

# Git Summary

## Git控制本地仓库local

### 配置相关用户信息

- 包括系统、全局（用户）、局部（当前使用仓库）信息
- 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` 参数的 `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` 参数让 Git 读写此文件。
- 3. 当前使用仓库的 Git 目录中的 `config` 文件 (就是 `.git/config`) :针对该repo。

### 全局

```
git config --global user.name "someone"
```

```
git config --global user.email "someone@someplace.com"//这里的email可以是虚构的
```

### 局部

```
git config user.name "someone"
```

```
git config user.email "someone@someplace.com"
```

### 查看配置信息

```
git config --list//查看所有配置
```

```
git config --global --list //查看全局的用户信息
```

```
git config --local -l//查看局部的用户信息
```

### 删除全局配置信息

```
git config --global --unset user.name "someone"
```

```
git config --global --unset user.email "someone@someplace.com"
```

### 编辑配置信息

```
git config --global --edit
```

### 配置参数详解

```
git config --global core.longpaths true           # 支持超长路径名
git config --global core.quotePath false          # 支持中文路径名
git config --global i18n.logOutputEncoding utf-8
git config --global i18n.commitEncoding utf-8
git config --global gui.encoding utf-8
```

```
git config --global svn.pathnameencoding utf-8
git config --global core.savecrlf true           #拒绝提交包含混合换行符的文件
git config --global core.autocrlf false         # 检出时不自动做换行符转换
git config --global core.ignorecase false       #区分大小写（包括文件名）

export LESSCHARSET=utf-8                       # less分页时正常显示中文
set output-meta on                             #日志信息可以输入中文
set covert-meta off
alias ls='ls --show-control-chars --color=auto' #ls 时 显示正常中文文件名
?
```

[filter "lfs"]:用Git内部的文本指针替换Large File Storage, 指向存储在远程服务器如GitHub.com上的文件

```
clean = git-lfs clean -- %f
smudge = git-lfs smudge -- %f
process = git-lfs filter-process
required = true

[core]:核心
editor = "E:\\Program\\VS Code\\Code.exe" --wait
autocrlf = true:自动回车换行CRLF
excludesfile = C:\\Users\\12393\\Documents\\gitignore_global.txt

[gui]
recentrepo = C:/Users/12393/Desktop/deno1
encoding = utf-8

[user]
name = Zakijxz
email = Zakijxz@github.com

[difftool "sourcetree"]
cmd = '"$LOCAL"' "$$REMOTE"

[mergetool "sourcetree"]
cmd = ""
trustExitCode = true

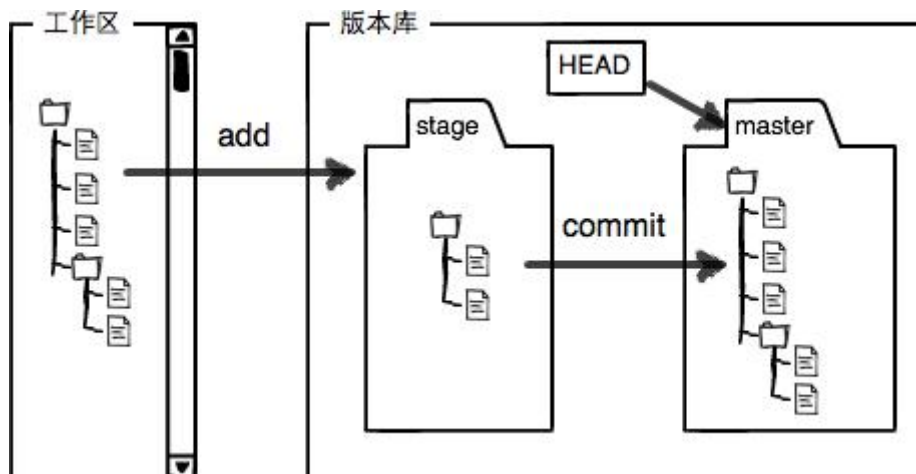
[alias]
co = checkout
br = branch
ci = commit
st = status
unstage = reset HEAD --
last = log -1 HEAD
lg = log --color --graph --pretty=format:'%Cred%h%Creset -
%C(yellow)%d%Creset msg:%s %Cgreen(%cr) %C(bold blue)<an>%Creset' --abbrev-
commit --relative-date

[i18n]
logoutputencoding = utf-8
```

## 初始化版本库respository

- 先用cd切换到对应的工作目录下, 也可以用以下命令创建一个工作目录, 再在该目录下创建版本库

```
$ mkdir dir_name
# cd dir_name
$ pwd
$ git init           #创建版本库respository
```



- stage暂存区, 初始化时Git为默认创建的第一个分支 `master`, 以及指向 `master` 的指针 `HEAD`

## 检查状态,工作区是否修改,stage区是否待commit

`git status`:检查状态

## 添加文件到repo

### Step1 先add到stage暂存区,文件一定要在工作区下

`cat filename.后缀名` 在命令行下显示具体内容 (linux)

`type filename.后缀名` 在命令行下显示具体内容 (Windows)

`git add .`//记住这后面有一个点, 这是默认添加所有的意思, 也可以指定要添加的文件名

### Step2 commit提交到当前分支(默认为master),并备注信息(message)

`git commit -m "msg"`:把暂存区的所有内容提交到当前分支

## 查看commit历史info

`git log`:查看提交历史信息

`git log --pretty=oneline`:查看提交记录的版本号 `commit id`和 `commit_messages`

`git log` 的常用选项

选项	说明
<code>-p</code>	按补丁格式显示每个提交引入的差异。
<code>--stat</code>	显示每次提交的文件修改统计信息。
<code>--shortstat</code>	只显示 <code>--stat</code> 中最后的行数修改添加移除统计。
<code>--name-only</code>	仅在提交信息后显示已修改的文件清单。
<code>--name-status</code>	显示新增、修改、删除的文件清单。
<code>--abbrev-commit</code>	仅显示 SHA-1 校验和所有 40 个字符中的前几个字符。
<code>--relative-date</code>	使用较短的相对时间而不是完整格式显示日期（比如，“2 weeks ago”）。
<code>--graph</code>	在日志旁以 ASCII 图形显示分支与合并历史。
<code>--pretty</code>	使用其他格式显示历史提交信息。可用的选项包括 <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> 和 <code>format</code> （用来定义自己的格式）。

修改后查看git status,提示如下

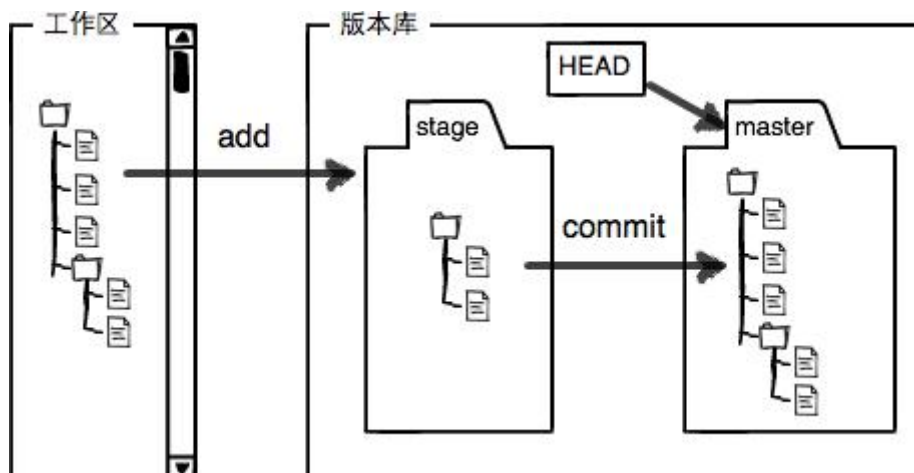
```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hello.py

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

`git commit -am "msg"`:先将所有工作区变动文件,提交到暂存区,再运行commit,不需要add操作

`git diff HEAD -- <filename,***>`:查看工作区和版本库里面最新版本的区别,有+和-

## 版本退回?撤销修改?



`git checkout -- file_name:`

- 当修改之前的文件是还未添加到stage里, 仅工作区的修改,撤回到上一次commit的版本库一致
- 当文件已经添加到stage里的, 又做了修改, 则恢复到上一次add到stage里的状态
- 恢复到修改前的工作区、Stage状态

`git checkout .:`后面有点,撤销所有文件的修改

`git reset HEAD file_name`:撤销add到stage里的修改, 将之撤回到工作区, HEAD代表是最新版本, 也可以换成其他版本的代号, 等同于unstage

## checkout commit\_id:临时检查以前的提交状态 (detached HEAD)

`git log --pretty=oneline`:获取提交记录的版本号 commit id

```
$ git log --pretty=oneline
5f976545ed7f77b874fbee05659d91efea8ecd35 (HEAD -> master) second
d1aa650dee14b2d00d48f0e8ae34ac18a5a32b started
```

`git checkout 前7位id`\\不加引号

如:`git checkout d1aa650`回到started提交状态, 但这仅进入到 **detached HEAD** 指针状态, 并未改变master分支, 多人开发时不要在此状态下进行修改, 使用 `git checkout master` 回到主分支的最新提交

## reset --hard:破坏master分支,恢复到指定的提交状态



`git log --pretty=oneline`:获取提交记录的版本号 commit id

`git reset --hard HEAD^`\\回到上一个版本

`git reset --hard HEAD^^`\\回到上上个版本, 有几个^就是上几个版本

`git reset --hard HEAD~num`\\回到前num个版本

就是将HEAD指针上移动或者下移动来切换版本, 上移需要用 `git reflog`:查询你的每一个更改命令id值

```
$ 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
```

- 可以根据id号切换

`git log --pretty=oneline`:获取提交记录的版本号 commit id

`git reset --hard 前7位id`:恢复master分支到指定的提交版本号

如`git reset --hard d1aa65`将master恢复到started提交状态, 此时master分支就只剩指定及之前的状态了

```
$ git log --pretty=oneline
d1aa650dee14b2d00d48f0e8ae34ac18a5a32b (HEAD -> master) started
```

## 补交未add文件至上一次commit

