

一、安装Git

1.设置邮箱和名字

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

2.创建版本库

```
# 创建空目录
$ mkdir learngit
$ cd learngit
$ pwd
/Users/michael/learngit

#使用 git init命令把目录变为仓库。
$ git init
Initialized empty Git repository in /Users/michael/learngit/.git/

#创建一个readme.txt文件

#把文件加入仓库 git add <file>
$ git add readme.txt

# 提交文件到仓库 git commit -m <message>
$ git commit -m "wrote a readme file"
[master (root-commit) eaadf4e] wrote a readme file
1 file changed, 2 insertions(+)
create mode 100644 readme.txt
```

可以多次add文件并一次提交：

```
$ git add file1.txt
$ git add file2.txt file3.txt
$ git commit -m "add 3 files."
```

二、版本控制

修改readme.txt文件：

Git is a distributed version control system.
Git is free software.

1.git status

`git status` 命令可以让我们时刻掌握仓库当前的状态：

```
$ git status
On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

    modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
#readme.txt 已经修改 但没有准备将要被提交的修改
```

2.git diff

`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 diff readme.txt
diff --git a/readme.txt b/readme.txt
index 46d49bf..9247db6 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,2 @@
-Git is a version control system.
+Git is a distributed version control system.
 Git is free software.
#第一行增加了一个'distributed'

#提交修改
#$ git add readme.txt
#$ git status
#On branch master
#Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#    modified:   readme.txt
#
#$ git commit -m "add distributed"
#[master e475afc] add distributed
# 1 file changed, 1 insertion(+), 1 deletion(-)
#
#$ git status
#On branch master
#nothing to commit, working tree clean
```

3.版本回退

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

假设现在有如下版本

版本1: wrote a readme file

```
Git is a version control system.  
Git is free software.:
```

版本2: add distributed

```
Git is a distributed version control system.  
Git is free software.
```

版本3: append GPL

```
Git is a distributed version control system.  
Git is free software distributed under the GPL.
```

使用 `git log` 查看历史记录

```
$ git log  
commit 1094adb7b9b3807259d8cb349e7df1d4d6477073 (HEAD -> master)  
Author: Michael Liao <askxuefeng@gmail.com>  
Date:   Fri May 18 21:06:15 2018 +0800  
  
    append GPL  
  
commit e475afc93c209a690c39c13a46716e8fa000c366  
Author: Michael Liao <askxuefeng@gmail.com>  
Date:   Fri May 18 21:03:36 2018 +0800  
  
    add distributed  
  
commit eaadf4e385e865d25c48e7ca9c8395c3f7dfaef0  
Author: Michael Liao <askxuefeng@gmail.com>  
Date:   Fri May 18 20:59:18 2018 +0800  
  
    wrote a readme file
```

需要友情提示的是, 你看到的一大串类似 `1094adb...` 的是 `commit id` (版本号)。

使用 `git reset` 命令从 `append GPL` 回退到 `add distributed` 版本

首先, Git必须知道当前版本是哪个版本, 在Git中, 用 `HEAD` 表示当前版本, 也就是最新的提交 `1094adb...` (注意我的提交ID和你的肯定不一样), 上一个版本就是 `HEAD^`, 上上一个版本就是 `HEAD^^`, 当然往上100个版本写100个 `^` 比较容易数不过来, 所以写成 `HEAD~100`。

```
$ git reset --hard HEAD^  
HEAD is now at e475afc add distributed
```

Git提供了一个命令 `git reflog` 用来记录你的每一次命令:

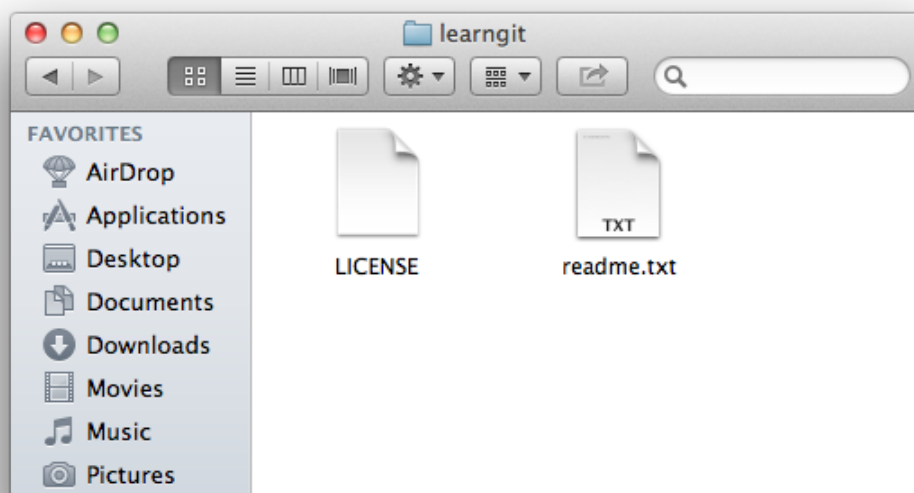
```
$ git reflog  
e475afc HEAD@{1}: reset: moving to HEAD^  
1094adb (HEAD -> master) HEAD@{2}: commit: append GPL  
e475afc HEAD@{3}: commit: add distributed  
eaadf4e HEAD@{4}: commit (initial): wrote a readme file
```

拿到 `append GPL` 版本号，“前进”到 `append GPL` 版本

