**Git使用教程**

**$ git remote add origin https://github.com/lyon-fan/Python-Repository.git**

**$ git push -u origin master**

**生于理想 死于现实 但 心中很有天堂**

**目录**

[1. git下载安装 1-1](#_Toc518770119)

[2. why git 2-1](#_Toc518770120)

[2.1 什么时候需要用 Git? 2-1](#_Toc518770121)

[2.2 什么文件可以被 Git 管理? 2-1](#_Toc518770122)

[2.3 什么文件不能被 Git 管理? 2-1](#_Toc518770123)

[3. Git 安装 3-2](#_Toc518770124)

[3.1 Linux 系统 ¶ 3-2](#_Toc518770125)

[3.2 MacOS 系统 3-2](#_Toc518770126)

[3.3 Windows 系统 3-2](#_Toc518770127)

[4. 第一个版本库 Repository 4-2](#_Toc518770128)

[4.1 创建版本库 (init) 4-2](#_Toc518770129)

[4.2 添加文件管理 (add) 4-3](#_Toc518770130)

[4.3 提交改变 (commit) ¶ 4-4](#_Toc518770131)

[5. 记录修改 (log & diff) 5-4](#_Toc518770132)

[5.1 修改记录 log 5-4](#_Toc518770133)

[5.2 查看 unstaged 5-6](#_Toc518770134)

[5.3 查看 staged (--cached) 5-6](#_Toc518770135)

[5.4 查看 staged & unstaged (HEAD) 5-6](#_Toc518770136)

[6. 回到从前 (reset) 6-7](#_Toc518770137)

[6.1 修改已 commit 的版本 6-7](#_Toc518770138)

[6.2 reset 回到 add 之前 6-8](#_Toc518770139)

[6.3 reset 回到 commit 之前 6-8](#_Toc518770140)

[7. 回到从前 (checkout 针对单个文件) 7-10](#_Toc518770141)

[7.1 改写文件 checkout 7-10](#_Toc518770142)

[8. 分支 (branch) 8-11](#_Toc518770143)

[8.1 分支 图例 8-11](#_Toc518770144)

[8.2 使用 branch 创建 dev 分支 8-12](#_Toc518770145)

[8.3 使用 checkout 创建 dev 分支 8-12](#_Toc518770146)

[8.4 在 dev 分支中修改 8-13](#_Toc518770147)

[8.5 将 dev 的修改推送到 master 8-13](#_Toc518770148)

[9. merge 分支冲突 9-14](#_Toc518770149)

[9.1 merge 分支冲突 9-14](#_Toc518770150)

[10. rebase 分支冲突 10-16](#_Toc518770151)

[10.1 什么是 rebase 10-16](#_Toc518770152)

[10.2 使用 rebase 10-17](#_Toc518770153)

[11. 临时修改 (stash) 11-19](#_Toc518770154)

[11.1 暂存修改 11-19](#_Toc518770155)

[11.2 做其它任务 11-20](#_Toc518770156)

[11.3 恢复暂存 11-21](#_Toc518770157)

[12. Github 在线代码管理 12-21](#_Toc518770158)

[12.1 建立 github 版本库 12-21](#_Toc518770159)

[12.2 连接本地版本库 12-22](#_Toc518770160)

[12.3 推送修改 12-23](#_Toc518770161)

# 命令

## 克隆(Clone)源代码到本地

Git clone [URL]

## 提交代码

git add .

git commit -m "edit file "

git push -u origin master

## 创建github

# git下载安装

请自己度娘git官网，下载，安装。

教程如下：

<https://morvanzhou.github.io/tutorials/others/git/>

# why git

Git 是一个分布式版本控制系统. 它的灵活性, 优越性使得它从2005年发布以来. 获得了越来越多的使用和支持.

## 什么时候需要用 Git?

* 当你已经成为码农, 或者已经在成为码农的路上;
* 当你觉得代码太多;
* 当你已经开始用日期或版本号命名的代码文件的时候.

你是想要这样的文件夹:

[](https://morvanzhou.github.io/static/results/git/1-1-1.png)

还是这样的文件夹:

[](https://morvanzhou.github.io/static/results/git/1-1-2.png)

## 什么文件可以被 Git 管理?

* 文本文件 (.txt) 等;
* 脚本文件 (.py) 等;
* 各种基于文本信息的文件.

## 什么文件不能被 Git 管理?

* 图片文件 (.jpg) 等;
* MS word (.doc) 等.

# Git 安装

## Linux 系统 [¶](https://morvanzhou.github.io/tutorials/others/git/1-2-install/#Linux-系统)

使用 Linux 的朋友们, 大家肯定都很熟悉怎么在 Linux 上装东西, 所以只要找到你的 Terminal, 并输入以下指令就可以啦:

# 如果你的 Linux 是 Ubuntu:

$ sudo apt-get install git-all

# 如果你的 Linux 是 Fedora:

$ sudo yum install git-all

## MacOS 系统

Git 已经为 Mac 用户做好了一个安装包, 我们可以在这里下载并安装 <https://git-scm.com/download/mac>

## Windows 系统

Git 也为 Windows 系统提供了简易的 .exe 安装包, 直接从这里下载并安装就可以了: <https://git-scm.com/download/win>

推荐使用默认安装参数, 然后一路下一步到底. 安装好之后, 在你的所有程序中, 将会出现 git 的文件夹, 而且里面会有一个程序叫做 git bash. 这个 git bash 是 git 在 Windows 上为了方便使用所设置的一个 Unix 的环境. 如果你是 Windows 用户, 之后的教程你也能用这个来学习使用 git.