```
$ git commit -m 'initial commit'
$ git add forgotten_file           #提交后才发现有遗忘add的文件
$ git commit --amend
#最终你只会会有一个提交——第二次提交将代替第一次提交的结果
```

## revert代码回滚和reset的区别

- `git revert`:用一次新commit逆转之前的commit, 相当于时间线依旧前行, 版本回滚, 所有历史均保留
- `git reset`:回到某次提交, 那次提交及之前的commit都会被保留, 但是此commit id之后的修改都会被删除
- `git reset --hard` 撤销到某次提交

## 删除

### doc命令删除工作区的文件?

`rm <file_name>`:dos删除命令,此时仓库中会检测到工作区这个文件已经被删除了

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    tste.txt

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

### 后续:删除仓库里的文件或恢复误删

1. 如果确实要从仓库中删除那个文件, 则需用 `git rm` 命令, 并提交新版本的信息

`git rm file_name`, 与 `git add file_name` 相反

`git commit -m "remove file_name"`

2. 如果是误删, 则可以根据仓库去恢复工作区文件

`git checkout -- file_name`

`git checkout` 其实是用版本库里的备份替换工作区的版本, 无论工作区是修改还是删除, 都可以“一键还原”

## 删除仓库:删除所有提交记录,不删除工作区内容

```
rm -rf .git
```

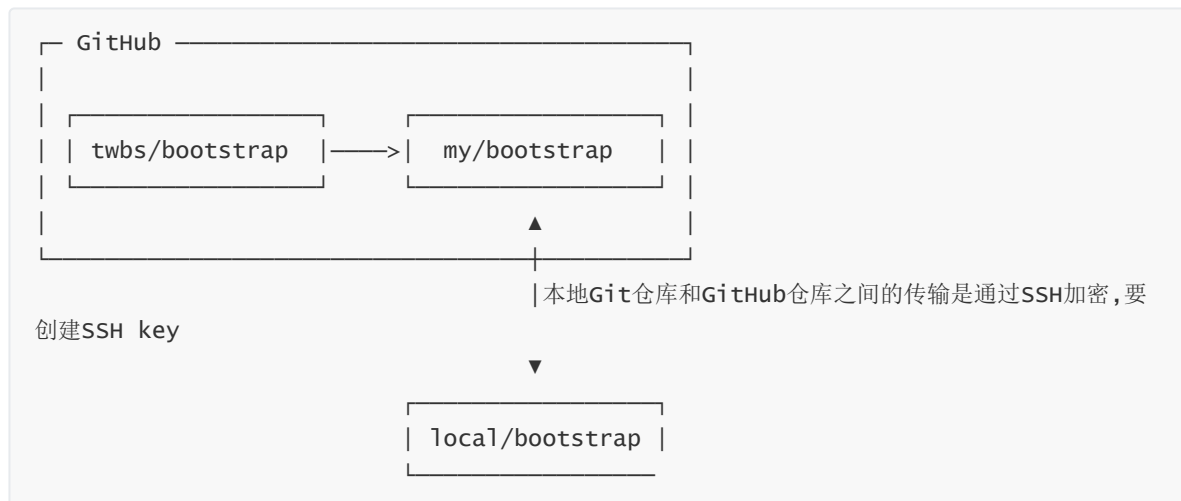
此时你已经做完了版本的开发, 则可以删除所有版本提交的历史记录, 构成一个新的开发起点。

## Git移动或者重命名文件

```
git mv [-v] [-f] [-n] [-k] <source> <destination>
git mv [-v] [-f] [-n] [-k] <source> ... <destination directory>

$ git mv cf.txt abc.txt      #重命名
$ git mv text.txt mydir     #移动
```

# Git Remot远程仓库:origin



twbs/bootstrap:其他人的GitHub服务器上的仓库bootstrap

my/bootstrap:我在GitHub服务器上的仓库bootstrap

local/bootstrap:电脑上的本地库bootstrap

Pull request:发起修改请求

clone: 从云上下载到本地库

## Step1 创建SSH Key密钥对

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

填写邮件地址, 然后一路回车, 使用默认值即可, 由于这个Key不是用于军事目的, 所以也无需设置密码。

去用户主目录 (~或C:\Users\\*\*\*) 里找到 .ssh 目录, 里面有 id\_rsa 和 id\_rsa.pub 两个文件, 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。

```
12393@zxj MINGW64 ~/Desktop/deno1 (master)
$ pwd
/c/Users/12393/Desktop/deno1

12393@zxj MINGW64 ~/Desktop/deno1 (master)
$ cd ~

12393@zxj MINGW64 ~
$ ssh-keygen -t rsa -C "youremail@example.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/12393/.ssh/id_rsa):
Created directory '/c/Users/12393/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/12393/.ssh/id_rsa.
Your public key has been saved in /c/Users/12393/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:AbAu7uLDxghwFH03BpICZxE1PePqBv/iNc7FRdWxoOc youremail@example.com
The key's randomart image is:
+---[RSA 3072]----+
|  o=+o      oo . |
| . o ..=    o o. |
| = . o +   o . . |
```

## Step2 remote 登记SSH Key

- 登陆GitHub,打开 Account settings 下 SSH and GPG Keys

### SSH keys

New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.



**First\_SSH\_RSA\_Pub**  
18:ac:28:e9:b4:ea:f6:ff:cc:50:67:64:d0:52:9c:09  
Added on Feb 7, 2020  
Never used — Read/write

Delete

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#).

- 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只有知道了你的公钥,才能给你推送的权限。

- GitHub允许添加多个Key

每个电脑需要一个key,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,就可实现多台电脑推送。

`ssh -T git@github.com`:利用 ssh 协议测试检验是否配置成功

## Step3 创建Bare Repo

- 在Github服务器上创建remote库,默认名为origin
- 注:为避免push冲突,不勾选initialize选项

Owner

ZakijxZ ▾

Repository name \*

test ✓

Great repository names are short and memorable. Need inspiration? How about [miniature](#)

Description (optional)

test

☒ Public

Anyone can see this repository. You choose who can commit.

☐ Private

You choose who can see and commit to this repository.


Skip this step if you're importing an existing repository.

☐ Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾

Add a license: None ▾

 可以查询license该怎么选

Create repository

- [关于license的选择](#)



- [具体的各种license说明](#)
- [各种license的说明](#)
- 为无license的repo创建license的方法

标注 1:Commit directly to the master branch.直接将此许可证提交到master分支

标注 2:Create a new branch for this commit and start a pull request.新建一个分支，然后提交到master，再进行合并



**我只想专心写代码**

**MIT License** 是一个宽松的协议，它允许别人用你的代码做任何事情，但必须保证你的所有权，并且你无须承担代码使用产生的风险。

**jQuery**, **.NET Core**, 和 **Rails** 使用 MIT License。



**我想保护我的专利**

**Apache License 2.0** 与 MIT License 比较类似，但提供了专利贡献者的明确授权，使用者需要放置版权说明。

**Android**, **Apache**, 和 **Swift** 使用 Apache License 2.0。



**我关心项目分享与改进**

**GNU GPLv3** 是一个 Copyleft 协议，所有使用了你源代码的人都必须按照相同的协议提供源代码，使用者需要放置版权说明。

**Bash**, **GIMP**, 和 **Privacy Badger** 使用 GNU GPLv3。

{ 上面的那些都不满足我的要求？ }

**非软件类内容**

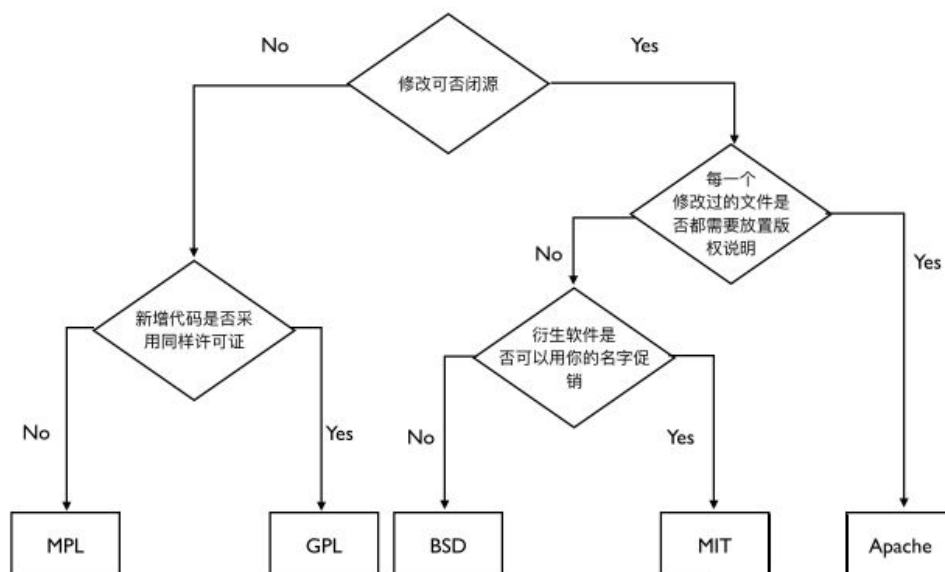
非软件类协议。

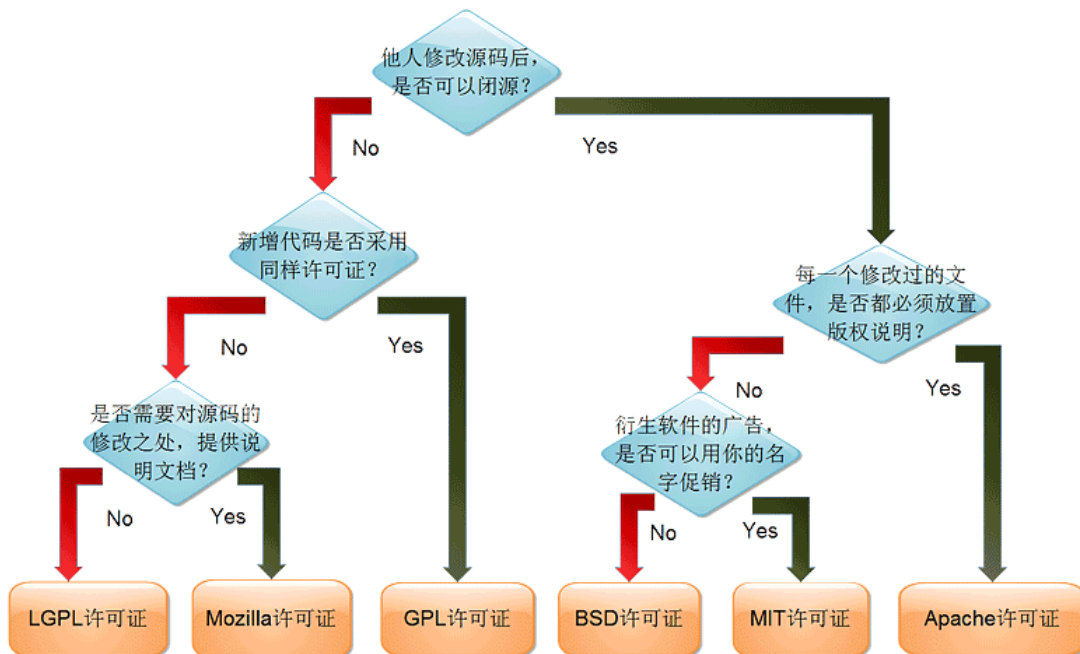
**我想要更多选择**

更多开源软件协议。

**我并不想要协议**

你确定要这么做？





## Step4 关联local repository | 从云上克隆repository

### 关联local repository

- 命令1: `git remote add <shortname> <url>` :添加一个新的远程 Git 仓库

如: `git remote add origin git@server-name:path/repo-name.git`

再如: `git remote add origin git@github.com:ZakijxZ/test.git`

`origin` 是Git中远程库的默认名字

- 命令2: `git push -u origin master` :第一次推送master分支的所有内容，会有SSH的一个 Question