```
# 版本号不一定要写全
$ git reset --hard 1094a
HEAD is now at 83b0afe append GPL
```

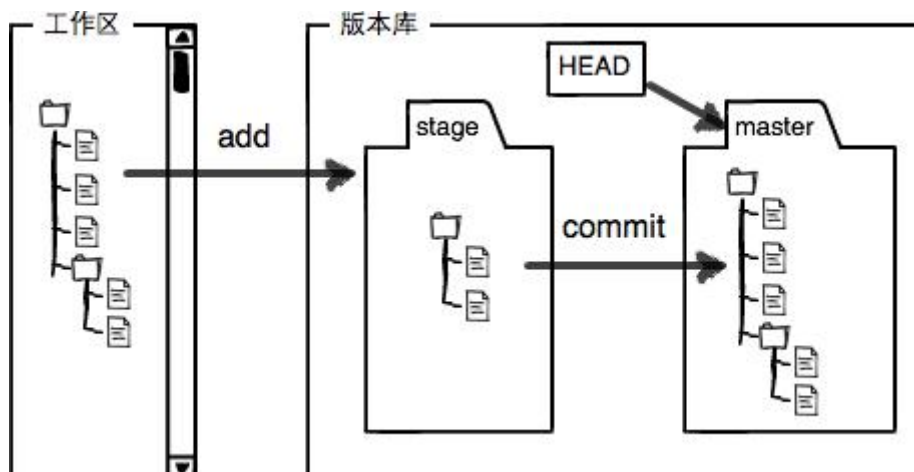
4.工作区和暂存区

工作区（Working Directory）就是你在电脑里能看到的目录



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

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



前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用 `git add` 把文件添加进去，实际上就是把文件修改添加到暂存区；

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

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

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

现在，我们再练习一遍，先对 `readme.txt` 做个修改，比如加上一行内容：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
```

然后，在工作区新增一个 `LICENSE` 文本文件（内容随便写）。

先用 `git status` 查看一下状态：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   readme.txt

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

    LICENSE

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

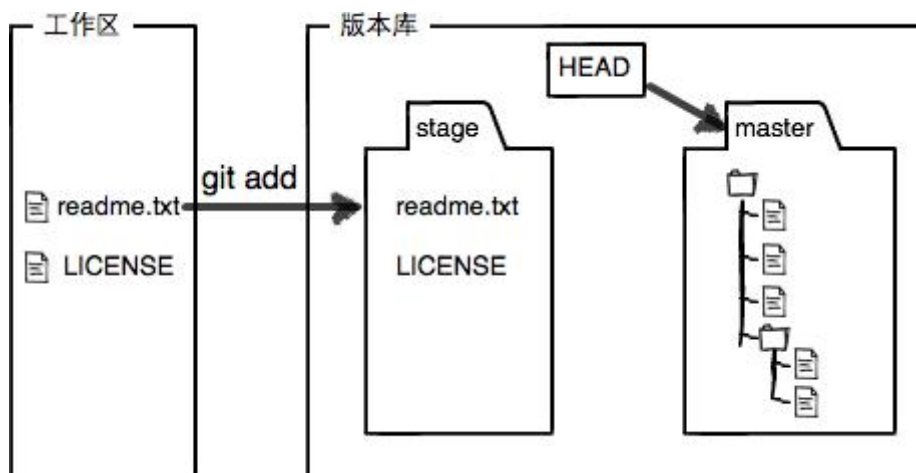
Git 非常清楚地告诉我们，`readme.txt` 被修改了，而 `LICENSE` 还从来没有被添加过，所以它的状态是 `Untracked`。

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

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   LICENSE
    modified:   readme.txt
```

现在，暂存区的状态就变成这样了：



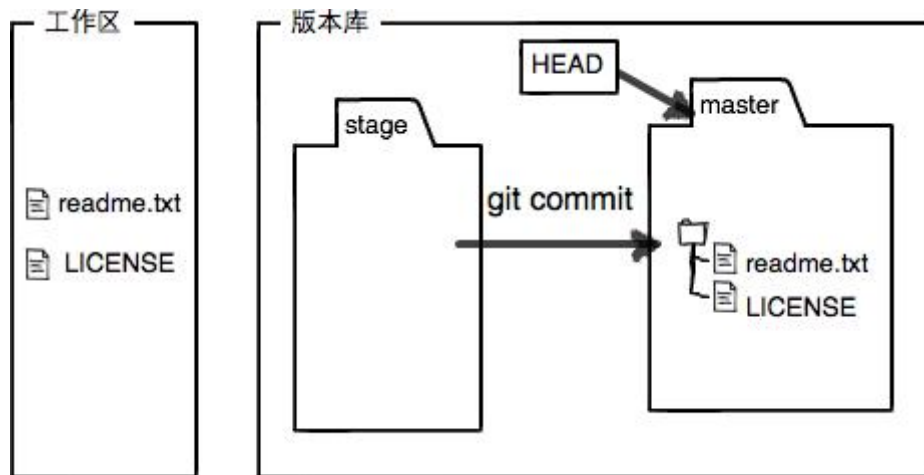
所以，`git add` 命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行 `git commit` 就可以一次性把暂存区的所有修改提交到分支。

```
$ git commit -m "understand how stage works"
[master e43a48b] understand how stage works
2 files changed, 2 insertions(+)
create mode 100644 LICENSE
```

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

