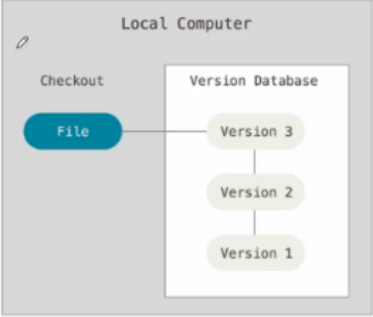
Git操作文档

1. 版本控制说明

版本控制用于实现追踪文件的变化、修改历史，回退到历史版本。版本控制的实现分为如下三种，本地版本控制、集中化版本控制以及分布式版本控制。

## 本地版本控制

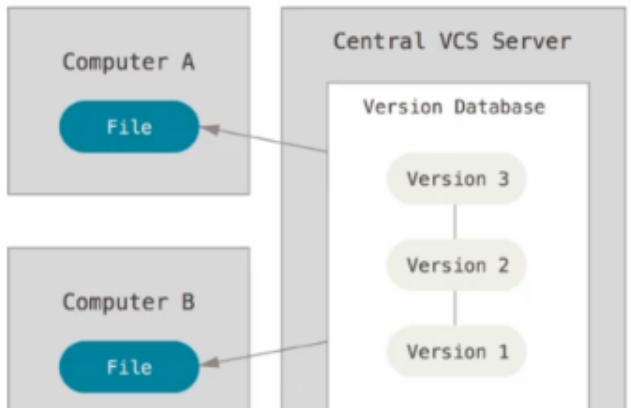
本地版本控制系统仅在本地进行版本控制，缺点是无法用于多人共享。



## 集中化版本控制

集中式版本控制系统(CVCS，Central Version Control System)，比如SVN版本控制系统。集中式版本控制系统的版本库是集中存放在中央服务器的，通常先在本地从中央服务器取得最新的版本，然后进行开发操作，完成后提交给中央服务器。

集中式版本控制系统缺点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，这样的使用体验就不好了。



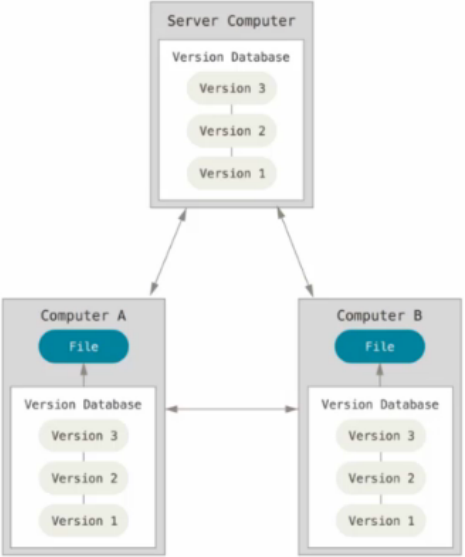
## 分布式版本控制

分布式版本控制系统(DVCS，Distribute Version Control System)，比如Git分布式版本控制系统。

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，在你操作时不需要进行联网，因为版本库在本地。而且分布式版本控制系统的安全性更高，因为某个人的本地库异常无法使用，可以重其他人那里复制一个版本库。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法进行更新同步了。

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。

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。



## 命令清单

|  |  |
| --- | --- |
| 命令 | 说明 |
| git config --global user.name "name" | 全局配置用户名称 |
| git config --global user.email "@xx" | 全局配置用户邮箱 |
| git init | 初始化本地仓库 |
| git add <file> | 将工作区下指定文件放入暂存区进行追踪 |
| git add <dir> | 将工作区下指定目录下所有文件进行追踪 |
| git add \* | 将当前目录下所有文件放入暂存区进行跟踪 |
| git commit –m “提交注释” | 将暂存区的文件进行提交到本地仓库 |
| git status | 查看工作区状态 |
| git diff filename | 对比暂存区和仓库文件的差异 |
| git log | 查看提交历史 |
| git reflog | 查看命令历史 |
| git reset --hard HEAD | commit\_id | 回退文件到指定的版本 |
| git reset HEAD filename | 将暂存区的修改回退到工作区 |
| git checkout -- filename | 撤销工作区的修改 |
| git rm filename | 删除版本库文件，需要提交才可生效 |
| git remote add origin <server> | 关联远程仓库 |
| git push –u origin master | 推送master分支到远程库 |
| git clone <server> | 克隆远程库 |
| git clone oldRepo/ newRepo/ | 克隆本地库 |
| git branch branch\_name | 创建branch\_name分支 |
| git checkout branch\_name | 切换到branch\_name分支 |
| git checkout –b branch\_name | 创建并切换到branch\_name分支 |
| git branch | 查看当前分支 |
| git merge branch\_name | 合并某个分支到当前分支 |
| git branch –d branch\_name | 删除分支branch\_name |
| git merge –no-ff | 新创建提交点合并 |
| git branch –D branch\_name | 删除未合并的分支 |
| git stash | 保存当前的工作区内容现状 |
| git stash list | 列举当前所有的工作区stash |
| git stash apply | 回复stage的工作区 |
| git stash drop | 删除指定的工作区stash |
| git stash pop | 恢复工作区stash同时删除 |
| git pull | 抓取分支内容 |
| git tag | 查看所有标签 |
| git tag tagName | 创建名称为tagName的标签 |
| git tag tagName versionId | 在指定的提交号上创建标签 |
| git tag -a <tagname> -m "ccc" | 指定标签信息 |
| git tag –d <tag-name> | 删除本地标签 |
| git push origin :refs/tags/tagname | 删除远程标签 |

1. Git的安装与操作

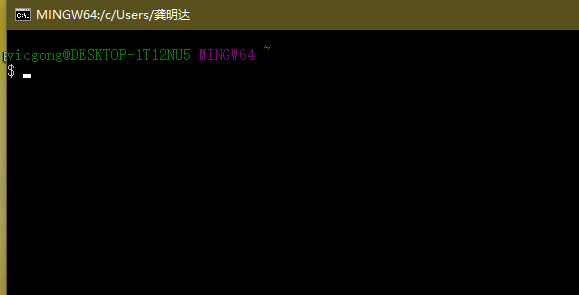
## 安装Git

Linux系统使用如下命令进行安装：

1. $ sudo yum install git
2. $ sudo apt-get install git

Windows系统安装文件下载并安装：<http://git-scm.com/download/win>

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！



## 配置config

Git软件安装后需要对其进行配置config信息， 由于Git是分布式版本控制系统，故而每台机器都必须自报家门：你的名字和Email地址。

设置git全局配置：

1. $ git congig --global user.name "vicgong"
2. $ git config --global user.email "vicgong@example.com"

注意：该命令使用“--global”选项，用于指定该机器上所有的Git仓库都使用该配置信息。当然也可以对某个仓库指定特殊的用户名和Email地址。

设置完成后可以获取git配置信息列表：

1. $ git config --list

## 初始化版本库

版本库又名仓库，英文名repository，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被“还原”。

创建并初始化版本库：

1. $ mkdir GitRepository
2. $ cd GitRepository
3. $ git init
4. # Initialized empty Git repository in E:/GitRepository/.git/

上述命令中初始化一个空的版本库(empty Git repository),同时创建了一个隐藏的“.git” 目录,该目录是Git用于跟踪管理版本库，切勿修改该目录下的文件，否则会破坏该版本库。

## 删除版本库

删除本地版本库就是删除本地版本库所在目录下隐藏的.git文件夹.

