本，不过前提是没有推送到远程库。

## 三、分支管理

几乎所有的版本控制系统都以某种形式支持分支。 使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线。 在很多版本控制系统中，这是一个略微低效的过程——常常需要完全创建一个源代码目录的副本。对于大项目来说，这样的过程会耗费很多时间。

有人把 Git 的分支模型称为它的`‘必杀技特性’'，也正因为这一特性，使得 Git 从众多版本控制系统中脱颖而出。 为何 Git 的分支模型如此出众呢？ Git 处理分支的方式可谓是难以置信的轻量，创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是一样便捷。 与许多其它版本控制系统不同，Git 鼓励在工作流程中频繁地使用分支与合并，哪怕一天之内进行许多次。 理解和精通这一特性，你便会意识到 Git 是如此的强大而又独特，并且从此真正改变你的开发方式。



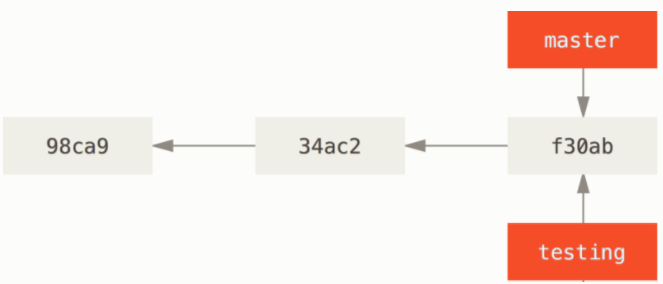
Git 的分支，其实本质上仅仅是指向提交对象的可变指针。 Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 它会在每次的提交操作中自动向前移动。（Git 的 “master” 分支并不是一个特殊分支。 它就跟其它分支完全没有区别。 之所以几乎每一个仓库都有 master 分支，是因为 git init 命令默认创建它，并且大多数人都懒得去改动它。）

### 1、分支的创建

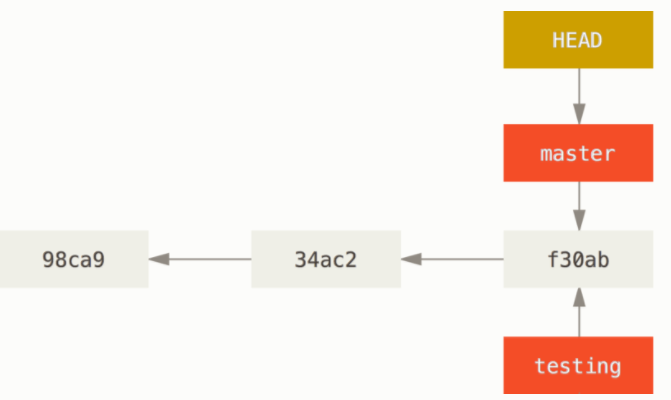
Git的分支创建很简单，它只是为你创建了一个可以移动的新指针。比如，创建一个testing分支，这会在当前所在的提交上创建一个指针。

[root@node1 git-test]# git branch testing

此时两个分支指向相同的提交。



那么问题来了，Git 又是怎么知道当前在哪一个分支上呢？ 也很简单，它有一个名为 HEAD 的特殊指针。 在 Git 中，它是一个指针，指向当前所在的本地分支（译注：将 HEAD 想象为当前分支的别名）。 在本例中，你仍然在 master 分支上。 因为 git branch 命令仅仅 创建 一个新分支，并不会自动切换到新分支中去。相当于下图：



[root@node1 git-test]# git branch

\* master

testing

git branch命令列出所有的分支，标\*号的表示当前分支。

[root@node1 git-test]# git log --oneline --decorate

b79795b (HEAD -> master, testing) tree submit

a306d94 second submit

5e874cc first submit

使用 git log --oneline --decorate 命令查看各个分支当前所指的对象

### 2、分支切换

使用git checkout 命令进行分支切换（注意与前面文件撤消操作的不同），现在切换分支到我们刚才创建的testing分支：

[root@node1 git-test]# git checkout testing

Switched to branch 'testing'

[root@node1 git-test]# git branch

master

\* testing

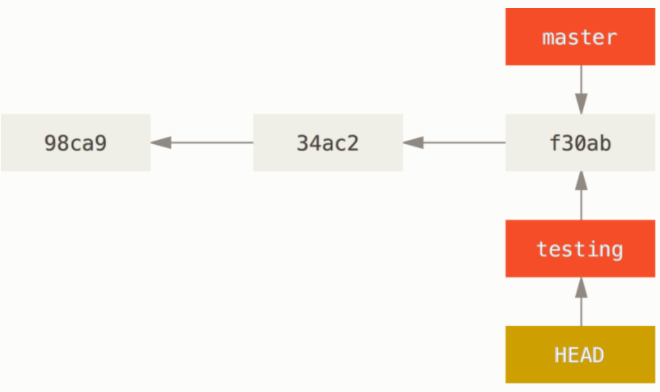
[root@node1 git-test]# git log --oneline --decorate

b79795b (HEAD -> testing, master) tree submit

a306d94 second submit

5e874cc first submit

现在HEAD 就指向 testing 分支



我们修改一下README文件，然后提交，再次查看分支的情况：

[root@node1 git-test]# echo "test branch one" >>README

[root@node1 git-test]# git commit -a -m "test branch"

[testing c32a494] test branch

1 file changed, 1 insertion(+)