```

$ git push -u origin maste
The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.
RSA key fingerprint is xx.xx.xx.xx.xx.
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。之后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

Warning: Permanently added 'github.com' (RSA) to the list of known hosts.

```

***@*** MINGW64 ~/Desktop/deno1 (master)
$ git push origin master
To github.com:ZakijxZ/test.git
 ! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:ZakijxZ/test.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
解决办法:
git pull origin master --allow-unrelated-histories
  
```

```
git pull --allow-unrelated-histories
```

 #或者初创repo时候不要初始化readme文件

[git 无法push远程仓库 Note about fast-forwards 问题解决](#)

[git 无法push远程仓库 Note about fast-forwards](#)

- 命令3: `git push origin <branch_name>` :推送某一支

## 从云上clone repo

```
git clone url
```

`git clone git@server-name:path/repo-name.git` :克隆云上已经存在的repo到local

```
$ git clone git@github.com:Zakijxz/test0.git
```

Cloning into 'test0'...

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

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

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

Git本身还支持SSH、HTTP、Git协议(最快)等,默认的 `git://` 使用ssh, 但也可以使用 `https` 等其他协议。使用 `https` 除了速度慢以外, 最麻烦是每次推送都必须输入口令, 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`。

HTTPS **SSH** SVN SVN+SSH

`git@gitee.com:whitesnow1970/my_m`

Copy

Clone with HTTPS ?

Use SSH

Use Git or checkout with SVN using the web URL.

`https://github.com/aymericdamien/TopDeep`



Open in Desktop

Download ZIP

## remote repo的操作

`git remote` :查看所有远程服务器列表的 `<short name>`

`git remote -v` :查看所有远程服务器列表详细信息( `<short name>`, `<url>`, `<权限>` )

```
$ git remote -v //查看所有远程服务器列表详细信息
```

```
origin git@github.com:michaelliao/learngit.git (fetch抓取权限)
```

```
origin git@github.com:michaelliao/learngit.git (push推送权限)
```

`git remote show <remote-name>` :查看某一个远程仓库的更多信息

```
$ git remote show origin//查看某一个远程仓库的更多信息
* remote origin
Fetch URL: https://github.com/schacon/ticgit
Push URL: https://github.com/schacon/ticgit
HEAD branch: master
Remote branches:
  master                                tracked
  dev-branch                            tracked
Local branch configured for 'git pull':
  master merges with remote master
Local ref configured for 'git push':
  master pushes to master (up to date)
```

`git remote rename <old_name> <new_name>` :重命名远程repo

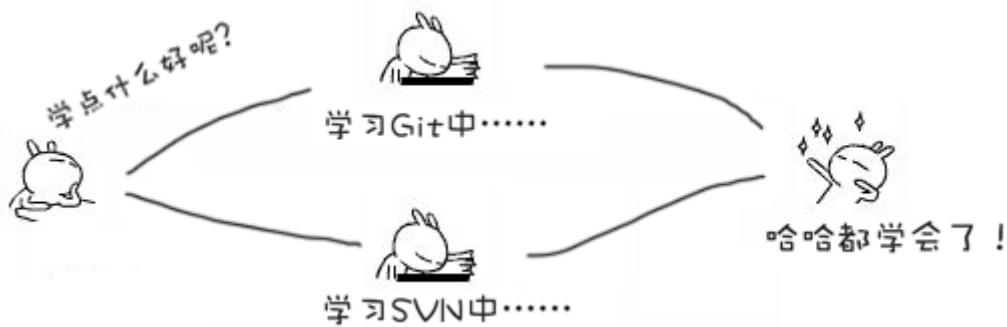
```
$ git remote rename pb paul
$ git remote
origin
paul
```

- 注:这同样也会修改你的远程分支名字。那些过去引用 `pb/master` 的现在会自动引用 `paul/master`。

`git remote rm <repo_short_name>` :删除远程repo

```
$ git remote rm paul
$ git remote
origin
```

## Step5 分支管理



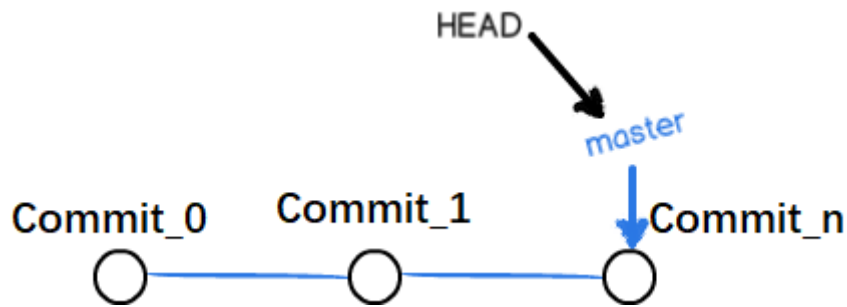
其他版本控制系统:如SVN(慢):SVN是subversion的缩写, 是一个开放源代码的版本控制系统, 通过采用分支管理系统的高效管理, 简言就是用于多个人共同开发同一个项目, 实现共享资源, 实现最终集中式的管理。

Git的分支管理(创建、切换和删除分支)很快(1秒钟之内就能完成)无论1个文件还是1万个文件的repo。

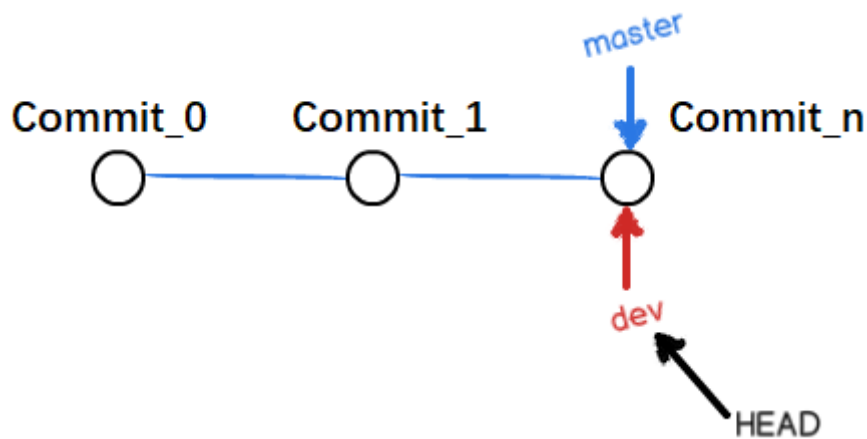
分支:在版本回退里, Git都把每一次Commit串成的一条时间线, 默认的第一条分支叫 `master` 分支。

## 工作原理

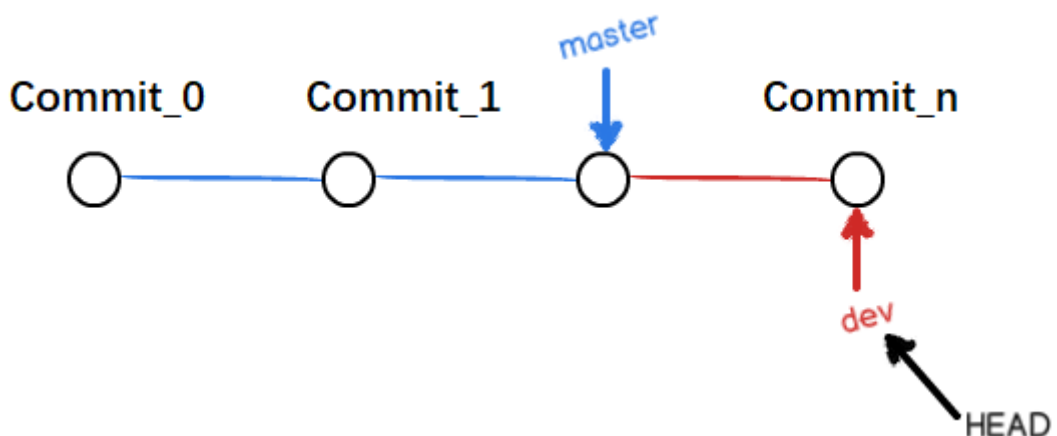
- 严格来说的指向关系: HEAD 指向 master, master 指向 commit, 即 HEAD 指向当前分支, 实现分支切换
- 最初, master 分支是一条线, Git 用 master 指向最新 commit, 再用 HEAD 指向 master, 就能确定当前分支, 以及当前分支的提交点



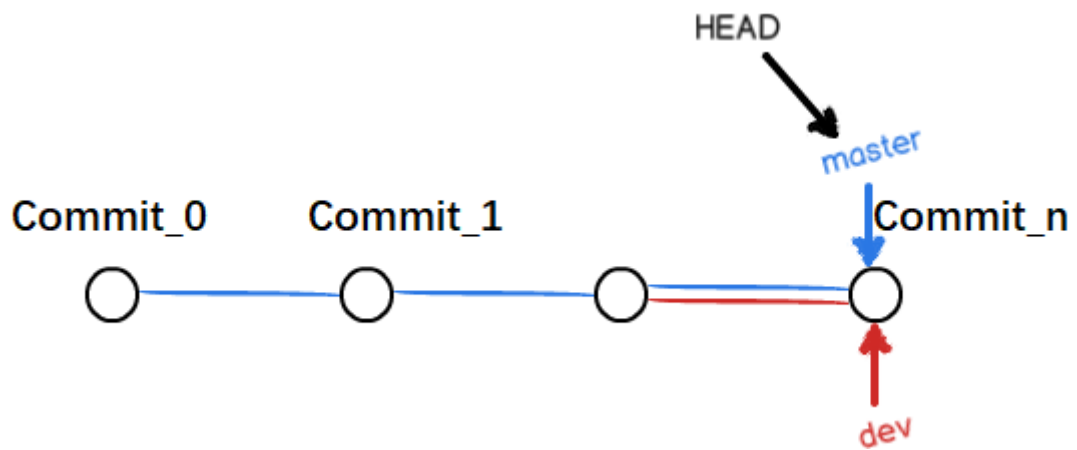
- 随着你的提交增多, master 分支就不断加长
- 当创建新的分支(如名为 dev) 时, Git 先新建了一个指针叫 dev, 指向 master 相同的提交, 再把 HEAD 指向 dev, 就表示当前分支在 dev 上, 工作区内容保持不变