1. $ cd /e/GitRepository
2. $ rm -rf .git
3. Git简述

## Git简史

Git由Linux 托瓦茨发明，基于BitKeeper的经验设计，目的是用于管理Linux Kernel的源代码。2005年4月3日诞生，2005年4月6日第一版发布，2005年6月16日开始管理内核源代码。

Git的记录方式不同于SVN，SVN记录文件的方式时基于初始文件然后记录修改的部分，而Git则是记录每次修改后的快照版本。

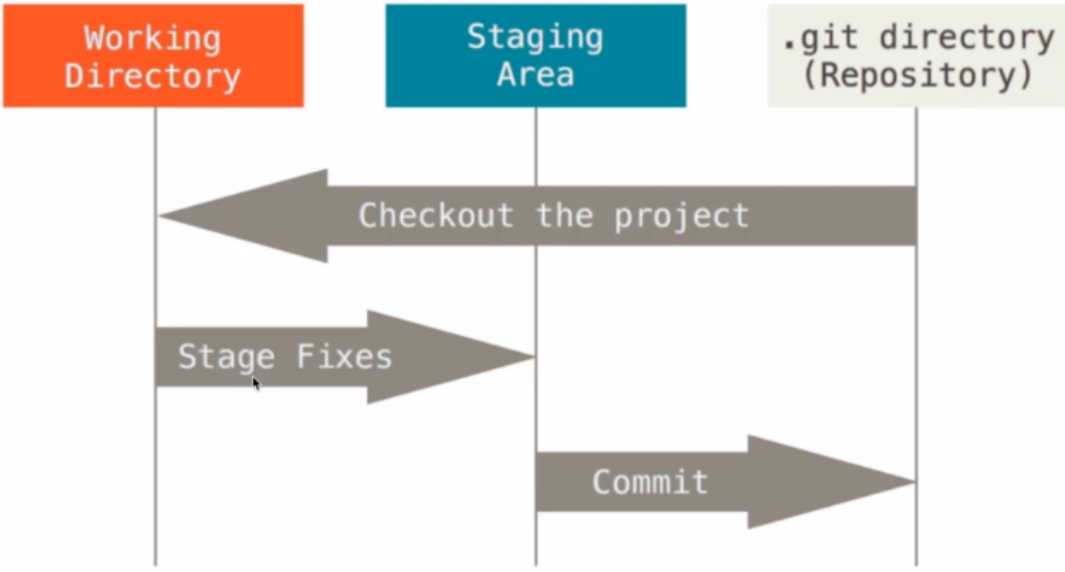
Git几乎所有的操作都是本地执行的，Git内部会对文件进行校验进而保证数据的完整性。Git一般只添加数据，Git中的文件只有三种状态：Commited(已提交)，Modified(已修改)，Staged(已暂存)。

## Git基本概念

* 远程仓库：指的是服务器上的版本库
* 本地仓库：指的是本地的版本库
* 工作区：指的是当前正在编辑或者是正在修改的工作目录
* 提交号：在Git中每做一次提交都会生成一个唯一的提交号
* 分支：指的是基于某个版本开设一个新的工作区进行控制
* 标签：通过标签对版本的进行标记，等同于建立一个标签名称的快照。

## Git工作区

创建Git版本库的本地目录称为工作区（work directory）。

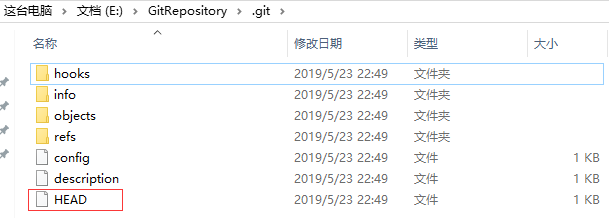


在工作区对文件进行修改后，可以通过添加命令添加到Staging Area暂存区，然后才能通过提交命令将暂存区的内容进行提交到版本库，同时生成一个提交号进行记录当前版本。当然也可以通过检出命令将版本库的内容检出到本地。

## 版本库

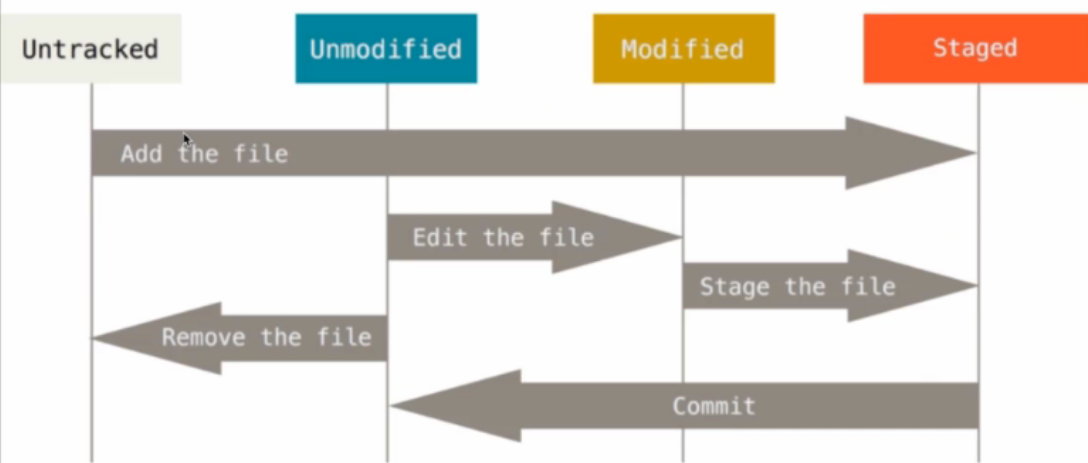
在Git工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库（Repository）。

## 暂存区



Git的版本库里存了很多东西，其中最重要的是stage（或者叫index）的暂存区，以及Git在创建版本库时自动创建的默认分支master，以及指向master的一个指针叫HEAD。

## 文件状态



在git管理的文件分为四种状态：Untracked、Staged、Modified、Unmodified。

* Untracked：文件未被追踪状态。
  + 工作区未添加到暂存区的文件，或者版本库中不存在的文件
  + 未被追踪状态文件添加到暂存区，文件状态变更为已暂存状态
* Staged:文件已暂存状态。
  + 将修改后的文件添加到暂存区，文件就为已暂存状态
* Modified:文件已修改状态。
  + 对文件进行修改后，文件状态变更为已修改状态
  + 将已修改的文件添加到暂存区，文件状态变更为已暂存状态
* Unmodified：文件未修改状态。
  + 该文件表示同版本库中一致
  + 将暂存区文件提交到版本库后，文件状态变更为未修改状态

1. Git文件管理

## 建立跟踪

对指定文件建立跟踪，可重复使用添加多个文件：

1. $ git add <file>

对工作区内的指定目录下所有的文件建立跟踪：

1. $ git add \*

对工作区内的当前目录下的所有文件建立跟踪：

1. $ git add <dir>

## 提交文件

对已跟踪的文件提交至版本库：

1. $ git commit -m "备注信息"

## Git提交原理

通常情况下将文件提交到版本库分为两步：

**第一步**：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（Stage）；

在工作区下创建文件readme.txt，文件内容如下：

1. This is vicgong using repository.
2. The repository only use study.

文件必须在工作区的目录下，否则Git无法找到该文件。然后将文件添加到暂存区。

1. $ git add readme.txt

比如：对文件追踪后如下图，文件添加到暂存区



**第二步**：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。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。

1. $ git commit -m "add init readme.txt"
2. [master def3d0d] add init readme.txt
3. 1 file changed, 2 insertions(+)
4. create mode 100644 readme.txt

git commit命令执行成功后会输出:

* 1 file changed：1个文件被改动（新添加的readme.txt文件）；
* 2 insertions：插入了两行内容（readme.txt有两行内容）。

因为创建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。

像这样，你可以不断对文件进行修改，然后不断提交修改到版本库里，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。

对暂存区的所有内容进行提交后，暂存区就没有任何内容了：



## 查看版本库状态

git status命令查看仓库当前的状态：

1. $ git status

修改readme.txt文件，改成如下内容：

1. This is vicgong using repository.
2. The repository only use study, no privacy involved.

使用git status查看版本库状态，命令输出readme.txt被修改过了，但还没有为提交而暂存。

1. $ git status
2. On branch master
3. Changes not staged **for** commit:
4. (use "git add <file>..." to update what will be committed)
5. (use "git checkout -- <file>..." to discard changes in working directory)
7. modified:   readme.txt
9. no changes added to commit (use "git add" and/or "git commit -a")