[root@node1 git-test]# git branch

master

\* testing

[root@node1 git-test]# git log --oneline --decorate

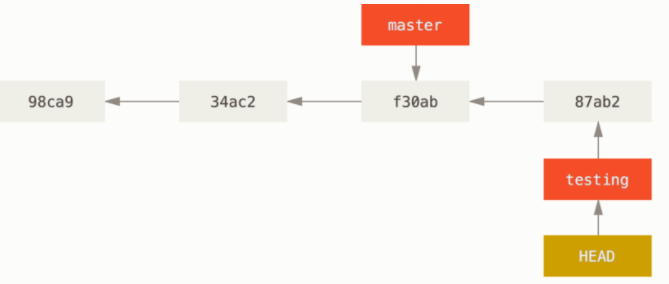
c32a494 (HEAD -> testing) test branch

b79795b (master) tree submit

a306d94 second submit

5e874cc first submit

我们可以看到testing 分支向前移动了，但是 master 分支却没有，它仍然指向运行 git checkout 时所指的对象（如下图）



现在我们将分支切换回到master，看看是什么情况：

[root@node1 git-test]# git checkout master

Switched to branch 'master'

[root@node1 git-test]# git branch

\* master

testing

[root@node1 git-test]# git log --oneline --decorate

b79795b (HEAD -> master) tree submit

a306d94 second submit

5e874cc first submit

[root@node1 git-test]# cat README

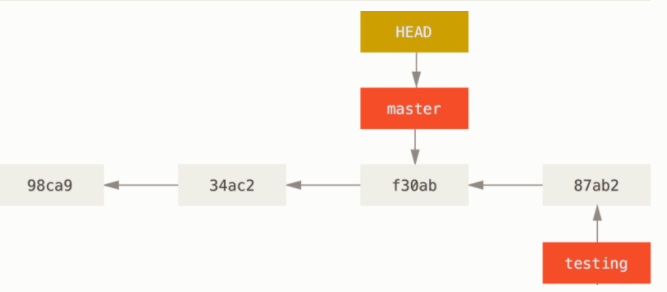
My first git test project

first modify

second modify

three modify

我们可以看到：一是使 HEAD 指回 master 分支，二是将工作目录恢复成 master 分支所指向的快照内容。（如下图）



我们在此基础上再次对工作区进行修改，这次我们新创建一个文件a.txt,并提交，查看：

[root@node1 git-test]# echo "branch checkout" >a.txt

[root@node1 git-test]# git add a.txt

[root@node1 git-test]# git commit -m "master branch submit"

[master fa337ff] master branch submit

1 file changed, 1 insertion(+)

create mode 100644 a.txt

[root@node1 git-test]# git branch

\* master

testing

[root@node1 git-test]# git log --oneline --decorate

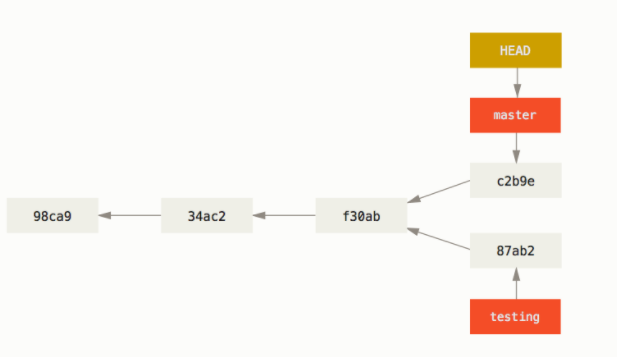
fa337ff (HEAD -> master) master branch submit

b79795b tree submit

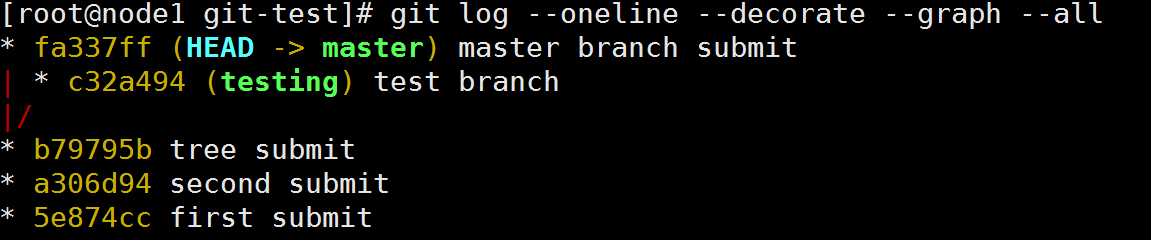
a306d94 second submit

5e874cc first submit

现在，这个项目的提交历史已经产生了分叉。 因为刚才创建了testing新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。 上述两次改动针对的是不同分支。情况如下图所示：



使用 git log 命令查看分叉历史。 运行 git log --oneline --decorate --graph --all ，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况。



### 3、分支的合并

#### 1）无冲突合并

到目前为止，我们已经有两个分支master,testing，根据实际情况，我们需要把testing分支上所做的更改合并到master分支上。可以使用git merae命令

[root@node1 git-test]# git merge testing

[root@node1 git-test]# cat README

My first git test project

first modify

second modify

three modify

test branch one

[root@node1 git-test]# git log --oneline --decorate

791f6d2 (HEAD -> master) Merge branch 'testing'