# 第一个版本库 Repository

## 创建版本库 (init)

我们先要确定要把哪个文件夹里的文件进行管理. 比如说我桌面上的一个叫 gitTUT 的文件夹. 然后在 Terminal (Windows 的 git bash) 中把当前目录调到这个文件夹 gitTUT, 我的做法是这样:

$ cd ~/Desktop/gitTUT

为了更好地使用 git, 我们同时也记录每一个施加修改的人. 这样人和修改能够对应上. 所以我们在 git 中添加用户名 user.name 和 用户 email user.email:

$ git config --global user.name "lyon-fan"

$ git config --global user.email [fanhongliang03@163.com](mailto:fanhongliang03@163.com)

查看

$ git config user.name

$ git config user.email

然后我们就能在这个文件夹中建立 git 的管理文件了:

$ git init

# Initialized empty Git repository in /Users/MorvanZhou/Desktop/gitTUT/.git/

因为这个文件夹中还没有任何的文件, 它返回出来一句话告诉我们已经建立了一个空的 git 管理库.

## 添加文件管理 (add)

通常我们执行 $ ls 就能看到文件夹中的所有文件, 不过 git 创建的管理库文件 .git 是被隐藏起来的. 所以我们要执行这一句才能看到被隐藏的文件:

$ ls -a

# . .. .git

建立一个新的 1.py 文件:

$ touch 1.py

现在我们能用 status 来查看版本库的状态:

$ git status

# 输出

On branch master # 在 master 分支

Initial commit

Untracked files:

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

1.py # 1.py 文件没有被加入版本库 (unstaged)

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

现在 1.py 并没有被放入版本库中 (unstaged), 所以我们要使用 add 把它添加进版本库 (staged):

$ git add 1.py

# 再次查看状态 status

$ git status

# 输出

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: 1.py # 版本库已识别 1.py (staged)

如果想一次性添加文件夹中所有未被添加的文件, 可以使用这个:

$ git add .

## 提交改变 (commit) [¶](https://morvanzhou.github.io/tutorials/others/git/2-1-repository/#提交改变-(commit))

我们已经添加好了 1.py 文件, 最后一步就是提交这次的改变, 并在 -m 自定义这次改变的信息:

$ git commit -m "create 1.py"

# 输出

[master (root-commit) 6bd231e] create 1.py

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

create mode 100644 1.py

整个上述过程可以被这张 git 官网上的流程图直观地表现:



# 记录修改 (log & diff)

## 修改记录 log

之前我们以Morvan Zhou 的名义对版本库进行了一次修改, 添加了一个 1.py 的文件. 接下来我们就来查看版本库的些施工的过程. 可以看到在 Author 那已经有我的名字和 email 信息了.

$ git log

# 输出

commit 13be9a7bf70c040544c6242a494206f240aac03c

Author: Morvan Zhou <mz@email.com>

Date: Tue Nov 29 00:06:47 2016 +1100

create 1.py # 这是我们上节课记录的修改信息

如果我们对1.py文件进行一次修改, 添加这行代码:

a = 1

然后我们就能在 status 中看到修改还没被提交的信息了.

$ git status

# 输出

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: 1.py # 这里显示有一个修改还没被提交

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

所以我们先把这次修改添加 (add) 到可被提交 (commit) 的状态, 然后再提交 (commit) 这次的修改:

$ git add 1.py

$ git commit -m "change 1"

# 输出

[master fb51216] change 1

1 file changed, 1 insertion(+) # 提示文件有一处添加

再次查看 log, 现在我们就能看到 create 1.py 和 change 1 这两条修改信息了. 而且做出这两条 commit 的 ID, 修改的 Author, 修改 Date 也被显示在上面.

$ git log

# 输出

commit fb51216b081e00db3996e14edf8ff080fab1980a

Author: Morvan Zhou <mz@email.com>

Date: Tue Nov 29 00:24:50 2016 +1100

change 1

commit 13be9a7bf70c040544c6242a494206f240aac03c

Author: Morvan Zhou <mz@email.com>

Date: Tue Nov 29 00:06:47 2016 +1100

create 1.py

如果删除一部分代码, 也会被记录上, 比如把 a = 1 改成 a = 2, 再添加一个 b = 1.

a = 2

b = 1

## 查看 unstaged

果想要查看这次还没 add (unstaged) 的修改部分 和上个已经 commit 的文件有何不同, 我们将使用 $ git diff:

$ git diff

# 输出

diff --git a/1.py b/1.py

index 1337a53..ff7c36c 100644

--- a/1.py

+++ b/1.py

@@ -1 +1,2 @@

-a = 1 # 删除了 a = 1

+a = 2 # 添加了 a = 2

+b = 1 # 添加了 b = 1

## 查看 staged (--cached)

如果你已经 add 了这次修改, 文件变成了 “可提交状态” (staged), 我们可以在 diff 中添加参数 --cached 来查看修改:

$ git add . # add 全部修改文件

$ git diff --cached

# 输出

diff --git a/1.py b/1.py

index 1337a53..ff7c36c 100644

--- a/1.py

+++ b/1.py

@@ -1 +1,2 @@

-a = 1

+a = 2

+b = 1

## 查看 staged & unstaged (HEAD)

还有种方法让我们可以查看 add 过 (staged) 和 没 add (unstaged) 的修改, 比如我们再修改一下 1.py 但不 add:

a = 2

b = 1

c = b

目前 a = 2 和 b = 1 已被 add, c = b 是新的修改, 还没被 add.

# 对比三种不同 diff 形式

$ git diff HEAD # staged & unstaged

@@ -1 +1,3 @@

-a = 1 # 已 staged

+a = 2 # 已 staged

+b = 1 # 已 staged

+c = b # 还没 add 去 stage (unstaged)

-----------------------

$ git diff # unstaged

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

a = 2 # 注: 前面没有 +

b = 1 # 注: 前面没有 +

+c = b # 还没 add 去 stage (unstaged)

-----------------------

$ git diff --cached # staged

@@ -1 +1,2 @@

-a = 1 # 已 staged

+a = 2 # 已 staged

+b = 1 # 已 staged

为了下节内容, 我们保持这次修改, 全部 add 变成 staged 状态, 并 commit.