## Git查看日志

git log命令用于查看用户的提交日志：

1. $ git log

git reflog命令用于查看用户的命令执行日志

1. $ git reflog

在实际工作中使用git log命令显示从最近到最远的提交日志：

1. $ git log
2. commit def3d0dc1c4cbffad13541c1dcdc14887695fb84 (HEAD -> master)
3. Author: vicgong <964744855@qq.com>
4. Date:   Thu May 23 23:55:24 2019 +0800
6. add init readme.txt
8. commit 6510e472b1d74dcbc3cf408dde63069b817af298
9. Author: vicgong <964744855@qq.com>
10. Date:   Thu May 23 23:34:30 2019 +0800
12. init inport shell scripttool

git log使用--pretty=oneline参数设定日志为单行输出：

1. $ git log --pretty=oneline
2. def3d0dc1c4cbffad13541c1dcdc14887695fb84 (HEAD -> master) addinit readme.txt
3. 6510e472b1d74dcbc3cf408dde63069b817af298 init inport shell scripttool

友情提示：

* 类似“def3d0…”的是commit id（版本号），这和SVN的版本号不一样。
* Git的commit id非递增的数字，而是通过SHA1计算出来的十六进制数字。
* Git是分布式的版本控制系统，由多人共同使用，故每次提交的版本号应该唯一。
* 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。

假如对readme.txt进行两次修改，修改的备注为modify 1和modify2。然后进行add和commit。最后回退到modify1版本。

1. $ git log
2. commit 379417b7d58dcf27848d44877b3437041e44061a (HEAD -> master)
3. Author: vicgong <964744855@qq.com>
4. Date:   Sat May 25 15:03:56 2019 +0800
6. modify2
8. commit c44e77c231406f4fd565ff815f4573bda8fb7d8a
9. Author: vicgong <964744855@qq.com>
10. Date:   Sat May 25 15:03:01 2019 +0800
12. modify1
14. commit def3d0dc1c4cbffad13541c1dcdc14887695fb84
15. Author: vicgong <964744855@qq.com>
16. Date:   Thu May 23 23:55:24 2019 +0800
18. add init readme.txt
20. commit 6510e472b1d74dcbc3cf408dde63069b817af298
21. Author: vicgong <964744855@qq.com>
22. Date:   Thu May 23 23:34:30 2019 +0800
24. init inport shell scripttool
26. $ git reset --hard c44e77
27. HEAD is now at c44e77c modify1

这时候如果想切换版本到modify2，可是通过git log命令无法看到modify2的版本号了。

1. $ git log
2. commit c44e77c231406f4fd565ff815f4573bda8fb7d8a (HEAD -> master)
3. Author: vicgong <964744855@qq.com>
4. Date:   Sat May 25 15:03:01 2019 +0800
6. modify1
8. commit def3d0dc1c4cbffad13541c1dcdc14887695fb84
9. Author: vicgong <964744855@qq.com>
10. Date:   Thu May 23 23:55:24 2019 +0800
12. add init readme.txt
14. commit 6510e472b1d74dcbc3cf408dde63069b817af298
15. Author: vicgong <964744855@qq.com>
16. Date:   Thu May 23 23:34:30 2019 +0800
18. init inport shell scripttool

通过git reflog查看提交历史获取每次提交的版本号。

1. $ git reflog
2. c44e77c (HEAD -> master) HEAD@{0}: reset: moving to c44e77
3. 379417b HEAD@{1}: commit: modify2
4. c44e77c (HEAD -> master) HEAD@{2}: commit: modify1
5. def3d0d HEAD@{3}: commit: add init readme.txt
6. 6510e47 HEAD@{4}: commit (initial): init inport shell scripttool

## Git版本切换

git进行版本切换命令如下：

1. 回退到上一个版本
2. $ git reset --hard HEAD^
4. 回退到上上个版本
5. $ git reset --hard HEAD^^
7. 回退到往上100个版本
8. $ git reset --hard HEAD~100
10. 回退到指定版本
11. $ git reset --hard <commit\_id>

在Git中，用HEAD表示当前版本，也就是最新的提交号对应的版本，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。

在之前我们切换到了modify1的版本，现在我们切换到modify2的版本。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。

1. $ git reset --hard 379417b
2. HEAD is now at 379417b modify2

Git的版本切换的速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你切换版本的时候，Git仅仅是把HEAD从指向指定的版本, 然后顺便把工作区的文件更新。所以你让HEAD指向哪个版本号，git就把当前版本定位在哪。

1. ┌────┐
2. │HEAD│
3. └────┘
4. │
5. └──> ○ modify2
6. │
7. ○ modify1
8. │
9. ○ add init readme.txt

改为指向modify1：

1. ┌────┐
2. │HEAD│
3. └────┘
4. │
5. │    ○ modify2
6. │    │
7. └──> ○ modify1
8. │
9. ○ add init readme.txt

## 撤销暂存区修改

使用如下命令可以将add到暂存区的修改撤销到工作区为追踪前的状态。

1. $ git reset HEAD <file>

例如，修改readme.txt，同时添加到暂存区，同时git status查看，修改只是添加到了暂存区，还没有提交：

1. $ echo This is a temp modify. >> readme.txt
2. $ git add readme.txt
3. $ git status
4. On branch master
5. Changes to be committed:
6. (use "git reset HEAD <file>..." to unstage)
7. modified:   readme.txt

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

1. $ git reset HEAD readme.txt
2. Unstaged changes after reset:
3. M   readme.txt

再用git status查看一下，现在暂存区是干净的，工作区有修改：

1. $ git status
2. On branch master
3. Changes not staged **for** commit:
4. (use "git add <file>..." to update what will be committed)
5. (use "git checkout -- <file>..." to discard changes in working directory)
7. modified:   readme.txt
9. no changes added to commit (use "git add" and/or "git commit -a")

## 丢弃工作区修改

使用如下命令撤销工作区的修改：

1. $ git checkout -- <file>

有两种含义：

* 文件修改后还没有添加到暂存区，此时撤销修改意味着从版本库检出文件；
* 文件已经添加到暂存区然后发生了修改，此时撤销修改意味着从暂存区检出文件；

总之，将文件切换到到最近一次git commit或git add时的状态。命令中的“--”很重要，没有“--”，就变成了“切换到另一个分支”的命令。

例如，readme.txt文件处于工作区未添加到暂存区，撤销工作区的修改。

1. $ cat readme.txt
2. This is vicgong using repository.
3. The repository only use study, no privacy involved.
4. This is a first modify.
5. This is a second modify.
6. This is a temp modify.
8. $ git checkout -- readme.txt
10. $ cat readme.txt
11. This is vicgong using repository.
12. The repository only use study, no privacy involved.
13. This is a first modify.
14. This is a second modify.

例如，修改readme.txt文件并添加到暂存区。然后再次修改readme.txt，撤销工作区的修改。

1. $ echo This is a temp modify. >> readme.txt
2. $ git add readme.txt
3. $ echo This is a temp2 modify. >> readme.txt
4. $ cat readme.txt
5. This is vicgong using repository.
6. The repository only use study, no privacy involved.
7. This is a first modify.
8. This is a second modify.
9. This is a temp modify.
10. This is a temp2 modify.
12. $ git checkout -- readme.txt
13. $ cat readme.txt
14. This is vicgong using repository.
15. The repository only use study, no privacy involved.
16. This is a first modify.
17. This is a second modify.
18. This is a temp modify.