```
$ git status
On branch master
nothing to commit, working tree clean
```

现在版本库变成了这样，暂存区就没有任何内容了：



(一下为摘抄评论区的评论)

感觉大家把简单问题复杂化了，看着头晕，

Git管理的文件分为：工作区，版本库，版本库又分为暂存区stage和暂存区分支master(仓库)

工作区>>>>暂存区>>>>仓库

git add把文件从工作区>>>>暂存区，git commit把文件从暂存区>>>>仓库，

git diff查看工作区和暂存区差异，

git diff --cached查看暂存区和仓库差异，

git diff HEAD 查看工作区和仓库的差异，

git add的反向命令git checkout，撤销工作区修改，即把暂存区最新版本转移到工作区，

git commit的反向命令git reset HEAD，就是把仓库最新版本转移到工作区。

5.管理修改

下面，我们要讨论的就是，为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

对readme.txt文件进行修改，并添加修改：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes.
```

```
#添加修改
$ git add readme.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   readme.txt
#
```

再次对readme.txt文件修改，但未添加修改：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
```

提交：

```
$ git commit -m "git tracks changes"
[master 519219b] git tracks changes
1 file changed, 1 insertion(+)
```

查看状态：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

       modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")
#显示第二次的修改没有被提交，因为 add readme.txt仅添加了第一次的修改，第二次的修改未添加，所以最后提交的只有第一次修改。

$ git diff HEAD -- readme.txt
diff --git a/readme.txt b/readme.txt
index 76d770f..a9c5755 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,4 +1,4 @@
  Git is a distributed version control system.
  Git is free software distributed under the GPL.
  Git has a mutable index called stage.
- Git tracks changes.
+ Git tracks changes of files.
#工作区比版本库最新版的区别：多了 'of files.'
```

6.撤销修改

`git checkout -- file` (新版被 `git restore <file>` 代替) :撤销工作区修改, 即把暂存区最新版本转移到工作区。

`git reset HEAD <file>` (新版被 `git restore --staged` 代替) :将暂存区修改重新放回工作区, 并撤销暂存区修改。

```
#增加readme.txt 一行'My stupid boss still prefers SVN.'
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
My stupid boss still prefers SVN.

#将修改添加到工作区
$ git add readme.txt

# 将暂存区的修改撤销并放回工作区
$ git reset HEAD readme.txt
Unstaged changes after reset:
M   readme.txt

#将工作区的修改丢弃
$ git checkout -- readme.txt

#文件恢复到
#Git is a distributed version control system.
#Git is free software distributed under the GPL.
#Git has a mutable index called stage.
#Git tracks changes of files.
```

文件已经提交到版本库则使用版本回退命令 `git reset` 命令 (git commit 后)

7.删除文件

如下场景:

创建一个test.txt文件, 并提交到版本库。

```
$ git add test.txt

$ git commit -m "add test.txt"
[master b84166e] add test.txt
1 file changed, 1 insertion(+)
create mode 100644 test.txt
```

在文件管理器中将文件删除:

```
$ rm test.txt
```

接下来你有两个选择:

①在版本库中将文件删除


```
$ git rm test.txt
# git add test.txt也可以

$ git commit -m "remove test.txt"
[master d46f35e] remove test.txt
1 file changed, 1 deletion(-)
delete mode 100644 test.txt
```

②删除文件系误删，你想还原

```
$ git restore test.txt
#将工作区恢复到上一次git add 或 git commit的状态
```

三、远程仓库

0.预备

第1步：创建SSH Key.

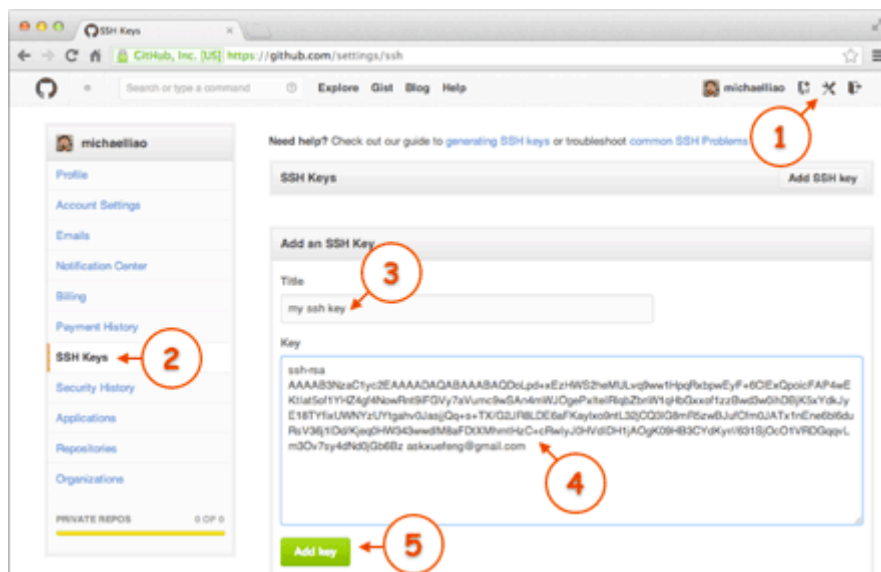
在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有 id_rsa 和 id_rsa.pub 这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