$ git add .

$ git commit -m "change 2"

# 输出

[master 6cc6579] change 2

1 file changed, 3 insertions(+), 1 deletion(-)

# 回到从前 (reset)

## 修改已 commit 的版本

有时候我们总会忘了什么, 比如已经提交了 commit 却发现在这个 commit 中忘了附上另一个文件. 接下来我们模拟这种情况. 上节内容中, 我们最后一个 commit 是 change 2, 我们将要添加另外一个文件, 将这个修改也 commit 进 change 2. 所以我们复制 1.py 这个文件, 改名为 2.py. 并把 2.py 变成 staged, 然后使用 --amend 将这次改变合并到之前的 change 2 中.

$ git add 2.py

$ git commit --amend --no-edit # "--no-edit": 不编辑, 直接合并到上一个 commit

$ git log --oneline # "--oneline": 每个 commit 内容显示在一行

# 输出

904e1ba change 2 # 合并过的 change 2

c6762a1 change 1

13be9a7 create 1.py

## reset 回到 add 之前

有时我们添加 add 了修改, 但是又后悔, 并想补充一些内容再 add. 这时, 我们有一种方式可以回到 add 之前. 比如在 1.py 文件中添加这一行:

d = 3

然后 add 去 staged 再返回到 add 之前:

$ git add 1.py

$ git status -s # "-s": status 的缩写模式

# 输出

M 1.py # staged

-----------------------

$ git reset 1.py

# 输出

Unstaged changes after reset:

M 1.py

-----------------------

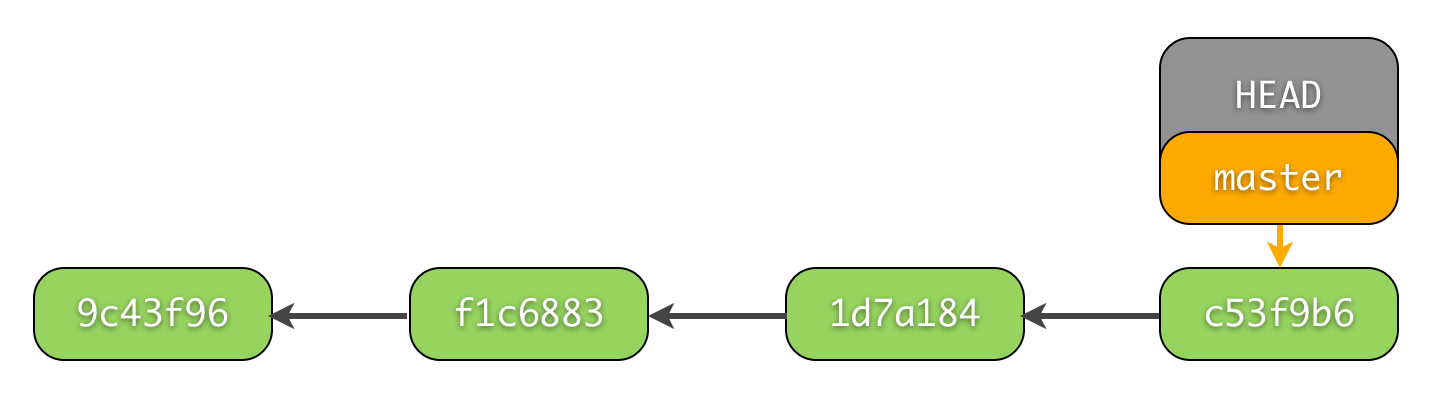
$ git status -s

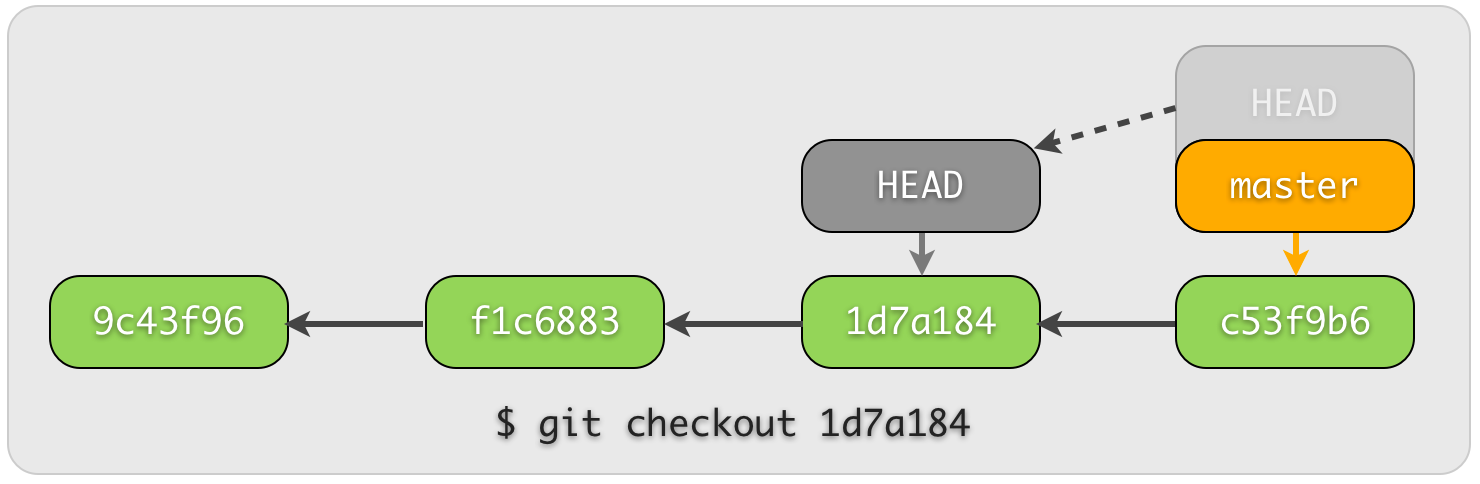
# 输出

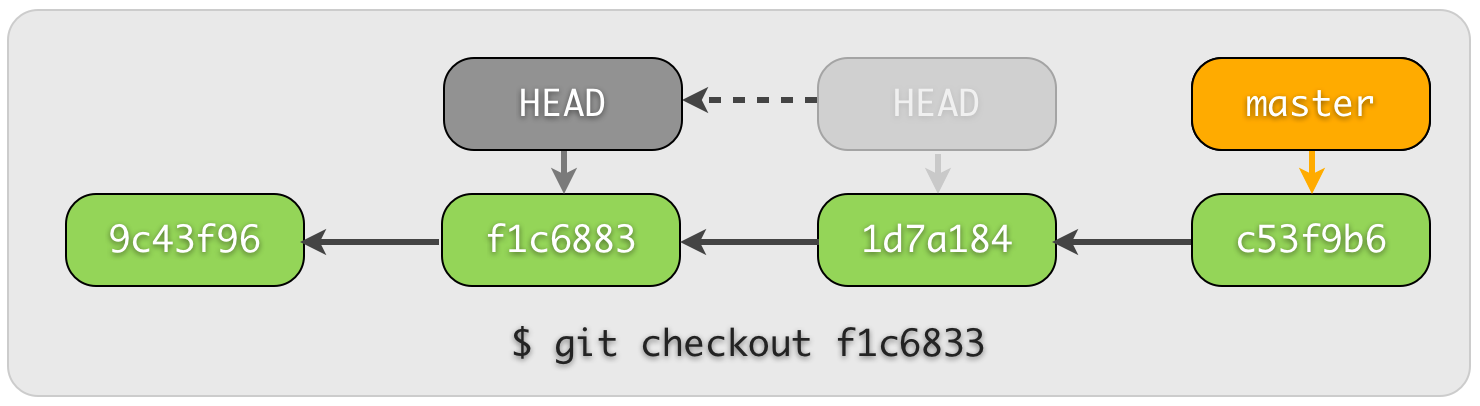
M 1.py # unstaged

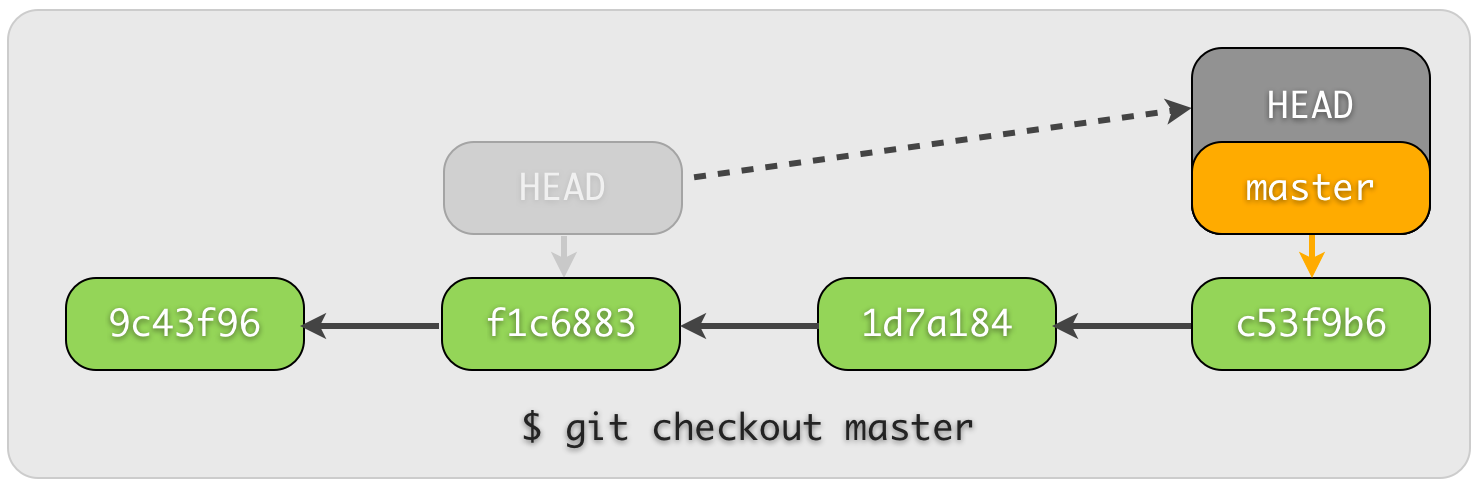
## reset 回到 commit 之前

在穿梭到过去的 commit 之前, 我们必须了解 git 是如何一步一步累加更改的.









每个 commit 都有自己的 id 数字号, HEAD 是一个指针, 指引当前的状态是在哪个 commit. 最近的一次 commit 在最右边, 我们如果要回到过去, 就是让 HEAD 回到过去并 reset 此时的 HEAD 到过去的位置.