例如，先将readme.txt的修改撤销到modify2的版本

1. $ git reset HEAD readme.txt
2. Unstaged changes after reset:
3. M       readme.txt
5. $ git checkout -- readme.txt
6. $ git status
7. On branch master
8. nothing to commit, working tree clean

## 删除版本库文件

如下命令用于删除版本库文件：

1. $ git rm <file>
2. $ git commit –m "del <file>"

例如，添加文件test.txt并提交到版本库，然后再本地删除test.txt文件

1. $ echo "this is a test delete repository fiel" > test.txt
2. $ git add test.txt
3. $ git commit -m "test delete"
4. [master 5b585da] test delete
5. 1 file changed, 1 insertion(+)
6. create mode 100644 test.txt
8. $ rm -f test.txt

在Git中，删除也是一个修改操作，查看版本库状态时就会显示出那些本地文件被删除了。

1. $ git status
2. On branch master
3. Changes not staged **for** commit:
4. (use "git add/rm <file>..." to update what will be committed)
5. (use "git checkout -- <file>..." to discard changes in working directory)
7. deleted:    test.txt
9. no changes added to commit (use "git add" and/or "git commit -a")

如果删错了，此时可以通过撤销工作区修改的命令进行撤销。但是未提交到版本库或者暂存区的文件，删除后无法恢复。

1. $ git checkout -- test.txt

如果确实要将版本库的该文件删除，那就用命令git rm删掉，并且git commit：

1. $ rm -f test.txt
2. $ git rm test.txt
3. rm 'test.txt'
5. $ git commit -m "delete test from repository"
6. [master 9dfc09c] delete test from repository
7. 1 file changed, 1 deletion(-)
8. delete mode 100644 test.txt

## 对比文件差异

对比本地文件和版本库文件差异命令如下：

1. $ git diff <file>

例如，修改readme.txt文件，对比本地和版本库readme.txt文件差异

1. $ echo "This is a third modify." >> readme.txt
2. $ git diff readme.txt
3. diff --git a/readme.txt b/readme.txt
4. index c953448..0bf7aa7 100644
5. --- a/readme.txt
6. +++ b/readme.txt
7. @@ -2,3 +2,4 @@ This is vicgong using repository.
8. The repository only use study, no privacy involved.
9. This is a first modify.
10. This is a second modify.
11. +This is a third modify.

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。

## git管理修改

如下操作过程：

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

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。

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：

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

总结：Git每次修改，如果不使用git add命令添加到暂存区，那就不会加入到commit中。

1. Git分支管理

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。在创建版本库后默认只有一条时间线，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分支。而master用于指向提交的版本号，所以HEAD指向的就是当前分支。

创建版本库后，只有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：



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，这样的合并称为fast forward,当然不止这一种合并方式。



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：



## 创建并切换分支

创建并切换分支：

1. git checkout –b <branch-name>
2. 等价于
3. git branch <branch-name> 创建分支
4. git checkout <branch-name> 切换分支

例如，我们创建dev分支，然后切换到dev分支：

1. $ git checkout -b dev
2. Switched to a **new** branch 'dev'

例如，创建dev2分支，然后切换到dev2分支：

1. $ git branch dev2
2. $ git checkout dev2
3. Switched to branch 'dev2'

## 删除分支

创建分支后，在未发生提交的情况下或者在合并完成后，可通过如下命令直接删除分支：

1. $ git branch -d <branch-name>

如果删除一个发生提交但是并没有合并的分支，需要通过“-D”参数进行强制删除，这就意味放弃该分支所做的所有提交：

1. $ git branch –D <branch-name>

由于创建、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。

例如，删除刚创建的dev2分支，需要先切换到其他分支

1. $ git status
2. On branch dev2
3. nothing to commit, working tree clean
5. $ git checkout master
6. Switched to branch 'master'
7. Your branch is up to date with 'origin/master'.
9. $ git branch -d dev2
10. Deleted branch dev2 (was 42f3581).
12. $ git branch
13. dev
14. \* master

例如，创建dev2分支，同时添加文件并提交至dev2分支，而后删除分支

1. $ git checkout -b dev2
2. Switched to a **new** branch 'dev2'
4. $ echo dddd>test.txt
5. $ git add test.txt
6. $ git commit -m "test del branch"
7. [dev2 9ed3031] test del branch
8. 1 file changed, 1 insertion(+)
9. create mode 100644 test.txt
11. $ git branch -d dev2
12. error: Cannot delete branch 'dev2' checked out at 'E:/GitDocument'
14. $ git checkout master
15. Switched to branch 'master'
16. Your branch is up to date with 'origin/master'.
18. $ git branch -d dev2
19. error: The branch 'dev2' is not fully merged.
20. If you are sure you want to delete it, run 'git branch -D dev2'.
22. $ git branch -D dev2
23. Deleted branch dev2 (was 9ed3031).

## 查看分支

查看当前分支git branch命令会列出所有分支，当前分支前面会标一个\*号。-v选项用于获取分支的当前版本号:

1. $ git branch
2. $ git branch -v

例如，查看当前分支信息

1. $ git branch
2. dev
3. \* master
5. $ git branch -v
6. dev    42f3581 Initial commit
7. \* master 42f3581 Initial commit

## 合并分支

git merge用于合并指定分支到当前分支：

1. $ git checkout master
2. $ git merge <branch-name>

例如，切换到dev分支，对README.md文件进行修改并提交

1. $ echo "this is a modify 1" >> README.md
2. $ git add README.md
3. $ git commit -m "modify 1"
4. [master ba9b20e] modify 1
5. 1 file changed, 1 insertion(+), 1 deletion(-)

现在，dev分支的工作完成，切换回master分支：

1. $ git checkout master
2. Switched to branch 'master'

切换回master分支后，再查看一个README.md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。合并后，再查看文件的内容，就可以看到，和dev分支的最新提交是完全一样的。

1. $ git merge dev
2. Updating 7dd8f76..ff2ecd7
3. Fast-forward
4. README.md | 1 +
5. 1 file changed, 1 insertion(+)

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。

一般在实际工作中合并完成分支后习惯性删除该分支：

1. $ git branch -d dev
2. Deleted branch dev (was ff2ecd7).

## 禁用ff合并模式

通常合并分支时，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。而强制禁用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被删除的分支信息。

例如，禁用ff合并模式。首先，仍然创建并切换dev分支，修改README.md文件并提交一个新的commit。

1. $ git checkout -b dev
2. Switched to a **new** branch 'dev'

修改readme.txt文件，并提交一个新的commit：

1. $ echo "this is modify 2" >> README.md
2. $ git add README.md
3. $ git commit -m "test modify 2 dev"
4. [dev 07b1727] test modify 2 dev
5. 1 file changed, 1 insertion(+)

现在，我们切换回master，准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward，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。

1. $ git checkout master
2. Switched to branch 'master'
4. $ git merge --no-ff -m "merge with no-ff" dev
5. Merge made by the 'recursive' strategy.
6. README.md | 1 +
7. 1 file changed, 1 insertion(+)

合并后，我们用git log看看分支历史：

1. $ git log --graph --pretty=oneline --abbrev-commit
2. \*   86b8f8d (HEAD -> master) merge with no-ff
3. |\
4. | \* 07b1727 (dev) test modify 2 dev
5. |/
6. \* ff2ecd7 test modify dev
7. \* 7dd8f76 rollback
8. \* ba9b20e modify 1
9. ...