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

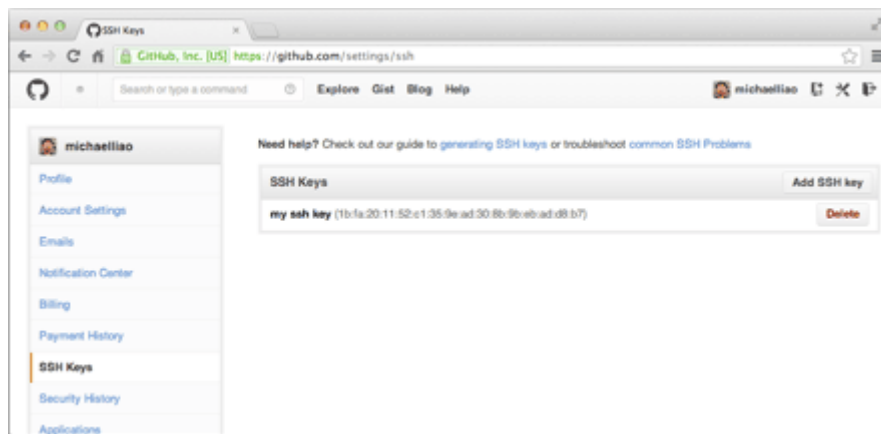
如果一切顺利的话，可以在用户主目录里找到 .ssh 目录，里面有 id_rsa 和 id_rsa.pub 两个文件，这两个就是SSH Key的密钥对，id_rsa 是私钥，不能泄露出去，id_rsa.pub 是公钥，可以放心地告诉任何人。

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：



1.添加远程库

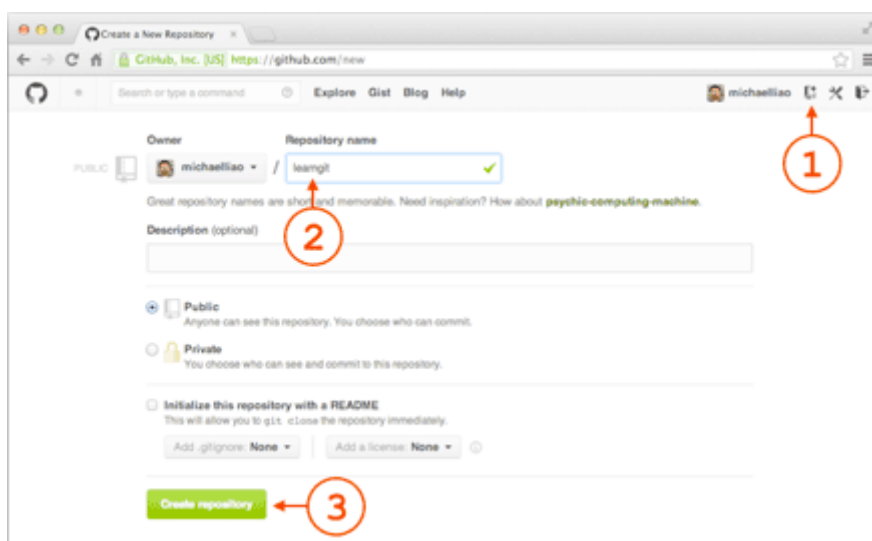
要关联一个远程库，使用命令 `git remote add origin git@server-name:path/repo-name.git`；

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

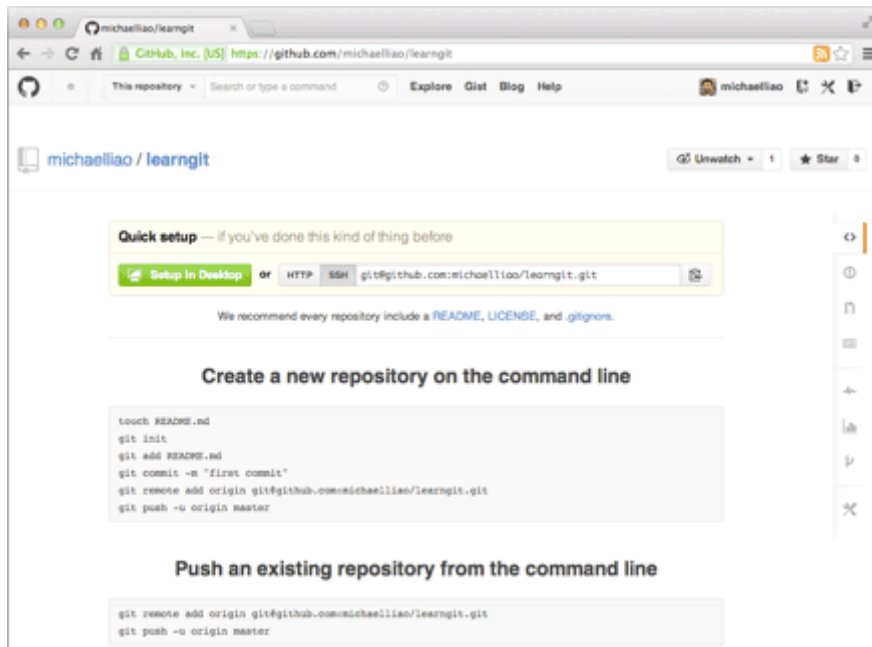
此后，每次本地提交后，只要有必要，就可以使用命令 `git push origin master` 推送最新修改；