Git 创建一个分支很快: 除增加一个 dev 指针并改变 HEAD 的指向, 工作区的文件不做任何改变; 但以后的对工作区的修改和提交就在另一条支路上进行, 原 master 分支的版本信息保持停留不变。

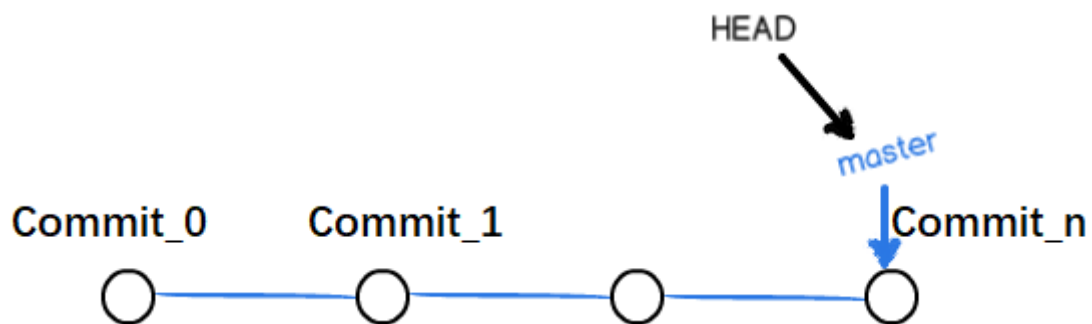


- 合并: 当沿着 dev 分支完成了开发工作, 通过移动 master 指向 dev 的当前 commit, 完成主线分支的合并



分支的合并也仅改动指针，工作区内容保持不变

- 删除多余分支:合并完分支后，为进一步推动开发，可以删除 dev 分支(删除 dev 指针即可),留下一条 master 分支完成后续开发的统一。



```
git checkout -b dev    #b参数表示创建并切换，等于以下两条命令
git branch dev        #创建分支dev
git checkout dev      #切换HEAD到分支dev
```

```
git branch            #查看当前分支，*代表当前分支
* dev
master
```

```
git checkout master   #切换回master分支
git merge dev         #把dev分支合并到master分支
```

提示信息(Fast-forward代表“快进模式”:直接把master指向dev的当前提交):

Updating \*\*\*\*\*

Fast-forward

\*\*\*\* | 1 +

1 file changed, 1 insertion(+)

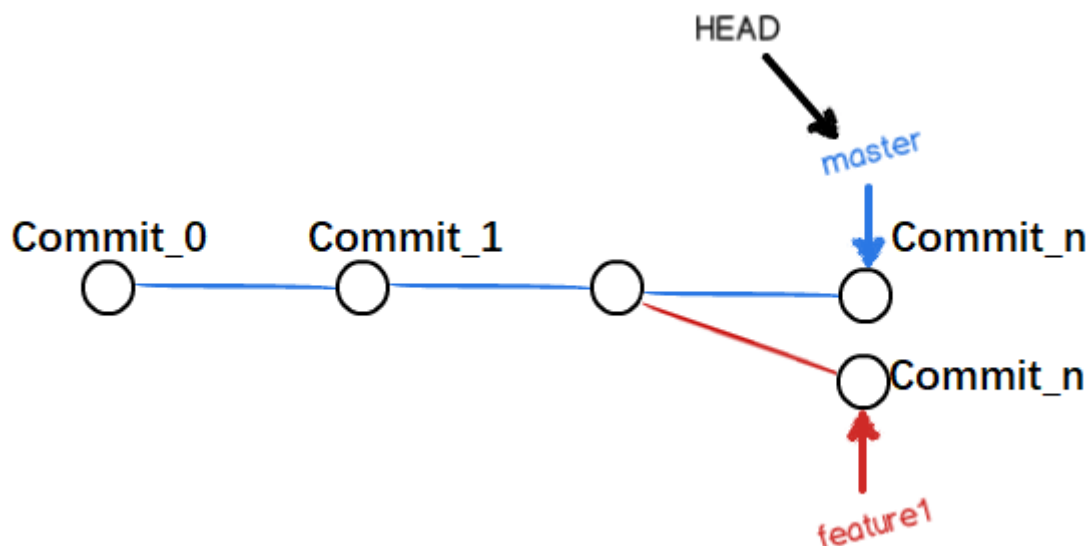
```
git branch -d dev    #删除dev分支
```

利用 branch 的工作原理，在进行每个环节开发时都在分支上进行，虽效果与在 master 上一样，但开发过程的安全性得到了提高。

含义	命令
查看分支	<code>git branch</code>
查看本地所有分支详细信息	<code>git branch -vv</code>
创建分支	<code>git branch &lt;name&gt;</code>
切换分支	<code>git checkout &lt;name&gt;</code> 或者 <code>git switch &lt;name&gt;</code>
创建+切换分支	<code>git checkout -b &lt;name&gt;</code> 或者 <code>git switch -c &lt;name&gt;</code>
合并某分支到当前分支	<code>git merge &lt;name&gt;</code>
删除分支	<code>git branch -d &lt;name&gt;</code>

## Conflict分支冲突

- 当分支 feature1 提交了新的 commit 后, 切换回 master 分支的分叉点也提交了 commit, 此时 master 分支和 feature1 分支各自都分别有新的 commit, 此时无法简单的“快速合并”:



当两条支路的 commit 是针对不同文件, 则直接合并即可。当两条支路的文件针对的是同一文件, 则会产生冲突, 必须手动解决冲突后再提交, 即手动修改合并到一半的文件后再 `add, commit`

```
$ git merge feature1
Auto-merging add.txt
CONFLICT (content): Merge conflict in add.txt
Automatic merge failed; fix conflicts and then commit the result.
//git status也可以查看到冲突
$ 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:   add.txt
```

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

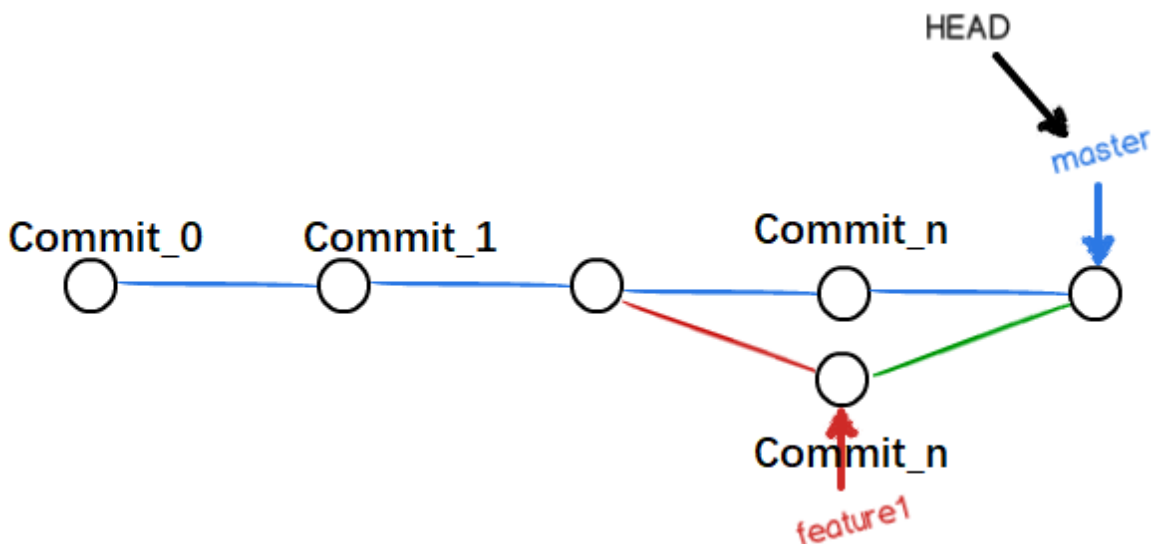
下面是文件add.txt发生合并冲突时的内容，Git用<<<<<<，=====  
>>>>>>标记出不同分支的内容。

```
base content
<<<<<< HEAD
kiujjjk
=====
nihao
>>>>>> feature1
```

手动修改内容并保存后重新提交

```
$ git commit -am"conflict fixed"
[master *****] conflict fixed
```

现在，`master`分支和`feature1`分支情况如下：



```
git log --graph --pretty=oneline --abbrev-commit:查看分支的合并情况
```

`git branch -d <name>`:删除多余分支

## git log --graph:命令分支合并图

## Fast forward快速合并模式?

```
git merge --no-ff -m "merge with no-ff(具体的msg)" <branch_name>
```

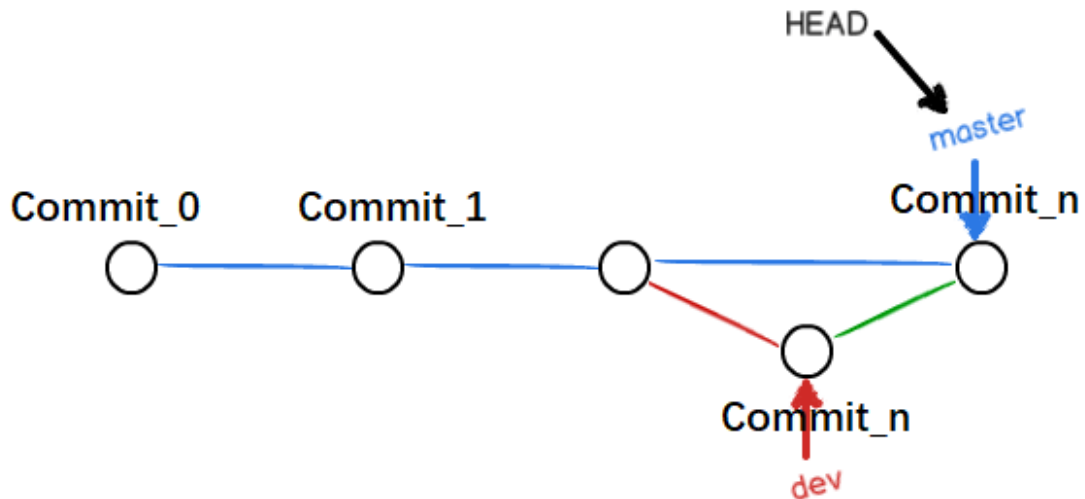
- 一般情况下，使用 Fast forward 模式的 dev 分支和删除，dev 分支信息丢失，被 master 分支直接占用，dev 指针无指向

```
$ git log --graph --pretty=oneline --abbrev-commit
* f22efed (HEAD -> master) dev
* b834144 conflict fixed
|\
| * 1ef66cf (feature2) -mfenzhi2
* | 03412fd "master2
*****
```