可以看到，不使用Fast forward模式，merge后就像这样：



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。

## 合并分支解决冲突

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。解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。用git log --graph命令可以看到分支合并图。

如下为解决冲突实例：

创建并切换feature分支，修改README.md文件，并且在feature分支上提交：

1. $ git checkout -b feature
2. Switched to a **new** branch 'feature'
4. $ echo **this** is a feature modify >> README.md
5. $ git add README.md README.md
6. $ cat README.md
7. # GitDocument
8. **this** is a modify 1
9. **this** is modify 2
10. **this** is a feature modify
12. $ git commit -m "feature modify"
13. [feature 3793c7e] feature modify
14. 1 file changed, 1 insertion(+)

切换到master分支，在master分支修改README.md文件，并提交：

1. $ git checkout master
2. Switched to branch 'master'
4. $ echo "this is a master modify" >> README.md
5. $ cat README.md
6. # GitDocument
7. **this** is a modify 1
8. **this** is modify 2
9. **this** is a master modify
11. $ git add README.md README.md
12. $ git commit -m "master modify"
13. [master 0254920] master modify
14. 1 file changed, 1 insertion(+)

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，但这种合并会有冲突，必须手动解决冲突后再提交。在执行merge后，直接查看readme.txt的内容。

1. $ git merge feature
2. Auto-merging README.md
3. CONFLICT (content): Merge conflict in README.md
4. Automatic merge failed; fix conflicts and then commit the result.
6. $ cat README.md
7. # GitDocument
8. **this** is a modify 1
9. **this** is modify 2
10. <<<<<<< HEAD
11. **this** is a master modify
12. =======
13. **this** is a feature modify
14. >>>>>>> feature

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容。“<<<<<<< HEAD”表示为当前的提交前的HEAD指向的文本点，之后的修改为master分支发生的修改。在往后为feature分支的修改。

在执行merge后需要手动的修改冲突文件然后再提交。文本内容修改如下：

1. $ cat README.md
2. # GitDocument
3. **this** is a modify 1
4. **this** is modify 2
5. **this** is a master modify
6. **this** is a feature modify
8. $ git add README.md
9. $ git commit -m "fixed confllict"
10. [master e8bf0cf] fixed confllict

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：

1. $ git log --graph --pretty=oneline --abbrev-commit
2. \*   e8bf0cf (HEAD -> master) fixed confllict
3. |\
4. | \* 3793c7e (feature) feature modify
5. \* | 0254920 master modify
6. |/
7. \*   86b8f8d merge with no-ff
8. |\
9. | \* 07b1727 test modify 2 dev
10. |/
11. \* ff2ecd7 test modify dev
12. \* 7dd8f76 rollback
13. \* ba9b20e modify 1
14. \* 42f3581 (origin/master, origin/HEAD) Initial commit

在解决冲突后，通常删除feature分支：

1. $ git branch -d feature
2. Deleted branch feature (was 3793c7e).

## 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：

* master分支应该保持稳定，仅用来发布新版本，不能再其进行开发；
* 在dev分支上进行开发，开发完成后，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；
* 开发者需要在本地的dev分支开发，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：



## 隐藏工作区修改

在某些情况在工作区内容并未完成而无法提交，这时也不允许切换分支。这时候可以通过stash命令另存工作区，在其他工作完成后进行恢复工作区修改。相关命令如下：

1. 隐藏工作区：
2. $ git stash
4. 查看隐藏列表
5. $ git stash list
7. 恢复隐藏的工作区（如果当前只有一个stash可不指定stash名称）
8. $ git stash apply <stash-name>
10. 删除隐藏的工作区
11. $ git stash drop <stash-name>
13. 恢复并删除隐藏的工作区
14. $ git stash pop <stash-name>

实例，创建dev分支，并进行修改，但此时开发未完成，不能提交。这时候需要修复一个代号101的bug的任务时，无法切换分支。git提示我们可以通过git stash功能将该修改隐藏起来，然后创建bug分支修复bug。等待bug完成后，切换回到dev, 发现工作区是干净的，刚才的工作现场被隐藏了。

1. $ echo "this is a temp modify" >> README.md
2. $ cat README.md
3. # GitDocument
4. **this** is a modify 1
5. **this** is modify 2
6. **this** is a master modify
7. **this** is a feature modify
8. **this** is a temp modify
10. $ git checkout master
11. error: Your local changes to the following files would be overwritten by checkout:
12. README.md
13. Please commit your changes or stash them before you **switch** branches.
14. Aborting
15. $ git stash
16. The file will have its original line endings in your working directory
17. Saved working directory and index state WIP on dev: e8bf0cf fixed confllict
18. #dev: e8bf0cf fixed confllict说明：
19. 保存的工作区的提交号为e8bf0cf和提交注释fixed confllict
21. $ git checkout -b bug
22. Switched to a **new** branch 'bug'
23. $ echo "fixed bug 101" >> README.md
24. $ git add README.md
25. $ git commit -m "bug 101 fixed"
26. [bug d0f212c] bug 101 fixed
27. 1 file changed, 1 insertion(+)
29. $ git checkout master
30. Switched to branch 'master'
31. Your branch is up to date with 'origin/master'.
33. $ git merge bug
34. Updating e8bf0cf..d0f212c
35. Fast-forward
36. README.md | 1 +
37. 1 file changed, 1 insertion(+)
39. $ git branch -d bug
40. Deleted branch bug (was d0f212c).
42. $ git checkout dev
43. Switched to branch 'dev'
45. $ git status
46. On branch dev
47. nothing to commit, working tree clean

git stash list命令查看，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。

1. $ git stash list
2. stash@{0}: WIP on dev: e8bf0cf fixed confllict

一种是先用apply恢复,但是恢复后隐藏的区间不会被删除，需要使用drop命令来删除该隐藏区间。

1. $ git stash apply
2. On branch dev
3. Changes not staged **for** commit:
4. (use "git add <file>..." to update what will be committed)
5. (use "git checkout -- <file>..." to discard changes in working directory)
7. modified:   README.md
9. no changes added to commit (use "git add" and/or "git commit -a")
11. $ git stash drop
12. Dropped refs/stash@{0} (fb66ba8a84d64d64557bb3e34ddae44759aebd24)

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

1. $ git stash pop
2. On branch dev
3. Changes not staged **for** commit:
4. (use "git add <file>..." to update what will be committed)
5. (use "git checkout -- <file>..." to discard changes in working directory)
7. modified:   README.md
9. Dropped refs/stash@{0} (fb66ba8a84d64d64557bb3e34ddae44759aebd24)

再用git stash list查看，就看不到任何stash内容了。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：

1. $ git stash apply stash@{0}
2. GitHub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。最原始的状态下，只有一台机器上有原始版本。此后，其他的机器可以“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。

实际情况往往是通过以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。同时各自把各自的提交推送到服务器仓库里，而且也从服务器仓库中拉取别人的提交。

当然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。

## GitHub添加SSH Key

由于本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要免密设置：

**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：

1. $ ssh-keygen -t rsa -C "964744855@qq.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。

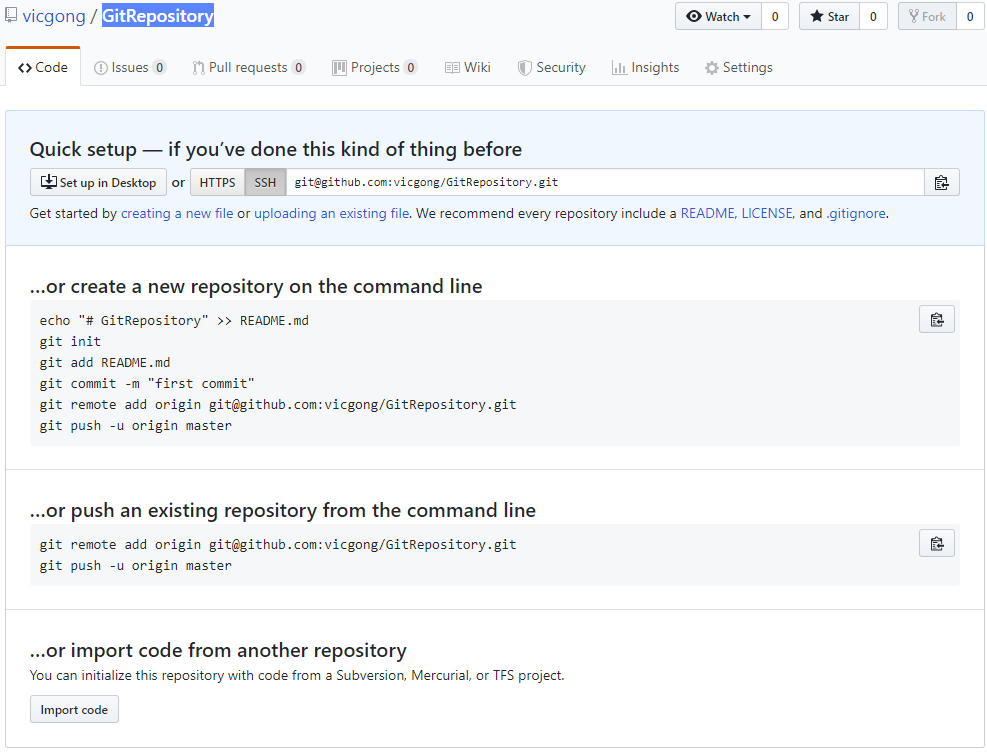
## GitHub创建远程库

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。

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GitRepository，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：