fa337ff master branch submit

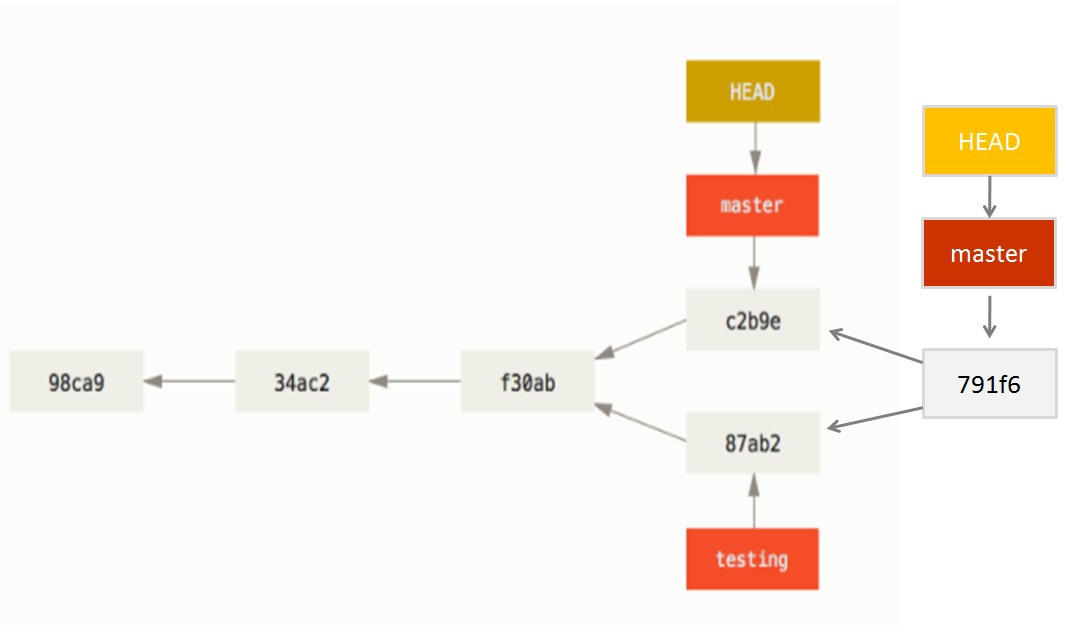
c32a494 (testing) test branch

b79795b tree submit

a306d94 second submit

5e874cc first submit

合并完成后，我们发现testing分支的内容（文件变更的内容，提交记录）已经合并到master分支中，并且生成了一次新的提交。如下图:



#### 2）分支合并冲突

我们分别在master和testing分支对Hello.txt做修改并提交，然后合并testing分支到master上：

在master分析上修改Hello.txt文件，并提交

[root@node1 git-test]# git branch

\* master

testing

[root@node1 git-test]# echo "aaa" >>Hello.txt

[root@node1 git-test]# git commit -a -m "master modify Hello.txt"

[master 0efdc93] master modify Hello.txt

1 file changed, 1 insertion(+)

[root@node1 git-test]# git log --oneline --decorate

0efdc93 (HEAD -> master) master modify Hello.txt

791f6d2 Merge branch 'testing'

fa337ff master branch submit

c32a494 (testing) test branch

b79795b tree submit

a306d94 second submit

5e874cc first submit

在testing分析上修改Hello.txt文件，并提交

[root@node1 git-test]# git checkout testing

Switched to branch 'testing'

[root@node1 git-test]# git branch

master

\* testing

[root@node1 git-test]# cat Hello.txt

Hello world

first modify

[root@node1 git-test]# echo "bbb" >> Hello.txt

[root@node1 git-test]# git commit -a -m "testing modify Hello.txt"

[testing bb73576] testing modify Hello.txt

1 file changed, 1 insertion(+)

[root@node1 git-test]# git log --oneline --decorate

bb73576 (HEAD -> testing) testing modify Hello.txt

c32a494 test branch

b79795b tree submit

a306d94 second submit

5e874cc first submit

切换回到master分析，并合并testing分支

[root@node1 git-test]# git checkout master

Switched to branch 'master'

[root@node1 git-test]# git branch

\* master

testing

[root@node1 git-test]# git merge testing

Auto-merging Hello.txt

CONFLICT (content): Merge conflict in Hello.txt

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

此时 Git 做了合并，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，等待你去解决合并产生的冲突。 你可以在合并冲突后的任意时刻使用 git status 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

[root@node1 git-test]# git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: Hello.txt

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

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。 Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。

冲突的文件会包含一些特殊的区段，如下所示：

[root@node1 git-test]# cat Hello.txt

Hello world

first modify

<<<<<<< HEAD

aaa

=======

bbb

>>>>>>> testing

这表示 HEAD 所指示的版本（也就是你的 master 分支所在的位置，因为你在运行 merge 命令的时候已经检出到了这个分支）在这个区段的上半部分（======= 的上半部分），而 testing分支所指示的版本在 ======= 的下半部分。为了解决冲突，你必须选择使用由 ======= 分割的两部分中的一个，或者你也可以自行合并这些内容。

我们修改Hello.txt文件并删除那些特殊标准，使用git add暂存文件，然后再使用git stauts命令查看状态：

[root@node1 git-test]# git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: Hello.txt

确定之前有冲突的文件都已经暂存后，我们使用git commit来完成合并。

[root@node1 git-test]# git commit -m "merge testing"

[master 6b0f755] merge testing

[root@node1 git-test]# git log --oneline --decorate

6b0f755 (HEAD -> master) merge testing

bb73576 (testing) testing modify Hello.txt

0efdc93 master modify Hello.txt

791f6d2 Merge branch 'testing'

fa337ff master branch submit

c32a494 test branch

b79795b tree submit

a306d94 second submit

5e874cc first submit

### 4、分支管理

git branch命令不带任何参数运行，可以得到当前所有分支的一个列表：

[root@node1 git-test]# git branch

\* master

testing

git branch -v 命令查看每一个分支的最后一次提交：

[root@node1 git-test]# git branch -v

\* master 9c612dc megre ccc

testing 6245c1e ccc testing

git branch –d 命令可以删除一个已经完成合并或者空的分支：

[root@node1 git-test]# git branch -d testing

Deleted branch testing (was 6245c1e).

[root@node1 git-test]# git log --oneline --decorate

9c612dc (HEAD -> master) megre ccc

6245c1e ccc testing

4f97f47 ccc master

6b0f755 merge testing

bb73576 testing modify Hello.txt

0efdc93 master modify Hello.txt

791f6d2 Merge branch 'testing'

fa337ff master branch submit

c32a494 test branch

b79795b tree submit

a306d94 second submit

5e874cc first submit

git branch –D 删除未合并的分支：

[root@node1 git-test]# git branch -d testing

error: The branch 'testing' is not fully merged.