- 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。

```
$ git merge --no-ff -m "merge with no-ff" dev
$ git log --graph --pretty=oneline --abbrev-commit
* a01f214 (HEAD -> master) merge with no-ff
|\
| * be46c2a (dev) dev
|/
* f22efed dev
提交的id号 ()里面的是分支名称 后面的是提交时备注的msg
*****
```



## Feature分支

- 每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支

`git branch -D <br_name>`:强行删除没有合并的分支，类似于开发完之后发现 master 分支不需要该分支，故需要就地完全删除。合并后的删除参数用的小 d。

## Bug分支

- 修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；
- 当手头工作没有完成时又不能提交时，先把此时工作现场储藏，此时工作区就是干净的，然后去修复临时来的bug任务，修复后，再重新回到上一工作现场；stash-藏

```
$ git stash//Step1 临时储藏工作现场
```

```
*****//Step2 Bug修复完成
```

```
$ git stash list//Step3 查看工作现场位置
```

```
stash@{0}: WIP(dev) on dev: f52c633 add merge
```

```
//Step4 恢复方法有两种
```

```
$ git stash apply//恢复，但是恢复后，stash内容并不删除；
```

```
$ git stash drop//删除stash内容
```

```
$ git stash pop//恢复的同时删除stash内容
```

```
$ git stash list//Step5对于只Stash了一次的repo，则再查不到stash内容
```

```
$ git stash apply stash@{0} //针对连续stash了多个工作现场的repo,则需要根据stash的编号去恢复
```

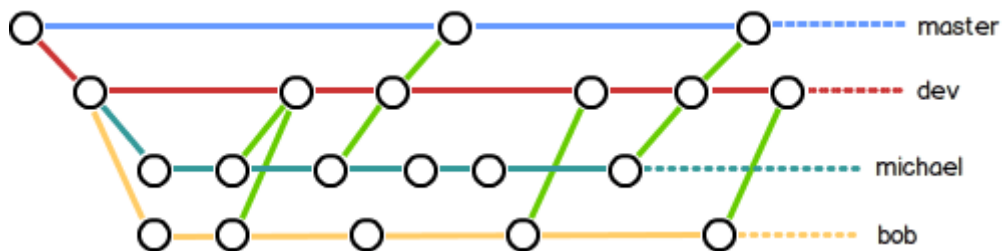
- 一般bug分支用issue编号, 如 `git checkout -b issue-101` 或 `git switch -c issue-101`
- `dev` 源自于 `master` 的分支, `master` 的bug在 `dev` 上也有。故在 `master` 分支上修复的bug, 想要合并到当前 `dev` 分支, 可以用 `git cherry-pick <提交的7位>` 命令:复制一个特定的提交到当前分支。同样的在 `dev` 上修复的bug可以类似的复制到 `master` 上

```
$ git commit -m "fix bug 101"
[issue-101 4c805e2] fix bug 101
1 file changed, 1 insertion(+), 1 deletion(-)
$ git cherry-pick 4c805e2
[master 1d4b803] fix bug 101 //两个commit的提交内容和备注信息一样, 但id号不同, 意味着不是同一个对象
1 file changed, 1 insertion(+), 1 deletion(-)
```

## Strategy分支策略

实际开发中分支管理的几个基本原则:

- `master` 分支应该是非常稳定的, 也就是仅用来发布新版本, 平时不能在上面干活;
- 干活都在 `dev` 分支上, 因为 `dev` 分支是不稳定的, 到某个时候, 比如版本发布时, 再把 `dev` 分支合并到 `master` 上, 再 `master` 分支发布1.0版本;
- 合作开发者每个人都有自己的分支, 都在 `dev` 分支上干活, , 平时往 `dev` 分支上合并, 发布时才合并到 `master` 分支, 就像这样:
- `dev`在软件开发中多用于开发软件的代号, 与Beta(测试版)的意思相近, 其意思为"开发中的版本".



## Push>>>origin?

- `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
- `feature` 分支是否推到远程, 取决于你是否和你的小伙伴合作在上面开发
- `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

## 多人协作

- `git remote -v`:查看远程库信息

## 推送push

- `git push <remote-name> <branch-name>`:推送某一本地分支
- `git push origin branch-name`:推送某一本地分支到远程分支且名字相同
- `git push --force`:强制推送, 覆盖

```
$ git push origin serverfix:awesomebranch
```

//将本地的 `serverfix`分支推送到远程仓库上的`awesomebranch`分支,远程分支与本地分支可以不同名

## credential cache凭证存储

如果你正在使用 HTTPS URL 来推送, Git 服务器会询问用户名与密码。默认情况下它会在终端中提示服务器是否允许你进行推送。如何避免每次输入密码?

设置一个“credential cache”。最简单的方式就是将其保存在内存中几分钟,可以简单地运行 `git config --global credential.helper cache` 来设置它。

更多[凭证存储](#)

## 冲突

- 如果出现远端已经有了 `dev` 分支,如你的同事率先创建了分支并与你针对相同部分进行了修改,则推送失败,如下;

```
$ git push origin dev0//你在提交dev0分支时与已经存在的远端origin/dev0分支产生修改冲突
To github.com:ZakijxZ/test.git
 ! [rejected]        dev0 -> dev0 (fetch first)
error: failed to push some refs to 'git@github.com:ZakijxZ/test.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.
```

- 此时需先用 `git pull` 抓取远程的同一条分支到本地进行合并,由于本地 `dev0` 与远程 `origin/dev0` 没有链接关系(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),也导致了失败。

```
$ git pull
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

    git branch --set-upstream-to=origin/<branch> dev0
```

- `git branch --set-upstream-to=origin/<branch> dev0`:建立远程分支与本地分支的链接

```
$ git branch --set-upstream-to=origin/dev0 dev0
Branch 'dev0' set up to track remote branch 'dev0' from 'origin'.
```

- 建立链接后再 `git pull` 合并冲突
- `git checkout -b branch-name origin/branch-name`:在本地创建和远程同名的分支,并保持跟踪。

- 从远程抓取分支, 使用 `git pull`, 如果有冲突, 要先处理冲突。

## 跟踪track

```
$ git checkout --track origin/serverfix //相同名分支的跟踪
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'

$ git checkout -b sf origin/serverfix //本地分支sf跟踪远程分支origin/serverfix,不同名分支的跟踪
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

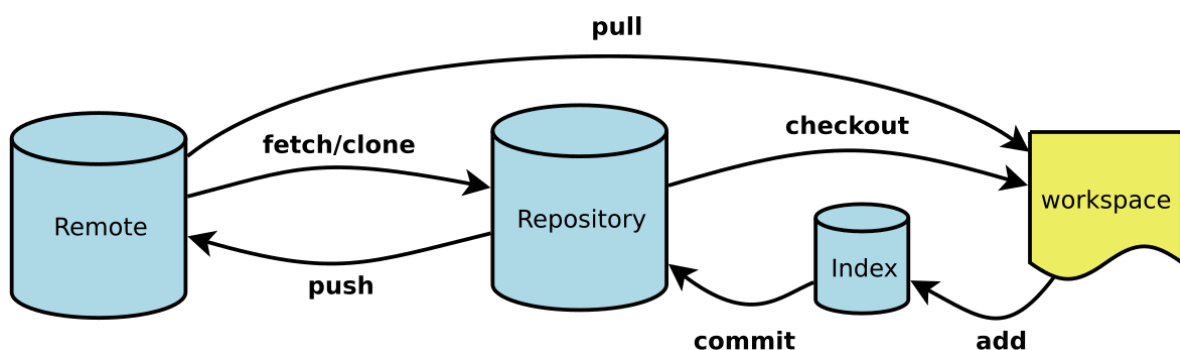
## 添加与上游的链接

```
$ git branch -u origin/serverfix //添加同名的链接, u其实是upstream的缩写
Branch serverfix set up to track remote branch serverfix from origin.

$ git branch --set-upstream-to=origin/dev0 dev0 //添加同名或者不同名的链接
Branch 'dev0' set up to track remote branch 'dev0' from 'origin'.
```

## 上游upstream的快捷方式

- 设置好跟踪分支后, 可通过 `@{upstream}` 或 `@{u}` 快捷方式来引用它。所以在 `master` 分支时并且它正在跟踪 `origin/master` 时, 可以使用 `git merge @{u}` 来取代 `git merge origin/master`。



## 拉取fetch和pull的区别

- `git fetch`: 相当于是从远程获取最新版本到本地, 不会自动merge, 需在用户检查了以后决定是否合并到工作本机分支中; 相当于第二次从远端克隆

```
git fetch <remote-name远程主机名> //将某个远程主机的更新全部取回本地
git fetch <远程主机名> <分支名> //从远端取回特定分支, 注意之间有空格
//取回的分支默认在本地为FETCH_HEAD:特定branch在服务器上的最新状态
git log -p FETCH_HEAD //在本地查看刚取回分支的更新信息
git merge FETCH_HEAD //将拉取下来的最新内容合并到当前所在的分支中

git pull <远程主机名> <远程分支名>:<本地分支名> //也可以将拉下来的最新分支命名
//取回远程主机某个分支的更新, 再与本地的指定分支合并。
git fetch origin master:tmp
//在本地新建一个tmp分支, 并将远程origin仓库的master分支代码下载到本地tmp分支
git diff tmp
//来比较本地代码与刚刚从远程下载下来的代码的区别
git merge tmp
```

```
git fetch origin master//
git log -p master..origin/master//比较本地的master分支和origin/master分支的差别
git merge origin/master
```

- `git pull`:相当于是从远程获取最新版本并merge到本地