# 不管我们之前有没有做了一些 add 工作, 这一步让我们回到 上一次的 commit

$ git reset --hard HEAD

# 输出

HEAD is now at 904e1ba change 2

-----------------------

# 看看所有的log

$ git log --oneline

# 输出

904e1ba change 2

c6762a1 change 1

13be9a7 create 1.py

-----------------------

# 回到 c10ea64 change 1

# 方式1: "HEAD^"

$ git reset --hard HEAD^

# 方式2: "commit id"

$ git reset --hard c6762a1

-----------------------

# 看看现在的 log

$ git log --oneline

# 输出

c6762a1 change 1

13be9a7 create 1.py

怎么 change 2 消失了!!! 还有办法挽救消失的 change 2 吗? 我们可以查看 $ git reflog 里面最近做的所有 HEAD 的改动, 并选择想要挽救的 commit id:

$ git reflog

# 输出

c6762a1 HEAD@{0}: reset: moving to c6762a1

904e1ba HEAD@{1}: commit (amend): change 2

0107760 HEAD@{2}: commit: change 2

c6762a1 HEAD@{3}: commit: change 1

13be9a7 HEAD@{4}: commit (initial): create 1.py

重复 reset 步骤就能回到 commit (amend): change 2 (id=904e1ba)这一步了:

$ git reset --hard 904e1ba

$ git log --oneline

# 输出

904e1ba change 2

c6762a1 change 1

13be9a7 create 1.py

我们又再次奇迹般的回到了 change 2.

# 回到从前 (checkout 针对单个文件)

## 改写文件 checkout

其实 checkout 最主要的用途并不是让单个文件回到过去, 我们之后会继续讲 checkout 在分支 branch 中的应用, 这一节主要讲 checkout 让文件回到过去.

我们现在的版本库中有两个文件:

- gitTUT

- 1.py

- 2.py

我们仅仅要对 1.py 进行回到过去操作, 回到 c6762a1 change 1 这一个 commit. 使用 checkout + id c6762a1 + -- + 文件目录 1.py, 我们就能将 1.py 的指针 HEAD 放在这个时刻 c6762a1:

$ git log --oneline

# 输出

904e1ba change 2

c6762a1 change 1

13be9a7 create 1.py

---------------------

$ git checkout c6762a1 -- 1.py

这时 1.py 文件的内容就变成了:

a = 1

我们在 1.py 加上一行内容 # I went back to change 1 然后 add 并 commit 1.py:

$ git add 1.py

$ git commit -m "back to change 1 and add comment for 1.py"

$ git log --oneline

# 输出

47f167e back to change 1 and add comment for 1.py

904e1ba change 2

c6762a1 change 1

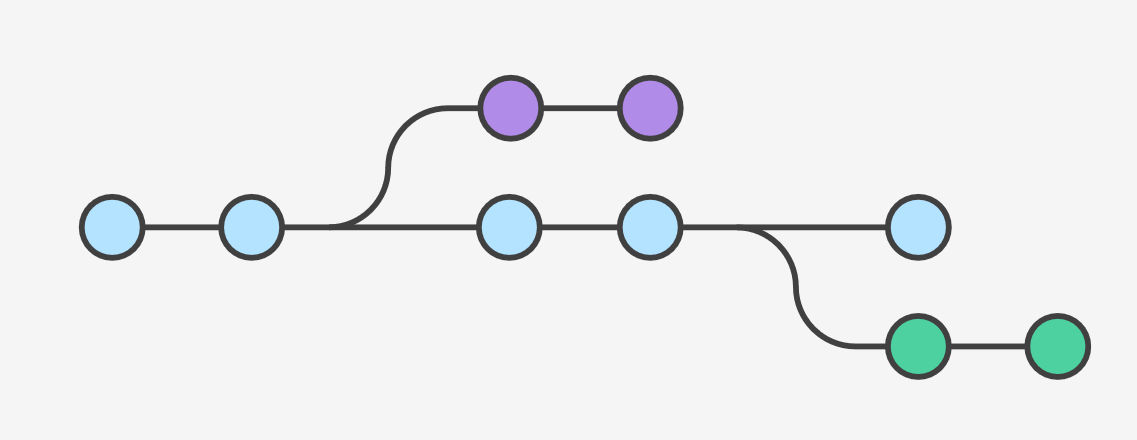
13be9a7 create 1.py

可以看出, 不像 reset 时那样, 我们的 change 2 并没有消失, 但是 1.py 却已经回去了过去, 并改写了未来.

# 分支 (branch)

## 分支 图例

之前我们说编辑的所有改变都是在一条主分支 master 上进行的. 通常我们会把 master 当作最终的版本, 而开发新版本或者新属性的时候, 在另外一个分支上进行, 这样就能使开发和使用互不干扰了.



## 使用 branch 创建 dev 分支

我们之前的文件当中, 仅仅只有一条 master 分支, 我们可以通过 --graph 来观看分支:

$ git log --oneline --graph

# 输出

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

## 使用 checkout 创建 dev 分支

接着我们建立另一个分支 dev, 并查看所有分支:

$ git branch dev # 建立 dev 分支

$ git branch # 查看当前分支

# 输出

dev

\* master # \* 代表了当前的 HEAD 所在的分支

当我们想把 HEAD 切换去 dev 分支的时候, 我们可以用到上次说的 checkout:

$ git checkout dev

# 输出

Switched to branch 'dev'

--------------------------

$ git branch

# 输出

\* dev # 这时 HEAD 已经被切换至 dev 分支

master

## 在 dev 分支中修改

使用 checkout -b + 分支名, 就能直接创建和切换到新建的分支:

$ git checkout -b dev

# 输出

Switched to a new branch 'dev'

--------------------------

$ git branch

# 输出

\* dev # 这时 HEAD 已经被切换至 dev 分支

master

## 将 dev 的修改推送到 master

dev 分支中的 1.py 和 2.py 和 master 中的文件是一模一样的. 因为当前的指针 HEAD 在 dev 分支上, 所以现在对文件夹中的文件进行修改将不会影响到 master 分支.

我们在 1.py 上加入这一行 # I was changed in dev branch, 然后再 commit:

$ git commit -am "change 3 in dev" # "-am": add 所有改变 并直接 commit

好了, 我们的开发板 dev 已经更新好了, 我们要将 dev 中的修改推送到 master 中, 大家就能使用到正式版中的新功能了.

首先我们要切换到 master, 再将 dev 推送过来.