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

[root@node1 git-test]# git branch -D testing

Deleted branch testing (was f057cdd).

## 四、标签管理

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。实际上，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。大家可以思考一下，Git已经有了commit，我们为什么还要标签呢？

### 1、创建标签

在Git上创建标签非常简单，首先我们切换到需要打标签的分支上，使用git tag <label-name>

[root@node1 git-test]# git branch

\* master

[root@node1 git-test]# git tag v1.0

使用git tag命令可以查看所有标签：

[root@node1 git-test]# git tag

v1.0

标签默认是打在最新的commit上，如果想要在历史commit上打标签可使用如下方式：

使用git log命令查看commit历史

[root@node1 git-test]# git log --oneline

6245c1e ccc testing

bb73576 testing modify Hello.txt

c32a494 test branch

b79795b tree submit

a306d94 second submit

5e874cc first submit

在第三次commit上打标签

[root@node1 git-test]# git tag v0.9 b79795b

[root@node1 git-test]# git tag

v0.9

v1.0

查看标签的信息，我们可以看到这是我们的第三次commit的信息

[root@node1 git-test]# git show v0.9

commit b79795b531900217191da0db8b36ee220c074ad4

Author: wendong <28152082@qq.com>

Date: Tue Nov 28 11:18:11 2017 +0800

tree submit

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

index 802992c..6b599f0 100644

--- a/Hello.txt

+++ b/Hello.txt

@@ -1 +1,2 @@

Hello world

+first modify

diff --git a/README b/README

index 771c673..5f9ac91 100644

--- a/README

+++ b/README

@@ -1,3 +1,4 @@

My first git test project

first modify

second modify

+three modify

### 2、使用标签

如果标签创建错误，我们可以使用-d删除一个标签

[root@node1 git-test]# git tag -d v0.9

Deleted tag 'v0.9' (was b79795b)

[root@node1 git-test]# git tag

v1.0

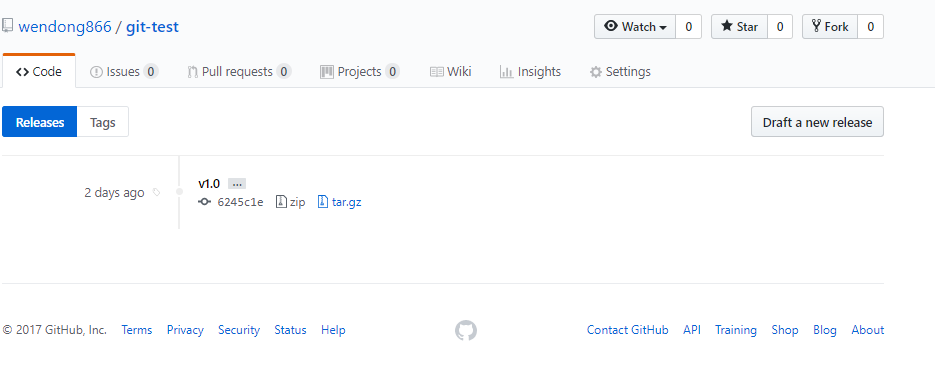
我们也可以把标签推送到远程服务器：

[root@node1 git-test]# git push origin v1.0

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

To github.com:wendong866/git-test.git

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



如果有多个标签要推送到远端，我们可以使用git push origin –tags

如果标签已经推送到了远端，要删除远程标签可以如下操作：

首先删除本地的标签

[root@node1 git-test]# git tag -d v1.0

Deleted tag 'v1.0' (was 6245c1e)