```
git pull <远程主机名> <远程分支名>:<本地分支名>
//取回远程主机某个分支的更新，再与本地的指定分支合并。
//如果是与本地的当前分支合并，则冒号后面的东西可以忽略。
```

## 四个同步命令的比较

```
$ git fetch:下载远端追踪各个的所有历史
$ git merge:将链接跟踪对应分支并合并到当前本地分支
$ git push:将所有本地分支提交上传到remote
$ git pull:拉取remote对应分支的所有新提交更新你当前的本地工作分支,等同于`git
fetch`+`git merge`
```

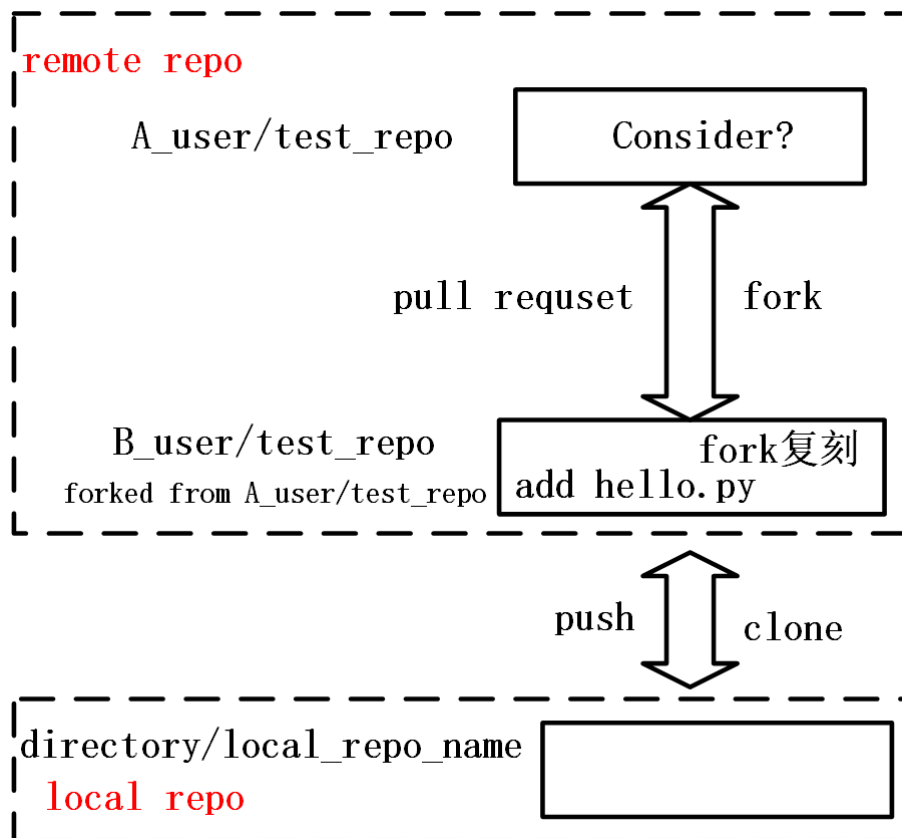
## 删除远程分支

```
$ git push origin --delete serverfix//从服务器上删除 serverfix 分支
To https://github.com/schacon/simplegit
- [deleted]          serverfix
```

- 注:这个命令做的只是从服务器上移除这个指针。Git 服务器通常会保留数据一段时间直到垃圾回收机制运行，所以如果不小心删除掉了，通常是很容易恢复的。

## 开源贡献流程

1. fork开源项目到自己的复刻远程库
2. clone自己的复刻远程库到本地库
3. 在本地库里面添加原始远程库的git服务器地址
4. 实时原始远程库的 更新并与自己本地库的开发合并
5. push到自己的远程复刻库
6. 在第三方平台上发起pull request请求
7. 等待开源作者的审查
8. 对方考虑是否采纳并回复
9. 完成开源合作



注：如果克隆第三方远程库,只能fetch不能push,因为远程库没有配置你的公钥,不能实现公钥与私钥的配对。

## 学习一下linux早期是如何贡献的

第一步

```
wget http://test.com/project.2015-07-04.zip
```

第二步

```
unzip project.2015-07-04.zip
```

第三步

```
cp -R project.2015-07-04 project-my-copy
```

第四步

```
cd project-my-copy
```

第五步

(做了某些修改)

第六步

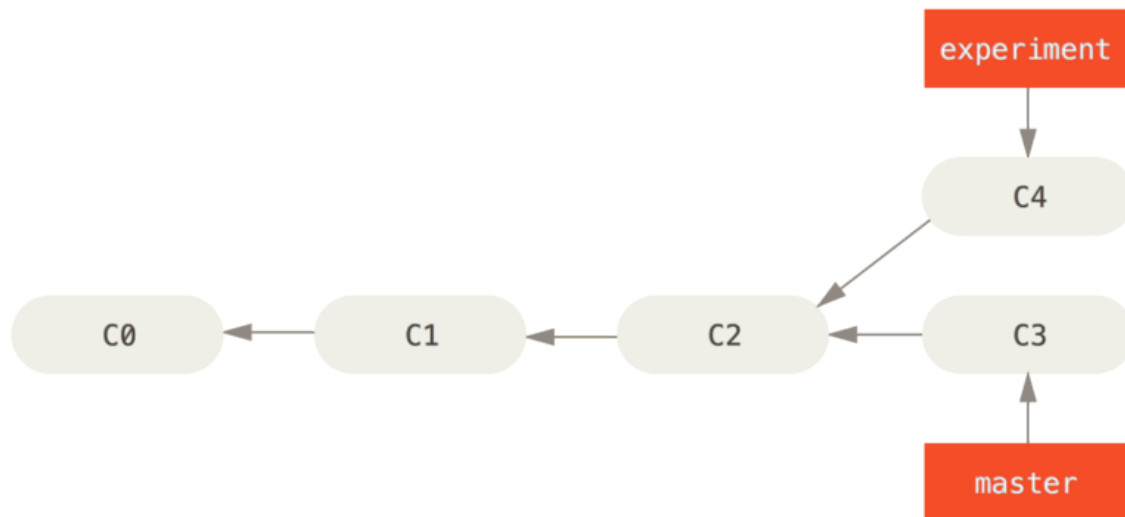
```
diff project-my-copy project.2015-07-04 > change.patch
```

第七步

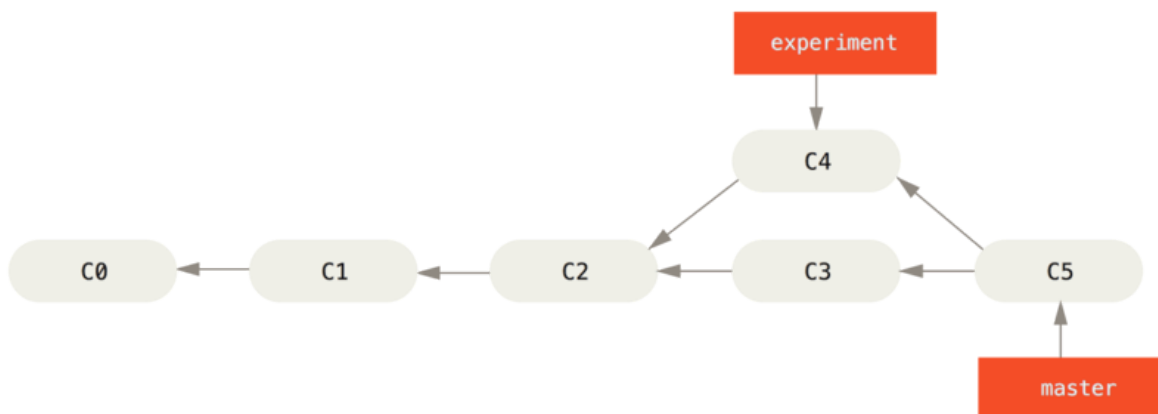
(通过E-mail发送修改补丁)

## Rebase变基与Merge

### Merge



`merge` 命令: 它会把两个分支的最新快照 (C3 和 C4) 以及二者最近共同祖先 (C2) 进行三方合并, 合并的结果是生成一个新的快照 (并提交)。

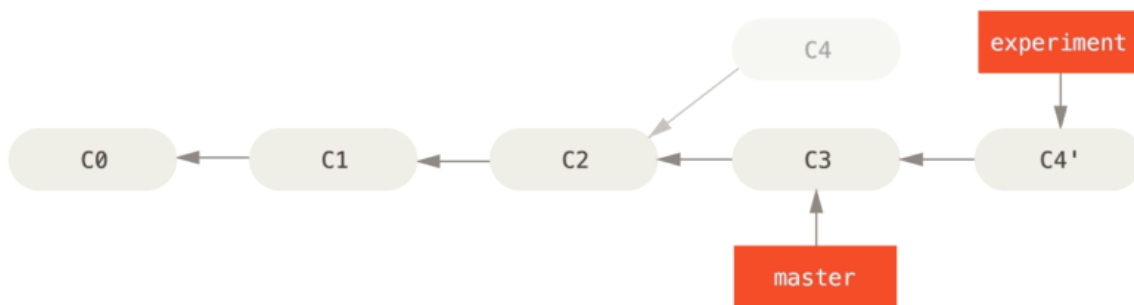


## 简单的rebase

- 将提交的历史记录整理成一条直线
- 你还可以提取在 C4 中引入的补丁和修改, 然后在 C3 的基础上应用一次, 称为变基, 也可以看做分支在主分支的重放。

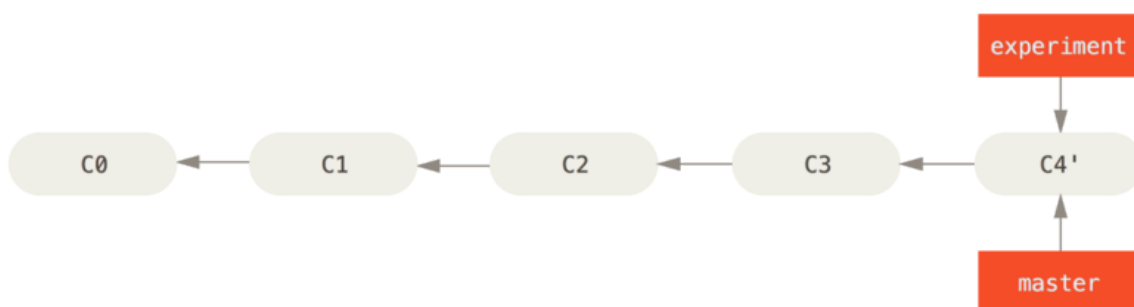
```
$ git checkout experiment//换在要移动的分支上
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