$ git checkout master # 切换至 master 才能把其他分支合并过来

$ git merge dev # 将 dev merge 到 master 中

$ git log --oneline --graph

# 输出

\* f9584f8 change 3 in dev

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

要注意的是, 如果直接 git merge dev, git 会采用默认的 Fast forward 格式进行 merge, 这样 merge 的这次操作不会有 commit 信息. log 中也不会有分支的图案. 我们可以采取 --no-ff 这种方式保留 merge 的 commit 信息.

$ git merge --no-ff -m "keep merge info" dev # 保留 merge 信息

$ git log --oneline --graph

# 输出

\* c60668f keep merge info

|\

| \* f9584f8 change 3 in dev # 这里就能看出, 我们建立过一个分支

|/

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

# merge 分支冲突

## merge 分支冲突

今天的情况是这样, 想象不仅有人在做开发版 dev 的更新, 还有人在修改 master 中的一些 bug. 当我们再 merge dev 的时候, 冲突就来了. 因为 git 不知道应该怎么处理 merge 时, 在 master 和 dev 的不同修改.

当创建了一个分支后, 我们同时对两个分支都进行了修改.

比如在:

* master 中的 1.py 加上 # edited in master.
* dev 中的 1.py 加上 # edited in dev.

在下面可以看出在 master 和 dev 中不同的 commit:

# 这是 master 的 log

\* 3d7796e change 4 in master # 这一条 commit 和 dev 的不一样

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

-----------------------------

# 这是 dev 的 log

\* f7d2e3a change 3 in dev # 这一条 commit 和 master 的不一样

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

当我们想要 merge dev 到 master 的时候:

$ git branch

dev

\* master

-------------------------

$ git merge dev

# 输出

Auto-merging 1.py

CONFLICT (content): Merge conflict in 1.py

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

git 发现的我们的 1.py 在 master 和 dev 上的版本是不同的, 所以提示 merge 有冲突. 具体的冲突, git 已经帮我们标记出来, 我们打开 1.py 就能看到:

所以我们只要在 1.py 中手动合并一下两者的不同就 OK 啦. 我们将当前 HEAD (也就是master) 中的描述 和 dev 中的描述合并一下.

a = 1

# I went back to change 1

# edited in master and dev

然后再 commit 现在的文件, 冲突就解决啦.

$ git commit -am "solve conflict"

再来看看 master 的 log:

$ git log --oneline --graph

# 输出

\* 7810065 solve conflict

|\

| \* f7d2e3a change 3 in dev

\* | 3d7796e change 4 in master

|/

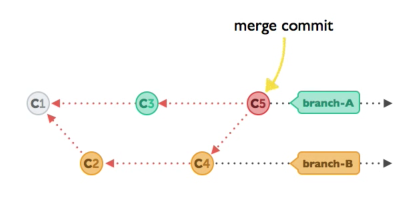
\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

回到这张图, 他也诠释了两个分支都有更改的时候的样子, 在这种情况下 merge, 我们就要使用上述的流程.

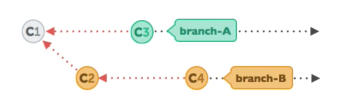


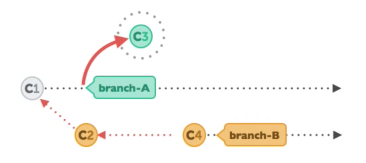
# rebase 分支冲突

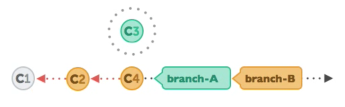
## 什么是 rebase

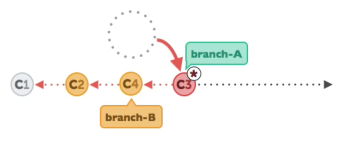
和上节内容一样, 不过我们今天来玩一个更高级的合并方式 rebase. 同样是合并 rebase 的做法和 merge 不一样.

假设共享的 branch 是 branch B, 而我在 branch A 上工作, 有一天我发现branch B已经有一些小更新, 我也想试试我的程序和这些小更新兼不兼容, 我也我想合并, 这时就可以用 rebase 来补充我的分支branch B的内容. 补充完以后, 和后面那张图的 merge 不同, 我还是继续在 C3 上工作, 不过此时的 C3 的本质却不一样了, 因为吸收了那些小更新. 所以我们用 C3' 来代替.

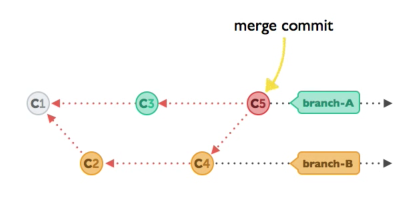








可以看出 rebase 改变了 C3 的属性, C3 已经不是从 C1 衍生而来的了. 这一点和 merge 不一样. merge 在合并的时候创建了一个新的 C5 commit. 这一点不同, 使得在共享分支中使用 rebase 变得危险. 如果是共享分支的历史被改写. 别人之前共享内容的 commit 就被你的 rebase 修改掉了.



所以需要强调的是 **!!! 只能在你自己的分支中使用 rebase, 和别人共享的部分是不能用 !!!**. 如果你不小心弄错了. 没事, 我们还能用在 [reset 这一节](https://morvanzhou.github.io/tutorials/others/git/3-1-reset/) 提到的 reflog 恢复原来的样子. 为了验证在共享分支上使用 rebase 的危险性, 我们在下面的例子中也验证一下.

## 使用 rebase

初始的版本库还是和上回一样, 在 master 和 dev 分支中都有自己的独立修改.

# 这是 master 的 log

\* 3d7796e change 4 in master # 这一条 commit 和 dev 的不一样

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

-----------------------------

# 这是 dev 的 log

\* f7d2e3a change 3 in dev # 这一条 commit 和 master 的不一样

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

当我们想要用 rebase 合并 dev 到 master 的时候:

$ git branch

# 输出

dev

\* master

-------------------------

$ git rebase dev

# 输出

First, rewinding head to replay your work on top of it...