再删除远程标签

[root@node1 git-test]# git push origin :refs/tags/v1.0

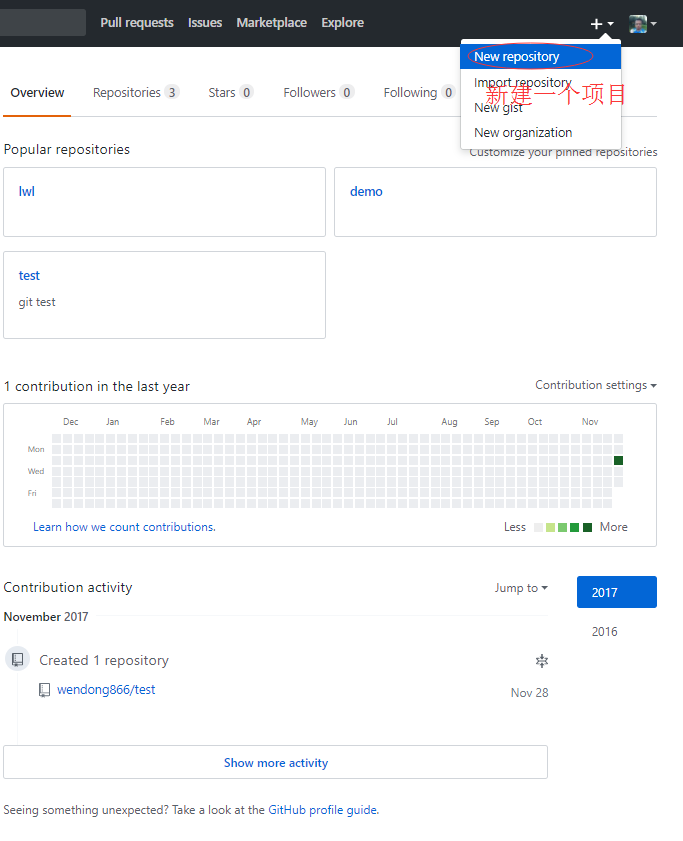
To github.com:wendong866/git-test.git

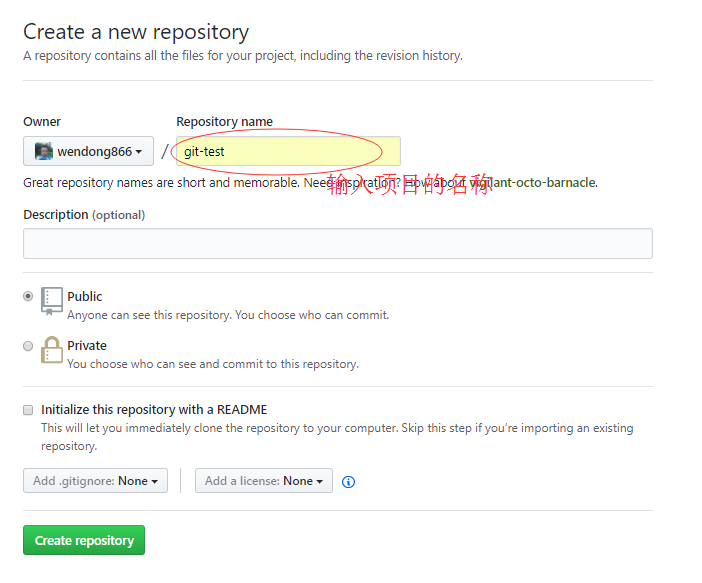
- [deleted] v1.0

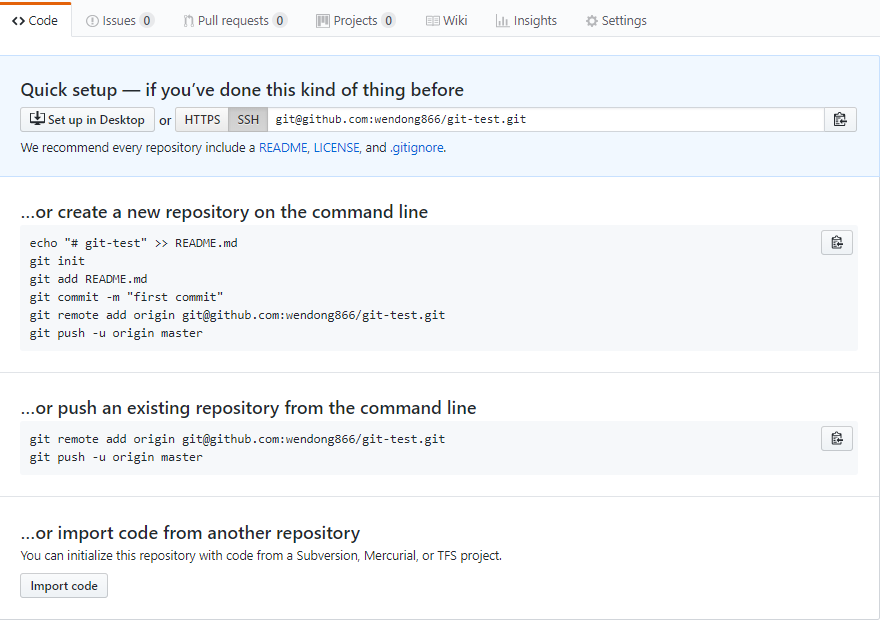
## 五、Git远程管理及github的使用

前面我们讲述的所有操作都是针对本地仓库，如果想要分享你的成果或者与他人协同使用仓库，我们需要创建远程仓库，将的代码推送到远程仓库，远程仓库既可以作为备份，又可以让其他人通过该仓库来协作。常用的远程仓库有两种，一是GitHub，这是一个发布在互联网的远程仓库，现在已经成为了每一个互联网丛业的必备。另外一个是Gitlab，我们在下一节课将详细讲述。本次课我们主要讲述github的使用。

### 1、在githab上创建项目



创建



创建完成git-test仓库后，这个仓库还是空的，最后这个页面告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个己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。

### 2、在本地git-test仓库上添加远程仓库

我们可以使用git remote add 命令为本地仓库添加远程仓库

[root@node1 git-test]# git remote add origin git@github.com:wendong866/git-test.git

[root@node1 git-test]# git remote -v

origin git@github.com:wendong866/git-test.git (fetch)

origin git@github.com:wendong866/git-test.git (push)

我们可以为一个本地仓库配置多个远程仓库。

#### 1、配置SSH-Key

为了能够在本机和你的github仓库之间推送数据，你配置配置ssh-key，首先在你的本机上生成ssh-key：

[root@node1 .ssh]# ssh-keygen -t rsa

Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /root/.ssh/id\_rsa.

Your public key has been saved in /root/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:XbxgUYdvfOd0NUzGjc2ZqyvUdn6O120z48d7q/QVJbU root@node1

The key's randomart image is:

+---[RSA 2048]----+

| ....=\*=|

| o..oB\*|

| o oo.E=|

| o o .+o\*|

| S . o..=o|

| . + .o|

| . ..+.+|

| ....\*@|

| ..+O@|

+----[SHA256]-----+

[root@node1 .ssh]#

[root@node1 .ssh]# ll

total 12

-rw------- 1 root root 1679 Nov 30 22:35 id\_rsa

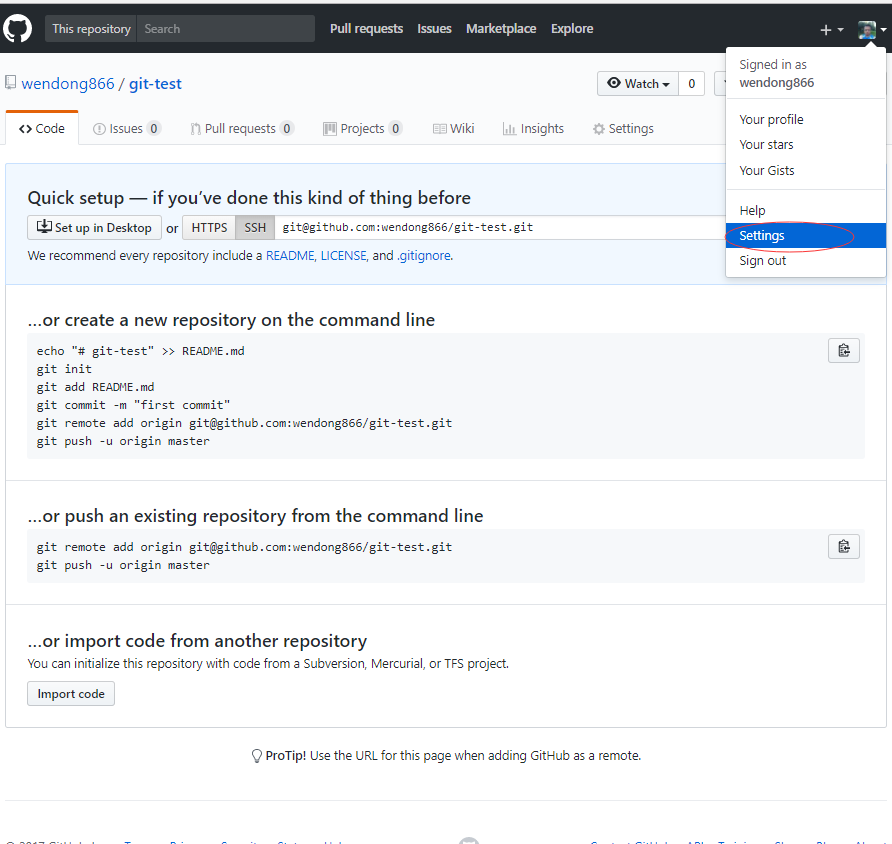
-rw-r--r-- 1 root root 392 Nov 30 22:35 id\_rsa.pub

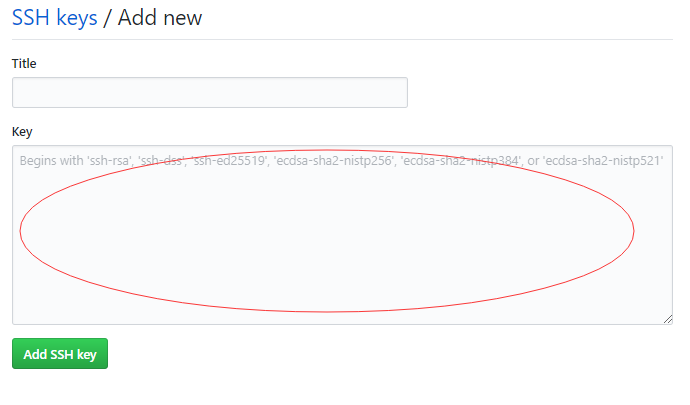
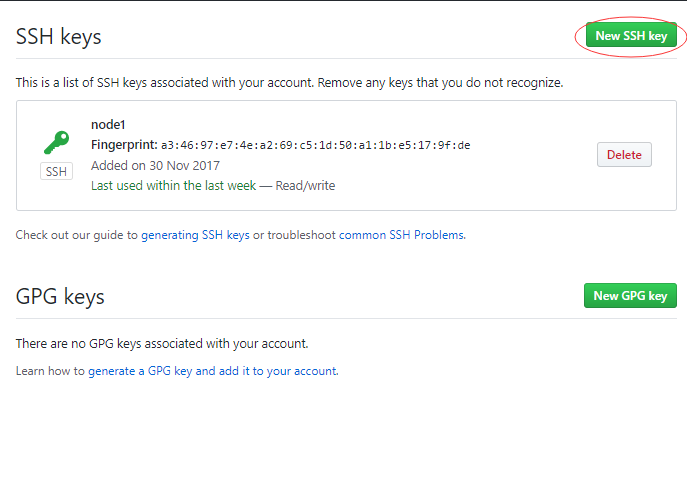
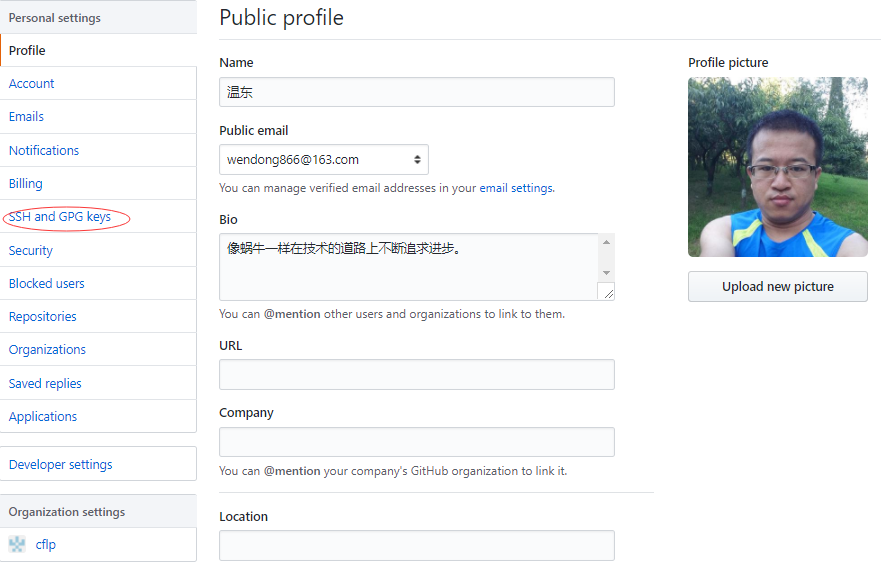
-rw-r--r-- 1 root root 407 Nov 28 07:37 known\_hosts