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

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



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



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

现在，我们根据GitHub的提示，在本地的 `learngit` 仓库下运行命令：

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

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

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

下一步，就可以把本地库的所有内容推送到远程库上：

```
$ git push -u origin master
Counting objects: 20, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (15/15), done.
writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.
Total 20 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), done.
To github.com:michaelliao/learngit.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

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

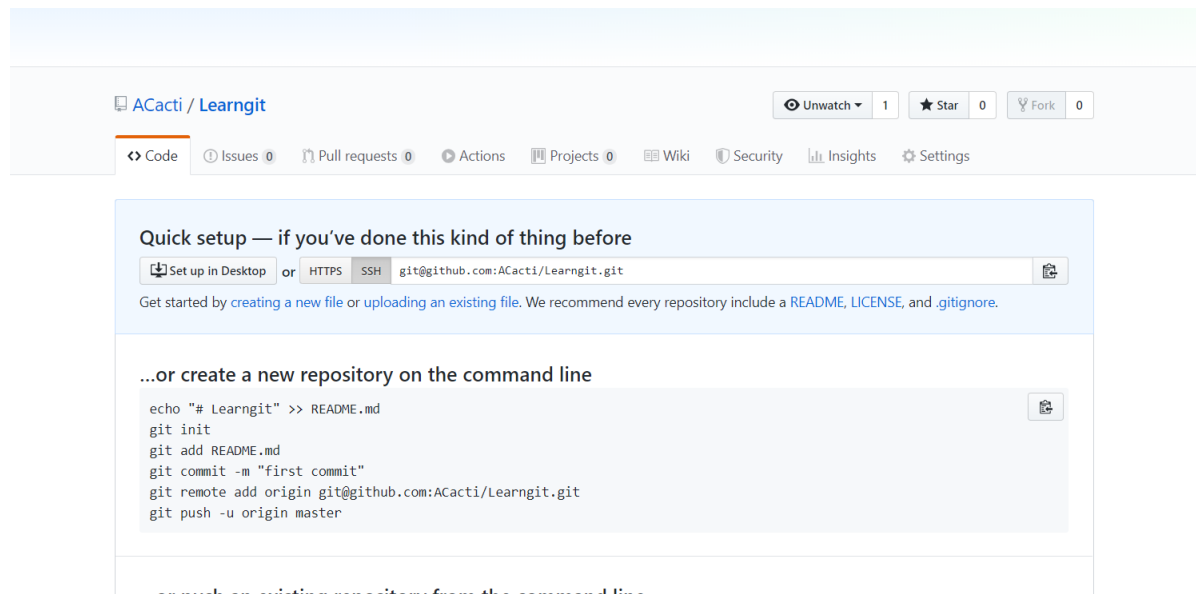
由于远程库是空的，我们第一次推送 `master` 分支时，加上了 `-u` 参数，Git不但会把本地的 `master` 分支内容推送的远程新的 `master` 分支，还会把本地的 `master` 分支和远程的 `master` 分支关联起来，在以后的推送或者拉取时就可以简化命令。

从现在起，只要本地作了提交，就可以通过命令：

```
$ git push origin master
```

2. 克隆仓库

在GitHub获取Git库地址：



```
#使用 git clone 命令克隆远程仓库，前提要知道Git库地址
$ git clone git@github.com:michaelliao/gitskills.git
#默认使用SSH较快
```

3.创建分支

提要：

查看分支： `git branch`

创建分支： `git branch <name>`

切换分支： `git checkout <name>` 或者 `git switch <name>`

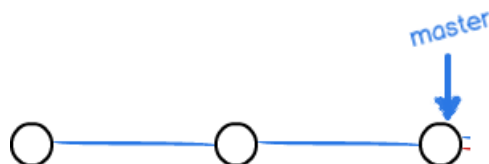
创建+切换分支： `git checkout -b <name>` 或者 `git switch -c <name>`

合并某分支到当前分支： `git merge <name>`

删除分支： `git branch -d <name>`

截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即 `master` 分支。`HEAD` 严格来说不是指向提交，而是指向 `master`，`master` 才是指向提交的，所以，`HEAD` 指向的就是当前分支。

只有一条主分支：



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：

```
$ git checkout -b dev # $ git switch -c dev
Switched to a new branch 'dev'
```

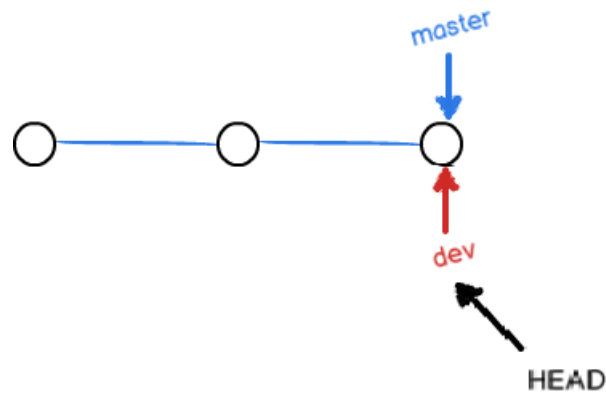
#查看当前分支

```
$ git branch
```

```
* dev
```

```
master
```

`git branch`命令会列出所有分支，当前分支前面会标一个*号。



新分支添加修改:

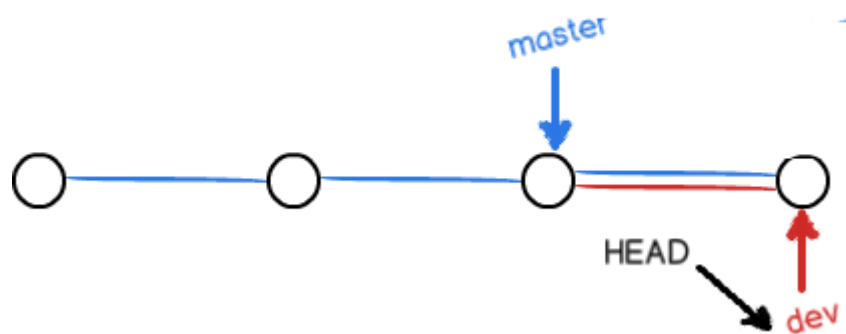
对readme.txt文件添加: Creating a new branch is quick.