## 远程库关联本地库

关联远程库：

1. $ git remote add origin git@github.com:username/repositoryName.git

推送本地库绑定master分支到远程库master分支：

1. $ git push –u origin master

实例如下：

在GitHub上的这个GitRepository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。

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。

1. HTTPS协议：https://github.com/username/repositoryName.git
2. SSH协议：git@github.com:username/repositoryName.git

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。但通过ssh支持的原生git协议速度最快。

在本地仓库关联远程仓库命令如下：

1. $ git remote add origin git@github.com:vicgong/GitRepository.git

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。

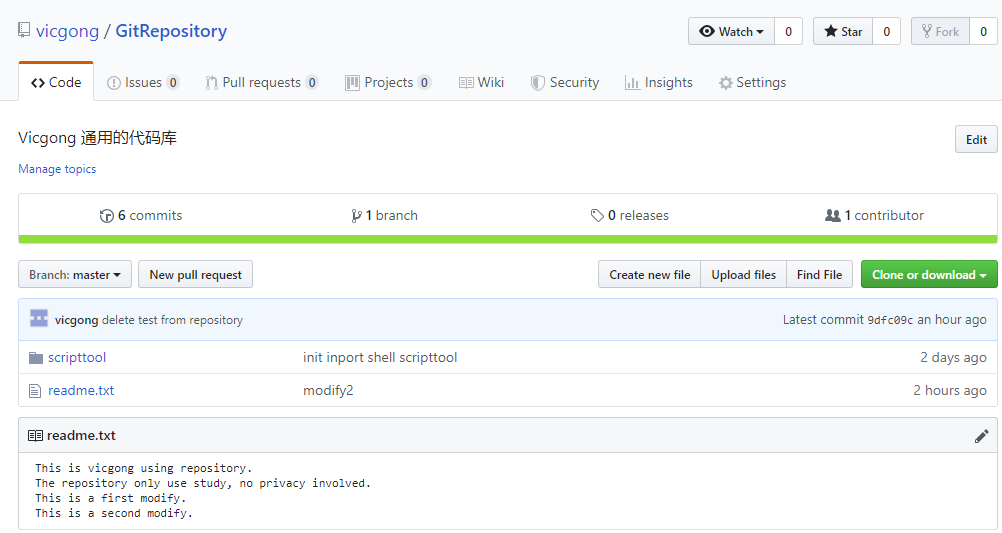
下一步，就可以把本地库的所有内容推送到远程库上：

1. $ git push -u origin master
2. Enumerating objects: 41, done.
3. Counting objects: 100% (41/41), done.
4. Delta compression using up to 4 threads
5. Compressing objects: 100% (36/36), done.
6. Writing objects: 100% (41/41), 21.28 KiB | 660.00 KiB/s, done.
7. Total 41 (delta 5), reused 0 (delta 0)
8. remote: Resolving deltas: 100% (5/5), done.
9. To github.com:vicgong/GitRepository.git
10. \* [**new** branch]      master -> master
11. 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：



## 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：

1. The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.
2. RSA key fingerprint is xx.xx.xx.xx.xx.
3. 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：

1. 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的指纹信息是否与SSH连接给出的一致。

## 推送分支

只要本地作了提交，就可以通过如下命令，将本地master分支的最新修改推送至GitHub：

1. $ git push origin master

如果要推送其他分支，比如dev如下：

1. $ git push origin dev

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！

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；

## 克隆版本库

克隆远程库至当前目录如下命令：

1. $ git clone git@github.com:username/repositoryName.git
2. $ cd ./repositoryName

克隆远程库至指定目录,该目录可以不存在，如果存在的话必须为空。克隆后版本库的名称为name

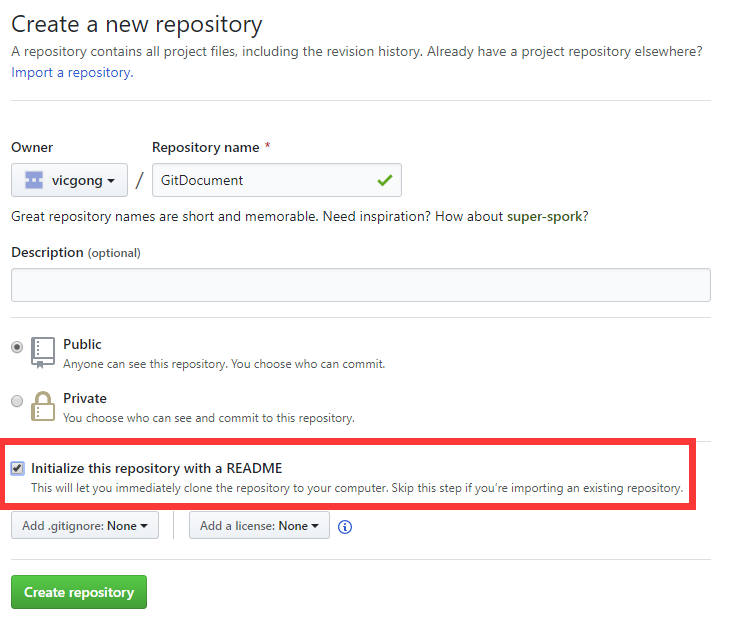
1. $ git clone git@github.com:username/repositoryName.git dirname/name
2. $ cd dirname/name

克隆本地仓库，克隆后版本库的名称为name2：

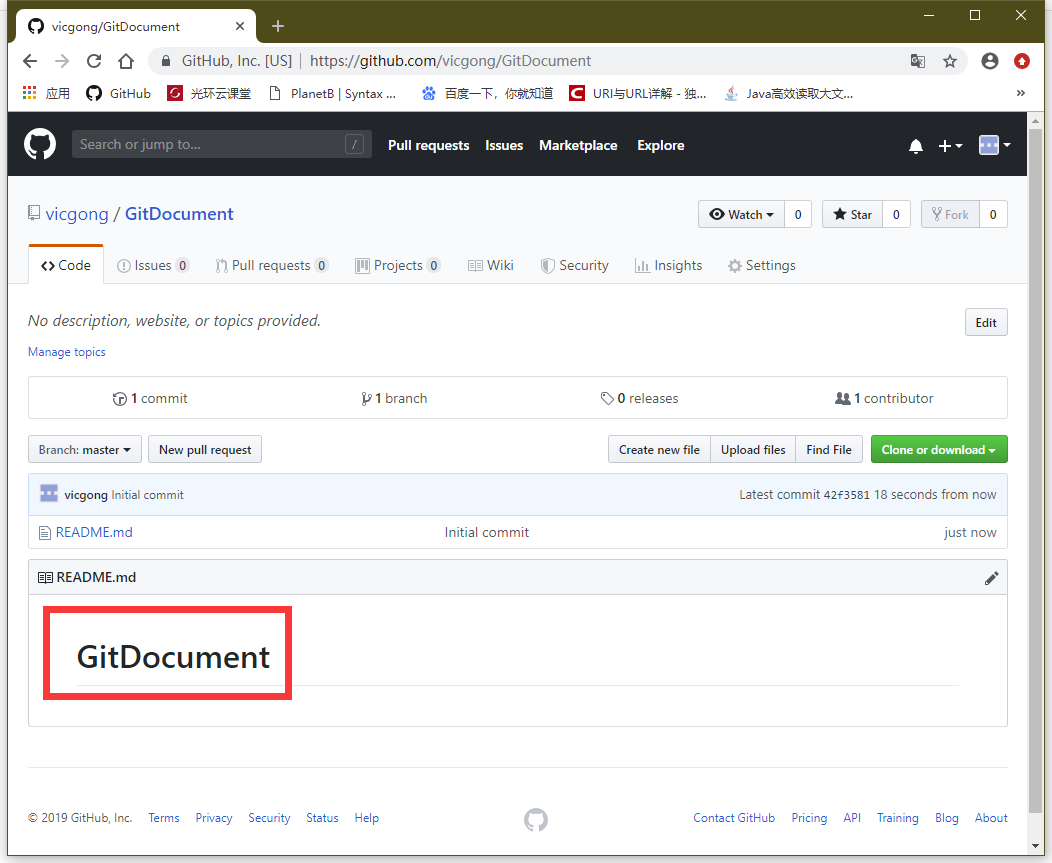
1. $ git clone dirname/name newDirname/name2
2. $ cd newDirname/name2