[root@node1 .ssh]# cat id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDAdQzT60jx7ORYUqo6wDa9hlE1t0fBMLBtRz8XcpAFJLRfJcfKx9EsWlN0+eJcvSoDCXOazMib0g/1COZEnDGIP0s/TcG5vgJvMFoMvQpxsZZeTgwbXdAkksPQu6/IKWeB60n8IMoomhT7xIQj9gVTkCFO9SpPnVReCCj0lTdTagp7kyeRt0CvVfnRLR15fCwYS1j1j3vdTx/lhQZiISbdIg1J08KDybzpCrPITv3Bl8s67BTCQFShqW9mI5YZgns37KCudPvXyn/LQgxVP++xCcDcoOt0DCFNBrcFdhAgiKhKOvNpTtW/Ydi8YTe5nyoUACXzlv1pdjmL5S4y65PR root@node1

然后将你的公钥添加到你的github的ssh-key中：





可以通过下面的命令验证是否成功：

[root@node1 git-test]# ssh -T git@github.com

Hi wendong866! You've successfully authenticated, but GitHub does not provide shell access.

#### 2、推送本地仓库到远程GitHub仓库

添加完ssh-key后，就可以将本地仓库的内容推送到远程仓库了。我们可以使用git push命令。

[root@node1 git-test]# git push -u origin master

Warning: Permanently added the RSA host key for IP address '192.30.255.113' to the list of known hosts.

Counting objects: 20, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (14/14), done.

Writing objects: 100% (20/20), 1.57 KiB | 0 bytes/s, done.

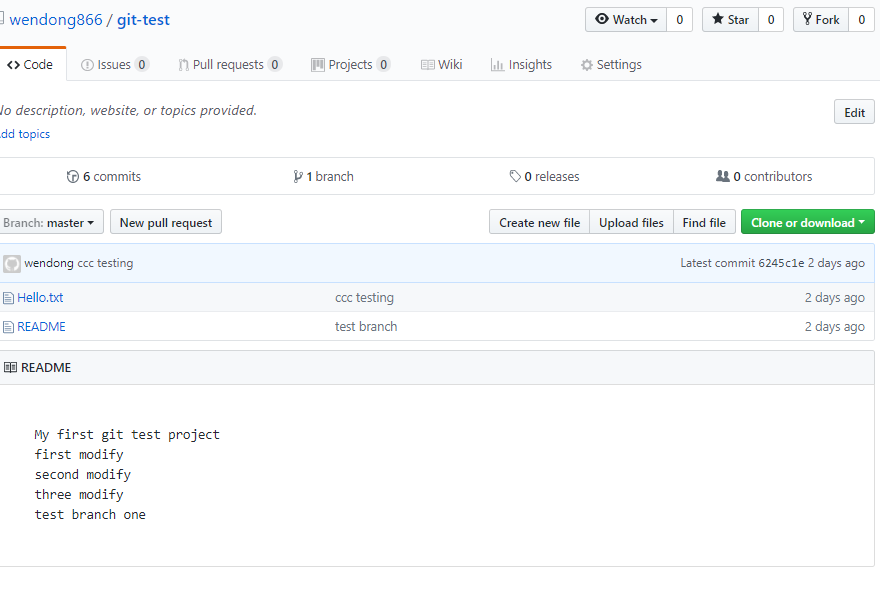
Total 20 (delta 2), reused 0 (delta 0)

remote: Resolving deltas: 100% (2/2), done.

To github.com:wendong866/git-test.git

\* [new branch] master -> master

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



我们已经可以在github仓库看到推送上去的内容了。

我们也可以把本地的其他分支推送到远程仓库上：

[root@node1 git-test]# git push origin dev

Counting objects: 3, done.

Delta compression using up to 2 threads.

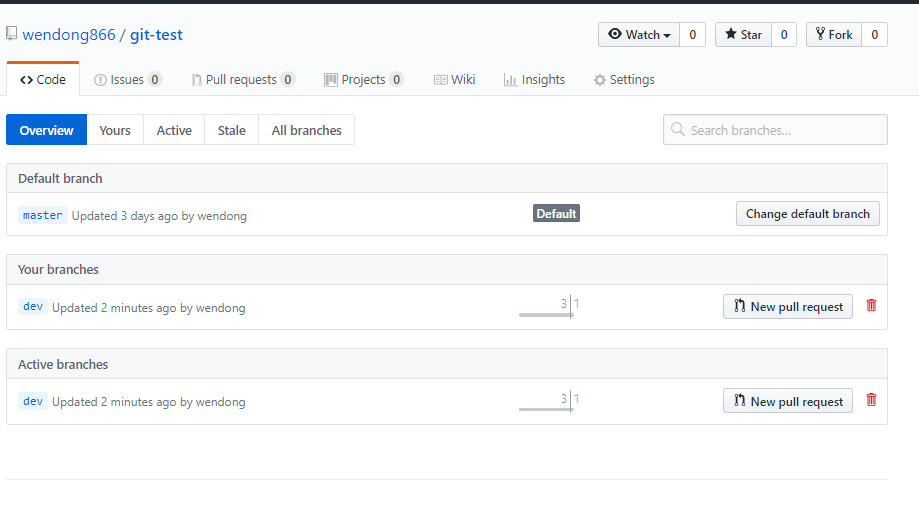
Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 298 bytes | 0 bytes/s, done.