```
$ git add readme.txt
$ git commit -m "branch test"
[dev b17d20e] branch test
1 file changed, 1 insertion(+)
```

#切换到master分支后可以发现没有刚才提交的修改。

```
# $ git checkout master
```

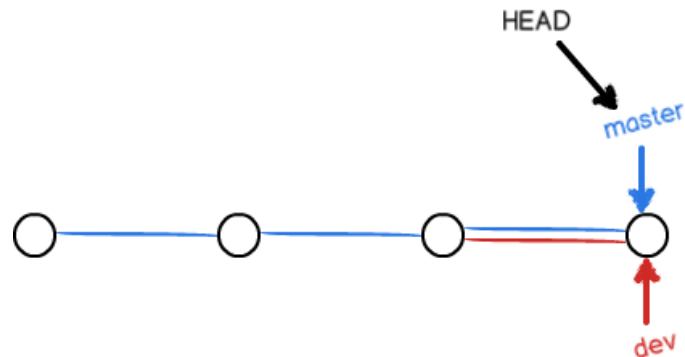
```
# Switched to branch 'master'
```



合并:

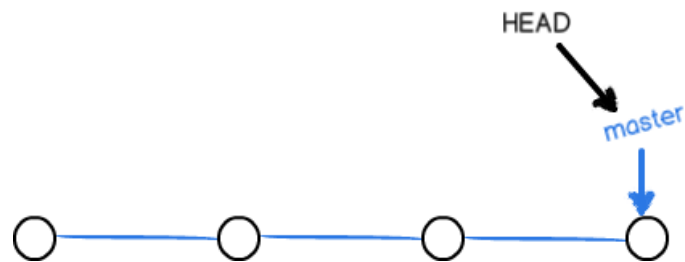
`git merge` 命令用于合并指定分支到当前分支

```
$ git merge dev
Updating d46f35e..b17d20e
Fast-forward
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```



删除dev分支：

```
$ git branch -d dev
Deleted branch dev (was b17d20e).
```



4.解决冲突

用带参数的 `$ git log --graph --pretty=oneline --abbrev-commit` 也可以看到分支的合并情况：

准备新的 `feature1` 分支，继续我们的新分支开发：

```
$ git checkout -b feature1
Switched to a new branch 'feature1'
```

修改 `readme.txt` 最后一行，改为：

```
Creating a new branch is quick AND simple.
```

在 `feature1` 分支上提交：

```
$ git add readme.txt

$ git commit -m "AND simple"
[feature1 14096d0] AND simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

切换到 `master` 分支：

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
```

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

在 master 分支上把 readme.txt 文件的最后一行改为：

```
Creating a new branch is quick & simple.
```

提交：

```
$ git add readme.txt
$ git commit -m "& simple"
[master 5dc6824] & simple
1 file changed, 1 insertion(+), 1 deletion(-)
```

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

```
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

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

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
(use "git push" to publish your local commits)

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:   readme.txt

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

我们可以直接查看readme.txt的内容：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
<<<<<<< HEAD
Creating a new branch is quick & simple.
=====
Creating a new branch is quick AND simple.
>>>>>>> feature1
```

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

```
Creating a new branch is quick and simple.
```

再提交：

```
$ git add readme.txt
$ git commit -m "conflict fixed"
[master cf810e4] conflict fixed
```

最后，删除 feature1 分支：

```
$ git branch -d feature1
Deleted branch feature1 (was 14096d0).
```

5.分支管理策略

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

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

下面我们实战一下 `--no-ff` 方式的 `git merge`：

首先，仍然创建并切换 `dev` 分支：

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

修改readme.txt文件，并提交一个新的commit：

```
$ git add readme.txt
$ git commit -m "add merge"
[dev f52c633] add merge
1 file changed, 1 insertion(+)
```

现在，我们切换回 `master`：

```
$ git checkout master
Switched to branch 'master'
```

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

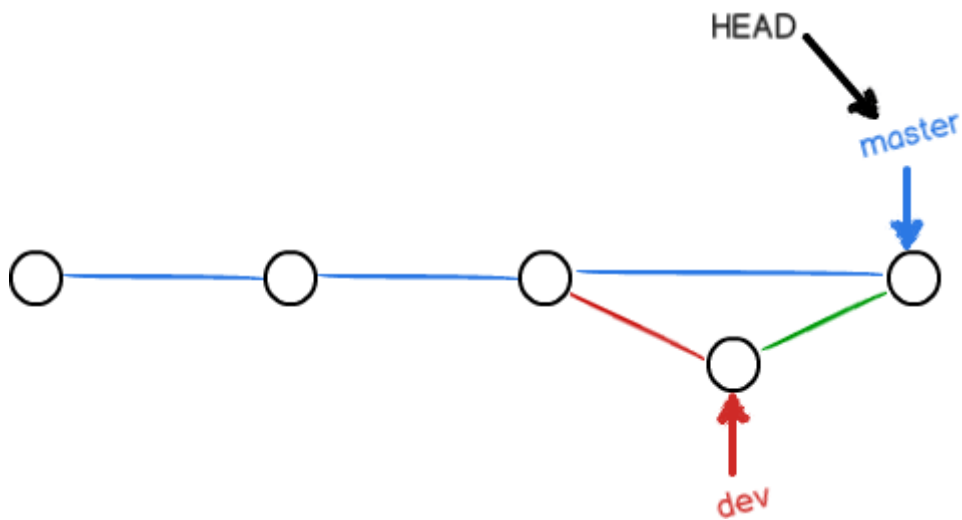

```
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

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

合并后，我们用 `git log` 看看分支历史：