Applying: change 3 in dev

Using index info to reconstruct a base tree...

M 1.py

Falling back to patching base and 3-way merge...

Auto-merging 1.py

CONFLICT (content): Merge conflict in 1.py

error: Failed to merge in the changes.

Patch failed at 0001 change 3 in dev

The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".

If you prefer to skip this patch, run "git rebase --skip" instead.

To check out the original branch and stop rebasing, run "git rebase --abort".

git 发现的我们的 1.py 在 master 和 dev 上的版本是不同的, 所以提示 merge 有冲突. 具体的冲突, git 已经帮我们标记出来, 我们打开 1.py 就能看到:

a = 1

# I went back to change 1

<<<<<<< f7d2e3a047be4624e83c1265a0946e2e8790f79c

# edited in dev

=======

# edited in master

>>>>>>> change 4 in master

这时 HEAD 并没有指向 master 或者 dev, 而是停在了 rebase 模式上:

$ git branch

\* (no branch, rebasing master) # HEAD 在这

dev

master

所以我们打开 1.py, 手动合并一下两者的不同.

a = 1

# I went back to change 1

# edited in master and dev

然后执行 git add 和 git rebase --continue 就完成了 rebase 的操作了.

$ git add 1.py

$ git rebase --continue

再来看看 master 的 log:

$ git log --oneline --graph

# 输出

\* c844cb1 change 4 in master # 这条 commit 原本的id=3d7796e, 所以 master 的历史被修改

\* f7d2e3a change 3 in dev # rebase 过来的 dev commit

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

**!! 注意 !!** 这个例子也说明了使用 rebase 要万分小心, 千万不要在共享的 branch 中 rebase, 不然就像上面那样, 现在 master 的历史已经被 rebase 改变了. master 当中别人提交的 change 4 就被你无情地修改掉了, 所以千万不要在共享分支中使用 rebase.

# 临时修改 (stash)

## 暂存修改

假设我们现在在 dev 分支上快乐地改代码:

$ git checkout dev

在 dev 中的 1.py 中加上一行 # feel happy, 然后老板的电话来了, 可是我还没有改进完这些代码. 所以我就用 stash 将这些改变暂时放一边.

$ git status -s

# 输出

M 1.py

------------------

$ git stash

# 输出

Saved working directory and index state WIP on dev: f7d2e3a change 3 in dev

HEAD is now at f7d2e3a change 3 in dev

-------------------

$ git status

# 输出

On branch dev

nothing to commit, working directory clean # 干净得很

## 做其它任务

然后我们建立另一个 branch 用来完成老板的任务:

$ git checkout -b boss

# 输出

Switched to a new branch 'boss' # 创建并切换到 boss

然后苦逼地完成着老板的任务, 比如添加 # lovely boss 去 1.py. 然后 commit, 完成老板的任务.

$ git commit -am "job from boss"

$ git checkout master

$ git merge --no-ff -m "merged boss job" boss

merge 如果有冲突的话, 可以像[上次那样](https://morvanzhou.github.io/tutorials/others/git/4-2-merge-conflict/) 解决.

a = 1

# I went back to change 1

<<<<<<< HEAD

# edited in master and dev

=======

# edited in dev

# lovely boss

>>>>>>> boss

通过以下步骤来完成老板的任务, 并观看一下 master 的 log:

$ git commit -am "solve conflict"

$ git log --oneline --graph

\* 1536bea solve conflict

|\

| \* 27ba884 job from boss

\* | 2d1961f change 4 in master

|/

\* f7d2e3a change 3 in dev

\* 47f167e back to change 1 and add comment for 1.py

\* 904e1ba change 2

\* c6762a1 change 1

\* 13be9a7 create 1.py

## 恢复暂存

轻松了, 现在可以继续开心的在 dev 上刷代码了.

$ git checkout dev

$ git stash list # 查看在 stash 中的缓存

# 输出

stash@{0}: WIP on dev: f7d2e3a change 3 in dev

上面说明在 dev 中, 我们的确有 stash 的工作. 现在可以通过 pop 来提取这个并继续工作了.

$ git stash pop

# 输出

On branch dev

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: 1.py

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

Dropped refs/stash@{0} (23332b7edc105a579b09b127336240a45756a91c)