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

To github.com:wendong866/git-test.git

\* [new branch] dev -> dev

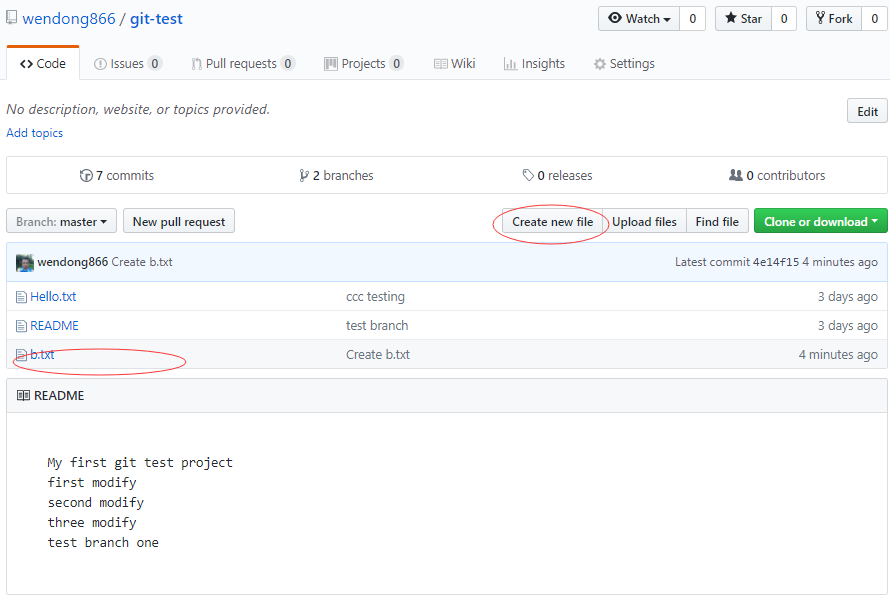


### 3、提取远程仓库

Git有两种方式用来提取远程仓库的更新：

1. 先使用git fetch从远程仓库获取最新版本到本地（不会自动merge），然后再使用git merge命令合并。

我们首先在githab上的git-test项目上创建一个新文件b.txt



[root@node1 git-test]# git fetch origin

remote: Counting objects: 3, done.

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

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

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

From github.com:wendong866/git-test

6245c1e..4e14f15 master -> origin/master

[root@node1 git-test]# git merge origin/master

Updating 6245c1e..4e14f15

Fast-forward

b.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 b.txt

[root@node1 git-test]# ll

total 16

-rw-r--r-- 1 root root 16 Dec 2 19:43 b.txt

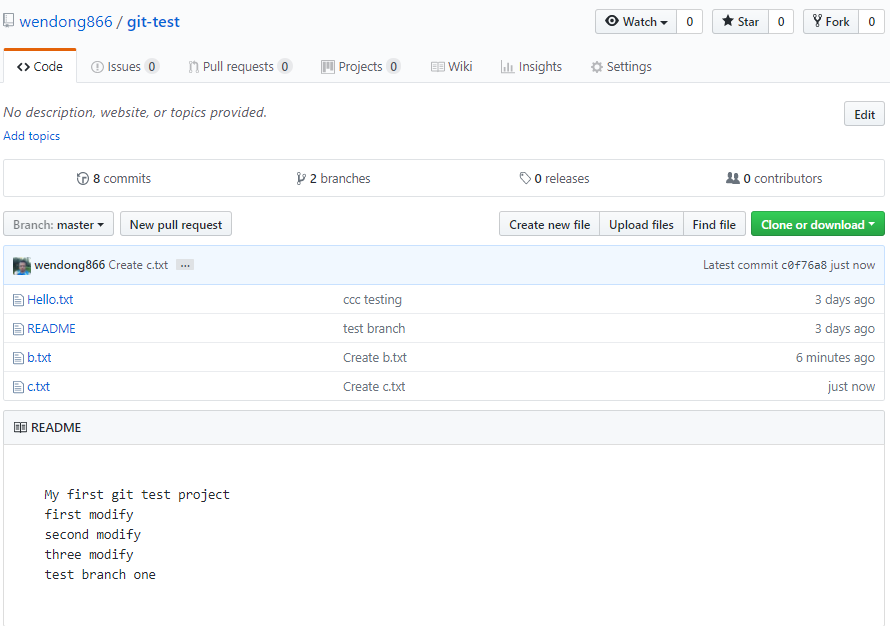
-rw-r--r-- 1 root root 33 Dec 2 19:42 Hello.txt

-rw-r--r-- 1 root root 70 Nov 29 22:45 Hello.txt.orig

-rw-r--r-- 1 root root 82 Dec 2 19:42 README

1. 我们也可以使用git pull命令直接获取远程最新版本，并自动合并到本地，相当第一种的两条命令。

同样我们在githab上创建c.txt文件：



[root@node1 git-test]# git pull

remote: Counting objects: 3, done.

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

remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0

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

From github.com:wendong866/git-test

4e14f15..c0f76a8 master -> origin/master

Updating 4e14f15..c0f76a8

Fast-forward

c.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 c.txt

[root@node1 git-test]# ll

total 20

-rw-r--r-- 1 root root 16 Dec 2 19:43 b.txt

-rw-r--r-- 1 root root 15 Dec 2 19:48 c.txt

-rw-r--r-- 1 root root 33 Dec 2 19:42 Hello.txt

-rw-r--r-- 1 root root 70 Nov 29 22:45 Hello.txt.orig

-rw-r--r-- 1 root root 82 Dec 2 19:42 README

[root@node1 git-test]# git log --oneline --graph

\* c0f76a8 Create c.txt

\* 4e14f15 Create b.txt

\* 6245c1e ccc testing

\* bb73576 testing modify Hello.txt

\* c32a494 test branch

\* b79795b tree submit

\* a306d94 second submit

\* 5e874cc first submit

[root@node1 git-test]#