原理: 首先找到这两个分支 (即当前分支 `experiment`、目标基底分支 `master`) 的最近共同祖先 C2, 然后对比当前分支相对于该祖先的历次提交, 提取相应的修改并存储为临时文件, 然后将当前分支指向目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用。(译注: 写明了 commit id, 以便理解, 下同)



现在回到 `master` 分支，进行一次快进合并。

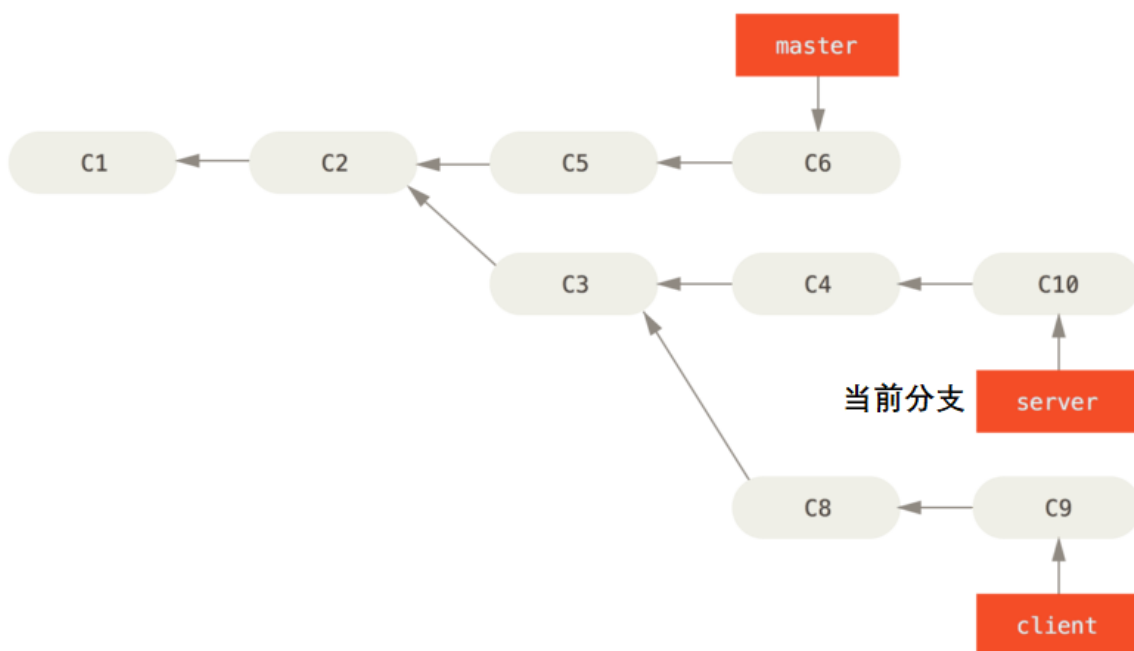
```
$ git checkout master
$ git merge experiment//将 C4 中的修改变基到 C3上
```



## 更多情况的rebase

- 从一个特性分支里再分出一个特性分支的提交历史。在对多个分支进行变基时，所生成的“重放”并不一定要在目标分支上应用，你也可以指定另外的一个分支进行应用。

比如你创建了一个特性分支 `server`，为服务端添加了一些功能，提交了 `C3` 和 `C4`。然后从 `C3` 上创建了特性分支 `client`，为客户端添加了一些功能，提交了 `C8` 和 `C9`。最后，你回到 `server` 分支，又提交了 `C10`。

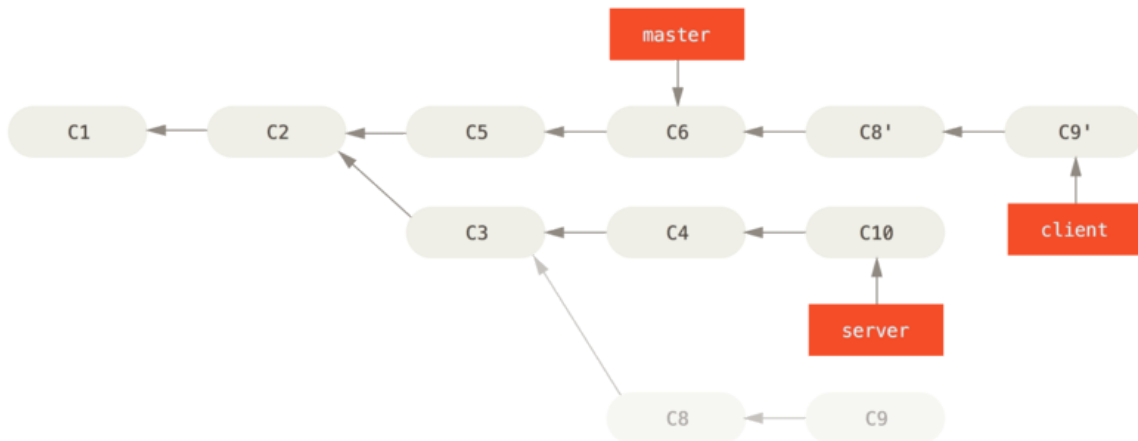




假设你希望将 `client` 中的修改合并到主分支并发布，但暂时并不想合并 `server` 中的修改，因为它们还需要经过更全面的测试。这时，你就可以使用 `git rebase` 命令的 `--onto` 选项，选中在 `client` 分支里但不在 `server` 分支里的修改（即 `C8` 和 `C9`），将它们在 `master` 分支上重放：

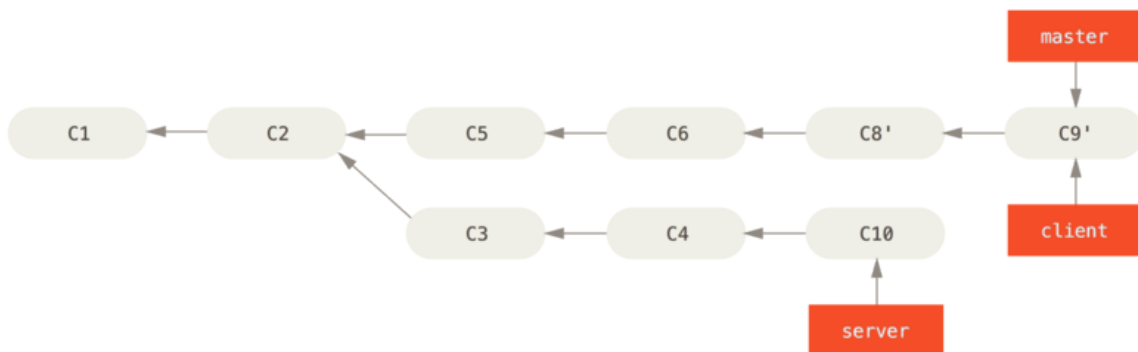
```
$ git rebase --onto master server client
```

以上意为：取出 `client` 分支，找出处于 `client` 分支和 `server` 分支的共同祖先之后的修改，然后把它们在 `master` 分支上重放一遍”。



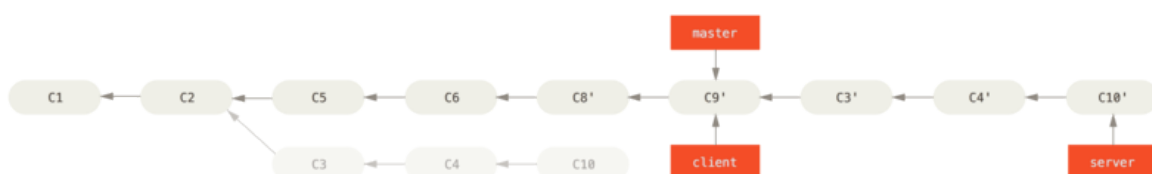
现在可以**快进合并** `master` 分支，使之包含来自 `client` 分支的修改

```
$ git checkout master
$ git merge client
```



测试稳定后，你决定将 `server` 分支中的修改也整合进来。使用 `git rebase [basebranch] [topicbranch]` 可以**直接**将特性分支（`server`）变基到目标分支（`master`）上。这样做能省去你先切换到 `server` 分支，再对其执行变基命令的多个步骤。`server` 中的代码被“续”到了 `master` 后面

```
$ git rebase master server
```



然后就可以**快进合并**主分支 master 了:

```
$ git checkout master // 换在目标分支上
$ git merge server
```

至此, `client` 和 `server` 分支中的修改都已经整合到主分支里了, 删除这两个分支, 最终提交历史会变成图一条直线时间线。

```
$ git branch -d client
$ git branch -d server
```



## 变基原则

- 不要对在你的仓库外有副本的分支执行变基

# 辅助Git功能

## Git Refs引用:references

### .git/refs中的引用

- .git/refs 目录下面找到这些包含 SHA-1 值的文件,其树结构为:

```
.git/refs/
  heads/
    master
    test
  remotes/
    origin/
      master
  tags/
    v0.9
```

**heads:** 描述当前仓库所有本地分支, 每一个文件名对应相应的分支, 文件内部存储了当前分支最新的 commit hash值

**remotes:** remotes文件夹将所有由 `git remote` 命令创建的所有远程分支存储为单独的子目录。在每个子目录中, 可以发现被fetch进仓库的对应的远程分支

**tags:** 描述当前仓库的tag信息, 其工作原理与heads一致。

## 特殊的引用 (Refs)

- 除了引用目录外, 还有一些特别的引用存在与.git路径顶部, 默认的名字

```
.git
HEAD
FETCH_HEAD
ORIG_HEAD
MERGE_HEAD
```

**HEAD:** 当前检出的commit/branch。HEAD 文件存储的是refs/heads/分支的引用。

**FETCH\_HEAD:** 最新从远程分支获取的分支。FETCH\_HEAD 文件存储的是远程分支的最新的commit信息。

**ORIG\_HEAD:** 作为备份指向危险操作前的HEAD

**MERGE\_HEAD:** 使用 `git merge` 命令合并进当前分支的提交

## 忽略某些文件夹/文件

- 如 `__pycache__` 等，在工作区创建一个 `.gitignore` 文件里面写入要忽略的文件夹/文件名：

```
touch .gitignore
vim .gitignore #创建忽略文件
```