```
$ git log --graph --pretty=oneline --abbrev-commit
* e1e9c68 (HEAD -> master) merge with no-ff
|\
| * f52c633 (dev) add merge
|/
* cf810e4 conflict fixed
...
```

可以看到，不使用 `Fast forward` 模式，merge后就像这样：



6. Bug分支

当有手头工作修改还没提交，但要修复bug时：

1. 可以用 `git stash` 命令保留工作现场
2. 切换到bug分支修复bug
3. 恢复工作现场
 - `git stash list` 查看保存的工作现场
 - `git stash apply <commit>` 恢复工作现场，但会保留工作现场记录
 - `git stash pop` 恢复工作现场，并删除工作现场记录
4. `git cherry-pick <commit>` 为不同分支提交同一次改动，此命令会产生一个新的commit

演示：

dev未完成工作

```
$ git status
On branch dev
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   hello.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   readme.txt
```

`git stash` 保留工作现场

```
$ git stash
Saved working directory and index state WIP on dev: f52c633 add merge
```

在master分支创建临时分支修改bug

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
  (use "git push" to publish your local commits)

$ git checkout -b issue-101
Switched to a new branch 'issue-101'
```

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

```
$ git add readme.txt
$ git commit -m "fix bug 101"
[issue-101 4c805e2] fix bug 101
 1 file changed, 1 insertion(+), 1 deletion(-)
```

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

```
$ git switch master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
  (use "git push" to publish your local commits)

$ git merge --no-ff -m "merged bug fix 101" issue-101
Merge made by the 'recursive' strategy.
  readme.txt | 2 +-
  1 file changed, 1 insertion(+), 1 deletion(-)
$ git branch -d issue-101

#git branch -D issue-101 强行删除未合并过的分支, 此操作将丢失掉被删除分支的修改
```

修复完bug回到dev分支

```
$ git switch dev
Switched to branch 'dev'

$ git status
On branch dev
nothing to commit, working tree clean
```

恢复工作现场

```
$ git stash pop
On branch dev
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   hello.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   readme.txt

Dropped refs/stash@{0} (5d677e2ee266f39ea296182fb2354265b91b3b2a)
```

为dev分支提交之前的bug修复

```
$ git branch
* dev
  master
$ git cherry-pick 4c805e2
[master 1d4b803] fix bug 101
1 file changed, 1 insertion(+), 1 deletion(-)
```

7.多人协作

用 `git remote` 查看远程库信息

```
$ git remote
origin
```

用 `git remote -v` 显示更详细的信息

```
$ git remote -v
origin  git@github.com:michaelliao/learngit.git (fetch)#可抓取
origin  git@github.com:michaelliao/learngit.git (push)#可推送
```

推送分支

```
$ git push origin master
$ git push origin dev
#如果推送失败, 先用git pull抓取远程的新提交
```

抓取分支

```
$ git pull
```

#因为第一次从远端克隆时只能看到本地的master分支，所有得创建远程origin的dev分支到本地：

```
$ git checkout -b dev origin/dev
```

#删除远端分支

```
$ git branch -r -d origin/branch-name
```

```
$ git push origin :branch-name
```

因此，多人协作的工作模式通常是这样：

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

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

#本地dev分支与远程origin/dev分支的链接

```
$ git branch --set-upstream-to=origin/dev dev
```

8.*Rebase(未整理)

在上一节我们看到了，多人在同一个分支上协作时，很容易出现冲突。即使没有冲突，后push的童鞋不得不先pull，在本地合并，然后才能push成功。

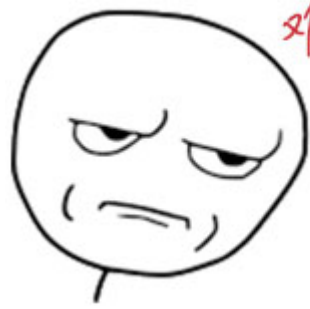
每次合并再push后，分支变成了这样：

```
$ git log --graph --pretty=oneline --abbrev-commit
* d1be385 (HEAD -> master, origin/master) init hello
* e5e69f1 Merge branch 'dev'
|\
| * 57c53ab (origin/dev, dev) fix env conflict
| |\
| | * 7a5e5dd add env
| * | 7bd91f1 add new env
| |/\
* | 12a631b merged bug fix 101
|\ \
| * | 4c805e2 fix bug 101
|/ /
* | e1e9c68 merge with no-ff
|\ \
| |/\
| * f52c633 add merge
|/
* cf810e4 conflict fixed
```

总之看上去很乱，有强迫症的童鞋会问：为什么Git的提交历史不能是一条干净的直线？

其实是可以做到的！

Git有一种称为rebase的操作，有人把它翻译成“变基”。



难道有什么特殊含义？

先不要随意展开想象。我们还是从实际问题出发，看看怎么把分叉的提交变成直线。

在和远程分支同步后，我们对 `hello.py` 这个文件做了两次提交。用 `git log` 命令看看：

```
$ git log --graph --pretty=oneline --abbrev-commit
* 582d922 (HEAD -> master) add author
* 8875536 add comment
* d1be385 (origin/master) init hello
* e5e69f1 Merge branch 'dev'
|\
| * 57c53ab (origin/dev, dev) fix env conflict
| |\
| | * 7a5e5dd add env
| * | 7bd91f1 add new env
...
```