----------------------

$ git status -s

# 输出

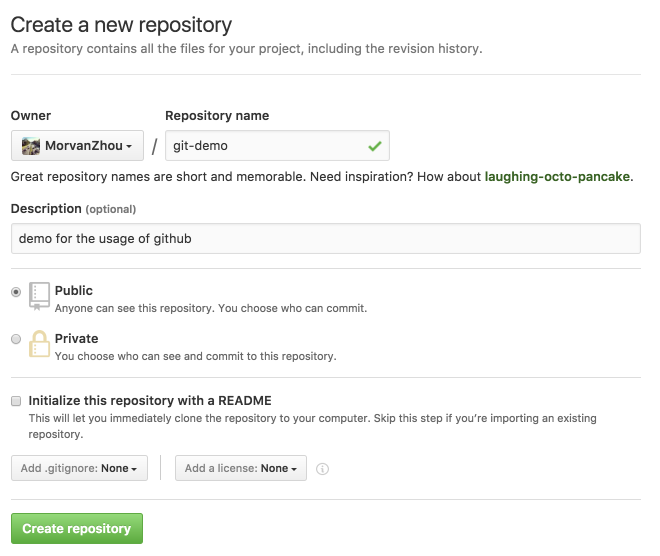
M 1.py # 和最开始一样了

# Github 在线代码管理

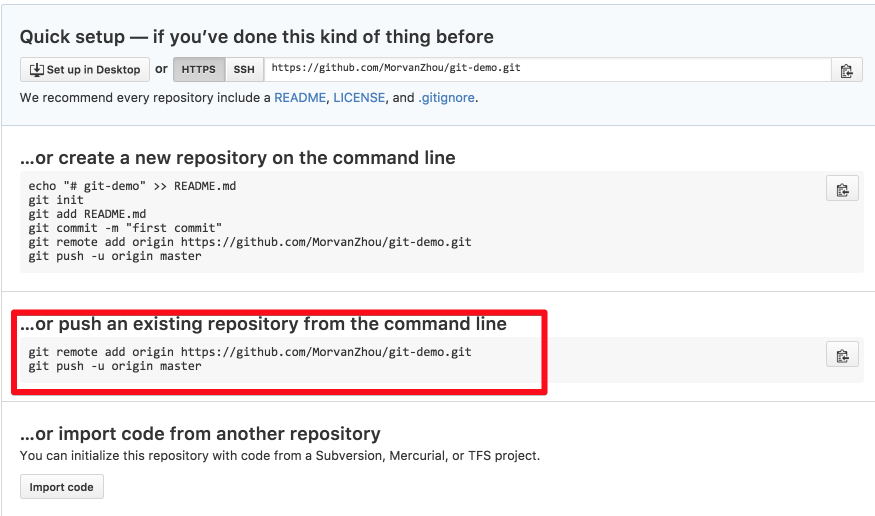
## 建立 github 版本库

在 [github](https://github.com/) 注册一个 github 账户, 这个不用我多说, 大家都知道注册.

然后添加你的一个 online 版本库 repository:

[](https://morvanzhou.github.io/static/results/git/5-1-1.png)

添加好了以后, 会出现下面的介绍, 你可以选择红框里的代码链接上你的本地版本库.

[](https://morvanzhou.github.io/static/results/git/5-1-2.png)

## 连接本地版本库

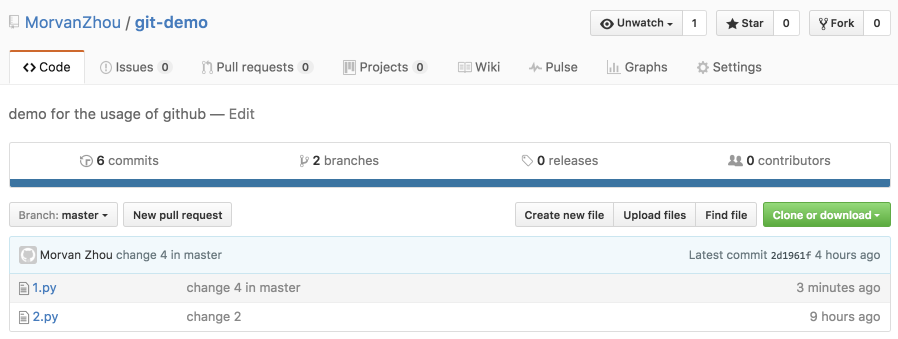
使用这节内容的初始例子文件, 然后将本地的版本库推送到网上:

$ git remote add origin https://github.com/MorvanZhou/git-demo.git

$ git push -u origin master # 推送本地 master 去 origin

$ git push -u origin dev # 推送本地 dev 去 origin

现在网上就已经有了你推上去的版本库了.

[](https://morvanzhou.github.io/static/results/git/5-1-3.png)

你甚至能在这里观看之前有哪些 commit 和 commit 具体做了什么:

[](https://morvanzhou.github.io/static/results/git/5-1-4.png)

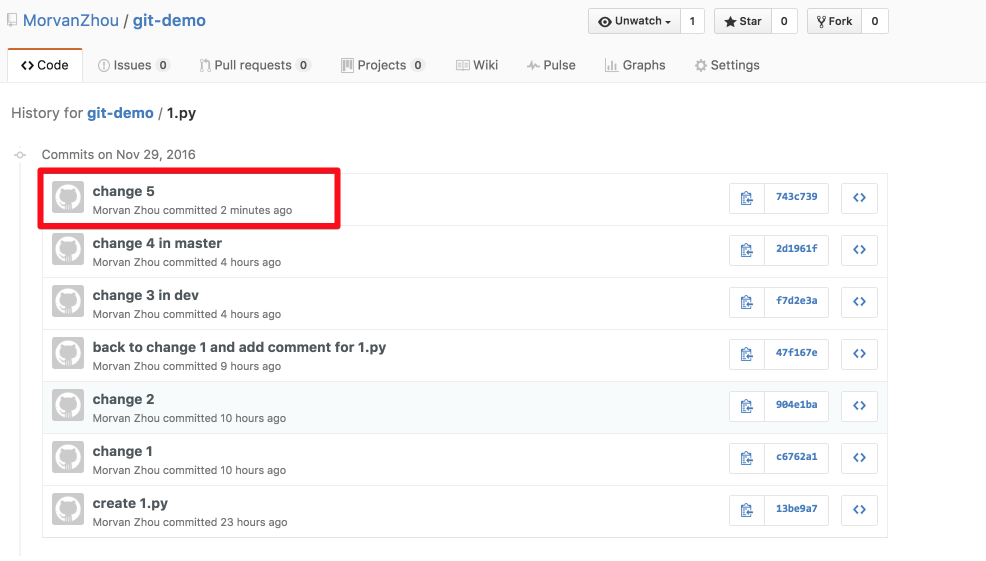
## 推送修改

如果在本地再进行修改, 比如在 1.py 文件中加上 # happy github, 然后 commit 并推上去:

$ git commit -am "change 5"

$ git push -u origin master

github 中就会查到这个:

[](https://morvanzhou.github.io/static/results/git/5-1-5.png)

## 克隆(Clone)源代码到本地

Git clone [URL]

## 提交代码

git add .

git commit -m "edit file "

git push -u origin master