名称	修改日期	类型	大小
learning_log	2019/12/7 14:24	文件夹	
learning_logs	2019/12/7 14:24	文件夹	
templates	2019/12/7 14:24	文件夹	
users	2019/12/7 14:24	文件夹	
.gitignore	2019/10/7 22:24	文本文档	1 KB
manage.py	2019/10/7 22:24	PY 文件	1 KB

.gitignore - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
ll_env/
__pycache__/
*.sqlite3
.DS_Store
```

## 忽略原则

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。
4. [更多忽略例子:针对数十种项目及语言的.gitignore 文件列表](#)

```
# 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/TOD
/TOD
```

```
# 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
#####
```

文件 `.gitignore` 的格式规范如下：

- 所有空行或者以 `#` 开头的行都会被 `Git` 忽略。
- 可以使用标准的 `glob` 模式匹配。
- 匹配模式可以以 `(/)` 开头防止递归。
- 匹配模式可以以 `(/)` 结尾指定目录。
- 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号 `(!)` 取反。

所谓的 `glob` 模式是指 `shell` 所使用的简化的正则表达式。

- 星号 `(*)` 匹配零个或多个任意字符；
- `[abc]` 匹配任何一个列在方括号中的字符（这个例子要么匹配 `a`，要么匹配一个 `b`，要么匹配一个 `c`）
- 问号 `(?)` 只匹配一个任意字符；
- 如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配（如 `[0-9]` 表示匹配所有 `0` 到 `9` 的数字）。
- 使用两个星号 `(*)` 表示匹配任意中间目录，比如 `a/**/z` 可以匹配 `a/z`，`a/b/z` 或 `a/b/c/z` 等。

```
#####
```

## 标签

### 列标签

- `git tag` 及特定模式

```
$ git tag
v0.1
v1.3

$ git tag -l 'v1.8.5*' //列出标签 1.8.5 系列
v1.8.5
v1.8.5-rc0//Release Candidate:发布候选版本
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

## 创建标签--轻量标签 (lightweight) & 附注标签 (annotated)

- 轻量标签 (lightweight) :很像一个不会改变的分支——它只是一个特定提交的引用。
- 附注标签是存储在 `Git 数据库` 中的一个完整对象。

它们是可以被校验的,包含打标签者的 `<名字>`、`<电子邮件地址>`、`<日期时间>`；还有一个标签信息；并且可以使用 `GPG (GNU Privacy Guard)` 签名与验证。通常建议创建附注标签，这样你可以拥有以上所有信息；

如果你只是想用一个临时的标签或因为某些原因不想要保存个私人开发那些信息，轻量标签也是可用的。

- 打标签需要切换到需要打标签的分支上,然后再 `git tag <tag_name>` 打标签

## 附注标签 (annotated)

```
$ git tag -a v1.4 -m "my version 1.4" //-a代表是附注，-m代表是附注标签中的具体信息
$ git tag
v0.1
v1.3
v1.4

$ git show v1.4//`git show <tag_name>`可以看到tag_name标签信息与对应的提交信息
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

## 附注标签 (lightweight) :将提交校验和存储到一个文件中——没有保存任何其他信息

```
$ git tag v1.4-lw//`git tag <tag_name>`
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.4-lw//`git show <tag_name>`查看不到任何其他的标签信息,只会显示出提交信息
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

## 给commit历史补标签

```
$ git log --pretty=oneline//前面是commit_id,后面是commit_messages
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
```

```
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

```
$ git tag -a v1.2 9fceb02//`git tag -a <tag_name> <commit_id>`给9fceb02补上标签
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

## 共享标签

`git push origin <tag_name>`:默认情况下不会传送到远程仓库服务器上, 必须显示的给出要推送的 `tag_name`

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]          v1.5 -> v1.5
```

如果想要一次性推送很多标签, 也可使用带有 `--tags` 选项的 `git push` 命令。这将会把所有不在远程仓库服务器上的标签全部传送到那里。

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]          v1.4 -> v1.4
 * [new tag]          v1.4-lw -> v1.4-lw
```

## 删除标签

`git tag -d <tag_name>`:删除某一具体标签

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
//注:上述命令并不会从远程仓库中移除这个标签,你必须使用 `git push :refs/tags/` 来更新你的远程仓库
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
- [deleted]          v1.4-lw
```

## 检出标签

如果你想查看某个标签所指向的文件版本,可以使用 `git checkout` 命令,虽然说这会使你的仓库处于“分离头指针 (detached HEAD)”状态——这个状态有些不好的副作用:

```
$ git checkout 2.0.0
Note: checking out '2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch>

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout 2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from
schacon/appendix-final
HEAD is now at df3f601... add atlas.json and cover image
```

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,除非确切的提交哈希。因此,如果你需要进行更改——比如说你正在修复旧版本的错误——这通常需要创建一个新分支:

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

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

## Git别名alias--类似于宏定义

`git config` 为每一个命令设置一个别名。

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

## 简单名字替换

- 例如，为了解决取消暂存文件的易用性问题，可以向 Git 中添加你自己的取消暂存别名：

```
$ git config --global alias.unstage 'reset HEAD --'
```

- 这会使下面的两个命令等价：

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

- 这样看起来更清楚一些。通常也会添加一个 `last` 命令，像这样：

```
$ git config --global alias.last 'log -1 HEAD'
```

- 这样，可以轻松地看到最后一次提交：

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebe1 <dreamer3@example.com>
Date: Tue Aug 26 19:48:51 2008 +0800

    test for current head

Signed-off-by: Scott Chacon <schacon@example.com>
```

- 一个 `lg` 的别名配置

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

//红色提交的简写哈希值,黄色的 `refs:`引用, `msg:`提交的信息, 绿色的提交如期, 蓝色的作者名字

\*表示一个commit, 注意不要管\*在哪一条主线上

|表示分支前进

/表示分叉

\表示合入

```
$ git lg
* 539861a - (HEAD -> master, origin/master) msg:Merge branch 'dev0' of github.
com:ZakijxZ/test into dev0 (2 days ago) <ZakijxZ>
|\
| * 49ea5de - msg:for test dev0 (2 days ago) <ZakijxZ>
* | 0a732df - msg:for test dev0 (2 days ago) <ZakijxZ>
|/
* 8b68957 - msg:dev f (2 days ago) <ZakijxZ>
* 75680e9 - msg:second (2 days ago) <ZakijxZ>
* 662b900 - msg:first (2 days ago) <ZakijxZ>
```



```
[user]
  name = ChivalrousDig
  email = ChivalrousDig@gmail.com
[color]
diff = auto
status = auto
branch = auto
interactive = true
  ui = true
[alias]
co = checkout
ci = commit
st = status
br = branch
hist = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)%Creset' --abbrev-commit --date=relative
type = cat-file -t
dump = cat-file -p
[core]
autocrlf = false

~
~
".gitconfig" 20 lines, 396 characters
```

Git log 输出格式含义:

参数	说明
%H	commit hash
%h	commit short hash
%T	tree hash树的完整哈希值
%t	tree short hash树的简写哈希值
%P	parent hash
%p	parent short hash
%an	作者名字
%aN	.mailmap 中对应的作者名字
%ae	作者邮箱
%aE	.mailmap 中对应的作者邮箱
%ad	-date=制定的日期格式
%aD	RFC2822 日期格式
%ar	日期格式，例如：1 day ago
%at	UNIX timestamp 日期格式
%ai	ISO 8601 日期格式
%cn	提交者名字
%cN	.mailmap 对应的提交的名字
%ce	提交者邮箱
%cE	.mailmap 对应的提交者的邮箱
%cd	-data=制定的提交日期的格式
%cD	RFC2822 提交日期的格式
%cr	提交日期的格式，例如：1day ago
%ct	UNIX timestamp 提交日期的格式
%ci	ISO 8601 提交日期的格式
%d	ref 名称references 或者 refs:引用
%e	encoding
%s	commit 信息标题
%f	过滤commit信息的标题使之可以作为文件名
%b	commit 信息内容
%N	commit notes

参数	说明
%gD	reflog selector, e.g., refs/stash@{1}
%gd	shortened reflog selector, e.g., stash@{1}
%gs	reflog subject
%Creset	重设颜色
%C(color)	制定颜色, as described in color.branch.* config option
%m	left right or boundary mark
%n	换行
%%	a raw %
%x00	print a byte from a hex code
%w([[.L.]])	switch line wrapping, like the -w option of git-shortlog(1).

## 执行外部命令，而不是一个 Git 子命令

- 在命令前面加入 `!` 符号。如果你自己要写一些与 Git 仓库协作的工具的话，那会很有用。我们现在演示将 `git visual` 定义为 `gitk` 的别名：

```
$ git config --global alias.visual '!gitk'
```

```
cd ~:进入用户目录 C:\Users\****
```

```
vim file_name.后缀名:编辑文件
```

## 在线 Git 代码托管平台

- [码云 Gitee 官网](#):五人协助，1000个仓库，单项目上限为1个G，单文件最大 100M，用户总仓库容量为5个G
- [GitHub 官网](#):荐1G以内.达到1G以后会受到GITHUB的通知邮件.上传超过50M的单个文件会 warning.无法上传超过100M的单个文件.目前大文件会提供一个1G的免费GIT-LFS(Large File Storage)空间
- [GitLab 官网](#)
- [Bitbucket 官网](#)
- [Atlassian社区](#)
- GitHub.com 找轮子(业界已经公认的软件或者库,不要Reinventing the wheel), [Coding.net](#) 托管私有项目。

## Git图形化界面管理项目

- TortoiseGit
- SmartGit
- SourceTree

解决预览区显示中文乱码

```
12393@zxj MINGW64 /i/0_CodeContent
$ git config --global i18n.logoutputencoding utf-8

12393@zxj MINGW64 /i/0_CodeContent
$ git config --global gui.encoding utf-8

12393@zxj MINGW64 /i/0_CodeContent
$ |
```

- Git GUI
- GithubDesktop
- [Public Git hosting sites](#)第三方托管的选择

## Git bash进入[vim](#)编辑器后怎么解决?

- 输入两大写字母 ZZ (记住是大写)
- 输入 :wq (保存后退出)或 :wq! (强行退出)
- [学习Vim](#)



## 参考教程

[Git Bash的相关教程](#)

[Git官方doc文档](#)

[Pro Git:self文件夹有中文版](#)

[git log 高级用法](#)

[git log日志格式设置](#)

[搭建远程服务器](#)

- 搭建Git服务器非常简单，通常10分钟即可完成
- 要方便管理公钥，用[Gitis](#)
- 要像SVN那样变态地控制权限，用[Gitolite](#)