实例：在GitHub创建一个远程库，然后，从远程库克隆到本地。

首先，登陆GitHub，创建一个新的仓库，名字叫GitDocument：



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：

1. $ git clone git@github.com:vicgong/GitDocument.git
2. Cloning into 'GitDocument'...
3. remote: Enumerating objects: 3, done.
4. remote: Counting objects: 100% (3/3), done.
5. remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
6. Receiving objects: 100% (3/3), done.

后进入GitDocument目录查看，已经有README.md文件了：

1. $ cd GitDocument/
2. $ ls
3. README.md

克隆远程库到指定目录：

1. $  git clone git@github.com:vicgong/GitDocument.git /d/local
2. Cloning into 'D:/local'...
3. remote: Enumerating objects: 3, done.
4. remote: Counting objects: 100% (3/3), done.
5. remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
6. Receiving objects: 100% (3/3), done.
8. $ cd /d/local
9. $ ls
10. README.md

克隆本地库至指定目录：

1. $ git clone /d/local/ /d/test
2. Cloning into 'D:/test'...
3. done.
5. $ cd /d/test/
6. $ ls
7. README.md

## 克隆远程分支

通常情况下克隆远程版本库仅能克隆master分支，如果需要克隆远程分支到本地分支

1. $ git checkout –b <branch-name> origin/<branch-name>

如下实例示范，克隆远程分支到本地。首先，提交一个dev分支到远程仓库。

1. $ git push origin dev
2. Total 0 (delta 0), reused 0 (delta 0)
3. remote:
4. remote: Create a pull request **for** 'dev' on GitHub by visiting:
5. remote:      https://github.com/vicgong/GitDocument/pull/new/dev
6. remote:
7. To github.com:vicgong/GitDocument.git
8. \* [**new** branch]      dev -> dev

克隆远程库到其他目录，默认情况下，只能看到本地的master分支。

1. $ git clone git@github.com:vicgong/GitDocument.git /d/GitDocumnet
2. Cloning into 'D:/GitDocumnet'...
3. remote: Enumerating objects: 28, done.
4. remote: Counting objects: 100% (28/28), done.
5. remote: Compressing objects: 100% (11/11), done.
6. remote: Total 28 (delta 4), reused 25 (delta 4), pack-reused 0
7. Receiving objects: 100% (28/28), done.
8. Resolving deltas: 100% (4/4), done.
10. $ cd /d/GitDocumnet/
11. $ git branch
12. \* master

现在，需要dev分支上开发，就必须创建远程origin的dev分支到本地：

1. $ git checkout -b dev origin/dev
2. Switched to a **new** branch 'dev'
3. Branch 'dev' set up to track remote branch 'dev' from 'origin'.

## 提取远程库内容

从远程版本库重新提取全部内容至本地库，如果本地库有提交的并不会被修改：

1. $ git fetch origin

实例，修改readme.txt,添加暂存区并提交至本地库

1. $ cat readme.txt
2. This is vicgong using repository.
3. The repository only use study, no privacy involved.
4. This is a first modify.
5. This is a second modify.
6. This is a third modify.
8. $ git status
9. On branch master
10. Your branch is up to date with 'origin/master'.
12. Changes not staged **for** commit:
13. (use "git add <file>..." to update what will be committed)
14. (use "git checkout -- <file>..." to discard changes in working directory)
16. modified:   readme.txt
18. no changes added to commit (use "git add" and/or "git commit -a")
20. $ git add readme.txt
21. $ git commit -m "modify3"
22. [master 87955f0] modify3
23. 1 file changed, 1 insertion(+)
25. $ git status
26. On branch master
27. Your branch is ahead of 'origin/master' by 1 commit.
28. (use "git push" to publish your local commits)
30. nothing to commit, working tree clean

从远程库提取全部内容：

1. $ git fetch origin
2. $ git status
3. On branch master
4. Your branch is ahead of 'origin/master' by 1 commit.
5. (use "git push" to publish your local commits)
7. nothing to commit, working tree clean
9. $ cat readme.txt
10. This is vicgong using repository.
11. The repository only use study, no privacy involved.
12. This is a first modify.
13. This is a second modify.
14. This is a third modify.

## 查看远程库信息

查看本地库关联的远程库信息：

1. $ git remote
2. origin

用git remote -v显示更详细的信息：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。

1. $ git remote -v
2. origin  https://github.com/vicgong/VicGit.git (fetch)
3. origin  https://github.com/vicgong/VicGit.git (push)

## 拉取分支版本

获取本地分支对应的远程分支内容，如下表示抓取远程master的分支内容到本地：

1. $ git branch
2. \* master
4. $ git pull

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，使用如下命令创建分支链接,然后再抓取到本地：

1. $ git branch --set-upstream-to <branch-name> origin/<branch-name>
2. $ git pull

## 推送提交解决冲突

在推送提交的过程中，也可能会发生合并冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。pull抓取远程版本试图合并，然后手动的解决冲突，然后再提交本地版本库，最后再push到远程库。

因此，多人协作的工作模式通常如下：

* 首先，可以试图用git push origin <branch-name>推送自己的修改；
* 如果推送失败，则因为远程分支新于你的本地版本，需要先用git pull试图合并；
* 如果合并有冲突，则解决冲突，并在本地提交；
* 没有冲突或者解决掉冲突后，再用push推送就能成功！

如下实例模拟推送冲突：

首先，在之前克隆远程分支时将服务器的dev分支，克隆到本地dev分支。目录位于D:/GitDocument。假设A继续在dev上修改，然后， dev分支push到远程：

1. $ cat README.md
2. # GitDocument
3. **this** is a modify 1
4. **this** is modify 2
5. **this** is a master modify
6. **this** is a feature modify
7. fixed bug 101
9. $ echo "/d/GitDocument modify" >> README.md
10. $ git add README.md
11. $ git commit -m "/d/GitDocument process"
12. [dev bddca66] D:/GitDocument process
13. 1 file changed, 1 insertion(+)
15. $ git push origin dev
16. Enumerating objects: 5, done.
17. Counting objects: 100% (5/5), done.
18. Delta compression using up to 4 threads
19. Compressing objects: 100% (2/2), done.
20. Writing objects: 100% (3/3), 298 bytes | 149.00 KiB/s, done.
21. Total 3 (delta 1), reused 0 (delta 0)
22. remote: Resolving deltas: 100% (1/1), completed with 1 local object.
23. To github.com:vicgong/GitDocument.git
24. d0f212c..bddca66  dev -> dev

A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：

1. $ cd /e/GitDocument/
2. $ cat README.md
3. # GitDocument
4. **this** is a modify 1
5. **this** is modify 2
6. **this** is a master modify
7. **this** is a feature modify
8. fixed bug 101
10. $ echo "/e/GitDocument modify" >> README.md
11. $ git add README.md
12. $ git commit -m "/e/GitDocument process"
13. [dev b2eb2c5] E:/GitDocument process
14. 1 file changed, 1 insertion(+)
16. $ git push origin dev
17. To github.com:vicgong/GitDocument.git
18. ! [rejected]        dev -> dev (fetch first)
19. error: failed to push some refs to 'git@github.com:vicgong/GitDocument.git'
20. hint: Updates were rejected because the remote contains work that you **do**
21. hint: not have locally. This is usually caused by another repository pushing
22. hint: to the same ref. You may want to first integrate the remote changes
23. hint: (e.g., 'git pull ...') before pushing again.
24. hint: See the 'Note about fast-forwards' in 'git push --help' **for** details.