注意到Git用 `(HEAD -> master)` 和 `(origin/master)` 标识出当前分支的HEAD和远程origin的位置分别是 `582d922 add author` 和 `d1be385 init hello`，本地分支比远程分支快两个提交。

现在我们尝试推送本地分支：

```
$ git push origin master
To github.com:michaelliao/learngit.git
! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

很不幸，失败了，这说明有人先于我们推送了远程分支。按照经验，先pull一下：

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:michaelliao/learngit
   d1be385..f005ed4  master      -> origin/master
* [new tag]         v1.0        -> v1.0
Auto-merging hello.py
Merge made by the 'recursive' strategy.
 hello.py | 1 +
 1 file changed, 1 insertion(+)
```

再用 `git status` 看看状态:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

加上刚才合并的提交, 现在我们本地分支比远程分支超前3个提交。

用 `git log` 看看:

```
$ git log --graph --pretty=oneline --abbrev-commit
* e0ea545 (HEAD -> master) Merge branch 'master' of
github.com:michaelliao/learngit
|\
| * f005ed4 (origin/master) set exit=1
* | 582d922 add author
* | 8875536 add comment
|/
* d1be385 init hello
...
```

对强迫症童鞋来说, 现在事情有点不对头, 提交历史分叉了。如果现在把本地分支push到远程, 有没有问题?

有!

什么问题?

不好看!

有没有解决方法?

有!

这个时候, rebase就派上了用场。我们输入命令 `git rebase` 试试:

```
$ git rebase
First, rewinding head to replay your work on top of it...
Applying: add comment
Using index info to reconstruct a base tree...
M   hello.py
Falling back to patching base and 3-way merge...
Auto-merging hello.py
Applying: add author
Using index info to reconstruct a base tree...
M   hello.py
Falling back to patching base and 3-way merge...
Auto-merging hello.py
```

输出了一大堆操作, 到底是啥效果? 再用 `git log` 看看:

```
$ git log --graph --pretty=oneline --abbrev-commit
* 7e61ed4 (HEAD -> master) add author
* 3611cfe add comment
* f005ed4 (origin/master) set exit=1
* d1be385 init hello
...
```

原本分叉的提交现在变成一条直线了！这种神奇的操作是怎么实现的？其实原理非常简单。我们注意观察，发现Git把我们本地的提交“挪动”了位置，放到了 f005ed4 (origin/master) set exit=1 之后，这样，整个提交历史就成了一条直线。rebase操作前后，最终的提交内容是一致的，但是，我们本地的commit修改内容已经变化了，它们的修改不再基于 d1be385 init hello，而是基于 f005ed4 (origin/master) set exit=1，但最后的提交 7e61ed4 内容是一致的。

这就是rebase操作的特点：把分叉的提交历史“整理”成一条直线，看上去更直观。缺点是本地的分叉提交已经被修改过了。

最后，通过push操作把本地分支推送到远程：

```
Mac:~/learngit michael$ git push origin master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 576 bytes | 576.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To github.com:michaelliao/learngit.git
    f005ed4..7e61ed4  master -> master
```

再用 git log 看看效果：

```
$ git log --graph --pretty=oneline --abbrev-commit
* 7e61ed4 (HEAD -> master, origin/master) add author
* 3611cfe add comment
* f005ed4 set exit=1
* d1be385 init hello
...
```

四、标签管理

1. 创建标签

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

2. 操作标签

- 命令 `git push origin <tagname>` 可以推送一个本地标签；
- 命令 `git push origin --tags` 可以推送全部未推送过的本地标签；
- 命令 `git tag -d <tagname>` 可以删除一个本地标签；

- 命令 `git push origin :refs/tags/<tagname>` 可以删除一个远程标签。
- 另一种删除远程标签的方式：

```
git push origin --delete <tagname>
```

3. 检出标签

使用 `git checkout` 命令检出标签，但会使仓库处于“分离头指针”状态。

在“分离头指针”状态下，如果你做了某些更改然后提交它们，标签不会发生变化，但你的新提交将不属于任何分支，并且将无法访问，除非通过确切的提交哈希才能访问。

如果想修复就版本中的错误，通常需要创建一个新分支，然后在此分支检出标签：

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

如果在这之后又进行了一次提交，`version2` 分支就会因为这个改动向前移动，此时它就会和 `v2.0.0` 标签稍微有些不同，这时就要当心了。

五、自定义Git

1. 忽略特殊文件

忽略文件的原則：

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

忽略文件的部分语法：

```
#排除某文件
filename.suffix

# 排除所有.开头的隐藏文件：
.*

# 排除所有.class文件：
*.class

# 不排除.gitignore和App.class：
!.gitignore
!App.class
```

与忽略文件相关的git命令：

```
#某文件被排除时，可强行添加
$ git add -f App.class

#检查那条忽略文件规则排除了该文件
$ git check-ignore -v App.class
.gitignore:3:*.class    App.class # .ignore 文件的第三行规则排除了该文件
```


2.配置别名

将 `git status` 配置别名 `git st`

```
$ git config --global alias.st status
```

记一段日后可能会用到的别名配置

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

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

- 而当前用户的Git配置文件放在用户主目录下的一个隐藏文件 `.gitconfig` 中
- 只对当前仓库起作用的配置文件放在仓库文件夹 `.git/config` 文件中。

六、SourceTree

Git图形化界面工具。

七、Git参考文档

官方文档: [Git - Book \(git-scm.com\)](http://git-scm.com)