结果推送失败，因为A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：

1. $ git pull
2. remote: Enumerating objects: 5, done.
3. remote: Counting objects: 100% (5/5), done.
4. remote: Compressing objects: 100% (1/1), done.
5. remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
6. Unpacking objects: 100% (3/3), done.
7. From github.com:vicgong/GitDocument
8. d0f212c..bddca66  dev        -> origin/dev
9. There is no tracking information **for** the current branch.
10. Please specify which branch you want to merge with.
11. See git-pull(1) **for** details.
13. git pull <remote> <branch>
15. If you wish to set tracking information **for** **this** branch you can **do** so with:
17. git branch --set-upstream-to=origin/<branch> dev

结果git pull也失败了，原因是“ no tracking information **for** the current branch”。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：

1. $ git branch --set-upstream-to=origin/dev dev
2. Branch 'dev' set up to track remote branch 'dev' from 'origin'.

再pull抓取最新的文件，查看文件内容并手动合并。然后推送到远程版本库：

1. $ git pull
2. Auto-merging README.md
3. CONFLICT (content): Merge conflict in README.md
4. Automatic merge failed; fix conflicts and then commit the result.
6. $ cat README.md
7. # GitDocument
8. **this** is a modify 1
9. **this** is modify 2
10. **this** is a master modify
11. **this** is a feature modify
12. fixed bug 101
13. <<<<<<< HEAD
14. /e/GitDocument modify
15. =======
16. /d/GitDocument modify
17. >>>>>>> bddca66a15e4e40dbcb86cfa5ec50347462c1367
19. $ vi README.md
20. $ cat README.md
21. # GitDocument
22. **this** is a modify 1
23. **this** is modify 2
24. **this** is a master modify
25. **this** is a feature modify
26. fixed bug 101
27. /e/GitDocument modify
28. /d/GitDocument modify
30. $ git add README.md
31. $ git commit -m "fixed origin dev reflict"
32. [dev e1bdca1] fixed origin dev reflict
34. $ git push origin dev
35. Enumerating objects: 10, done.
36. Counting objects: 100% (10/10), done.
37. Delta compression using up to 4 threads
38. Compressing objects: 100% (4/4), done.
39. Writing objects: 100% (6/6), 544 bytes | 136.00 KiB/s, done.
40. Total 6 (delta 2), reused 0 (delta 0)
41. remote: Resolving deltas: 100% (2/2), completed with 1 local object.
42. To github.com:vicgong/GitDocument.git
43. bddca66..e1bdca1  dev -> dev

A也可以获取最新的dev版本:

1. $ cd /d/GitDocumnet/
2. $ git pull
3. remote: Enumerating objects: 10, done.
4. remote: Counting objects: 100% (10/10), done.
5. remote: Compressing objects: 100% (2/2), done.
6. remote: Total 6 (delta 2), reused 6 (delta 2), pack-reused 0
7. Unpacking objects: 100% (6/6), done.
8. From github.com:vicgong/GitDocument
9. bddca66..e1bdca1  dev        -> origin/dev
10. Updating bddca66..e1bdca1
11. Fast-forward
12. README.md | 1 +
13. 1 file changed, 1 insertion(+)
15. $ cat README.md
16. # GitDocument
17. **this** is a modify 1
18. **this** is modify 2
19. **this** is a master modify
20. **this** is a feature modify
21. fixed bug 101
22. /e/GitDocument modify
23. /d/GitDocument modify
24. Git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。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！”

## 创建标签

在当前HEAD版本上创建标签，默认标签是打在最新提交的commit上的。

1. $ git tag <tag-name>

在指定的commit\_id上创建标签：

1. $ git tag <tag-name> <commit-id>

创建标签同时指定标签的备注信息

1. $ git tag -a <tag-name> -m "comment" <commit-id>

例如，在当前版本创建标签v1.0：

1. $ git branch
2. \* dev
3. master
4. $ git checkout master
5. Switched to branch 'master'
6. $ git tag v1.0

例如，在历史版本中的某个commit-id上创建标签，需要获取历史提交的commit id，比方说要对commit id是ba9b20的版本创建标签。

1. $ git tag v0.9 ba9b20

## 查看标签

查看所有标签：

1. $ git tag

查看标签详细信息：

1. $ git show <tag-name>

例如，查看所有的标签：

1. $ git tag
2. v0.9
3. v1.0

注意，标签不是按时间顺序列出，而是按字母排序的。git show查看标签的全部信息：

1. $ git show v0.9
2. commit ba9b20e1163ae497865bbf08fb9e10b7925071d8 (tag: v0.9)
3. Author: vicgong <964744855@qq.com>
4. Date:   Sat May 25 19:31:59 2019 +0800
6. modify 1
8. diff --git a/README.md b/README.md
9. index 42cd508..8eebe59 100644
10. --- a/README.md
11. +++ b/README.md
12. @@ -1 +1 @@
13. -# GitDocument
14. \ No newline at end of file
15. +# GitDocumentthis is a modify 1

还可以创建带有说明的标签，用-a指定标签名，-m指定备注信息：

1. $ git tag -a v0.9.1 -m "version 0.9.1 released" 7dd8f7

用命令git show <tagname>可以看到标签名及备注信息：

1. $ git show v0.9.1
2. tag v0.9.1
3. Tagger: vicgong <964744855@qq.com>
4. Date:   Sun May 26 15:21:28 2019 +0800
6. version 0.9.1 released
8. commit 7dd8f7656b88bab4ccd16f37f54cf7b64f31e650 (tag: v0.9.1)
9. Author: vicgong <964744855@qq.com>
10. Date:   Sat May 25 19:35:02 2019 +0800
12. rollback
14. diff --git a/README.md b/README.md
15. index 8eebe59..9941b95 100644
16. --- a/README.md
17. +++ b/README.md
18. @@ -1 +1 @@
19. -# GitDocumentthis is a modify 1
20. +# GitDocument

注意：标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签。

## 删除本地标签

删除本地标签：

1. $ git tag –d <tag-name>

由于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。

例如，标签打错了可以删除它：

1. $ git tag -d v0.1
2. Deleted tag 'v0.1' (was f15b0dd)

## 推送标签

推送标签：

1. $ git push <origin> <tag-name>

例如，推送某个标签到远程

1. $ git push origin v0.9
2. Total 0 (delta 0), reused 0 (delta 0)
3. To github.com:vicgong/GitDocument.git
4. \* [**new** tag]         v0.9 -> v0.9

## 删除远程标签

删除远程标签, 先从本地删除。然后，从远程删除:

1. $ git tag –d <tag-name>
2. $ git push origin :refs/tags/tag-name

例如，删除远程标签

1. $ git tag -d v0.9
2. Deleted tag 'v0.9' (was ba9b20e)
4. $ git push origin :refs/tags/v0.9
5. To github.com:vicgong/GitDocument.git
6. - [deleted]         v0.9