

# Git笔记

Git

---

本篇文章记录我的git学习笔记，具体大纲如下：

- [Git和Github](#)
  - [本地版本控制](#)
  - [集中式版本控制](#)
  - [分布式版本控制](#)
- [安装Git](#)
- [创建GitHub账号](#)
- [创建第一个仓库](#)
  - [创建仓库](#)
  - [ssh key生成](#)
  - [git简单工作流](#)
- [使用git进行管理仓库](#)
  - [上节回顾](#)
  - [文件跟踪](#)
  - [git常用命令](#)
    - [git add和 git commit](#)
    - [git status](#)
    - [git diff](#)
    - [git log](#)
    - [git rm和git mv](#)
    - [远程命令](#)
      - [git clone](#)
      - [git remote](#)
      - [git push](#)
      - [git fetch](#)
      - [git pull](#)
    - [git reset和git revert](#)

- git别名
- git tag
- .gitignore文件
- LICENSE文件
- 分支的使用
  - git保存数据方式与分支简介
  - 创建分支
  - 切换分支
  - 为什么要使用分支？
  - 合并分支
    - git merge
    - git rebase
      - 多重变基
      - 变基的风险
  - 分支冲突
  - 远程分支
  - 分支练习（强烈推荐）
- pull request的使用
- 参考

## 1. Git和Github

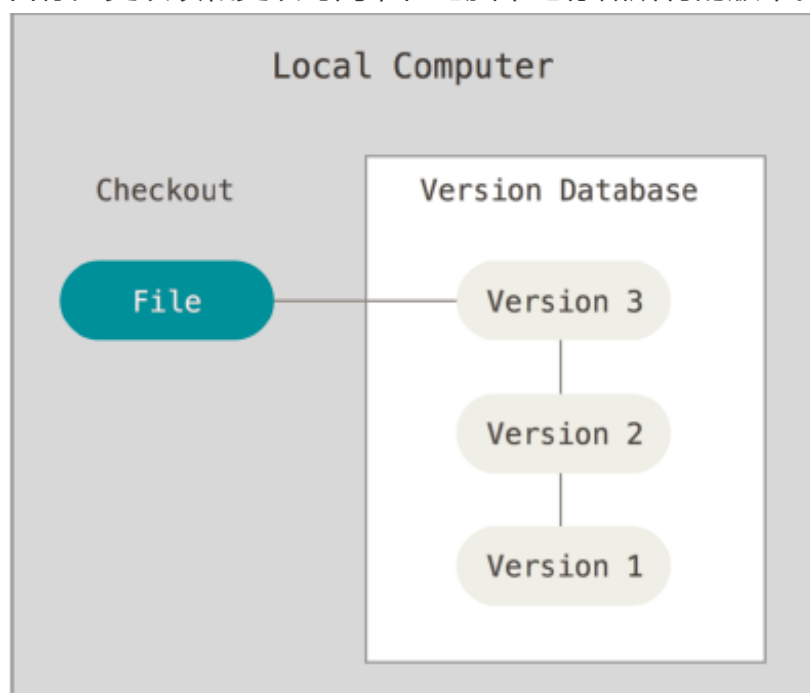
许多人刚开始学习git和github的时候，分不清git和github的关系，以为它们是一样的，我也犯过这个错误。那么git和github分别是什么呢？它们为什么总在一起称呼？

简单来说，git是一套管理软件开发版本的分布式控制系统，而GitHub是用来托管代码的网站。用户可以通过Git将软件代码上传到Github上，全世界的程序猿可以通过GitHub查看代码，使用Git共同开发不同版本的软件。

版本控制主要经历了三个时期，分别是本地版本控制，集中版本控制以及现在的分布式版本控制。下面分别来说说。

### 1.1 本地版本控制

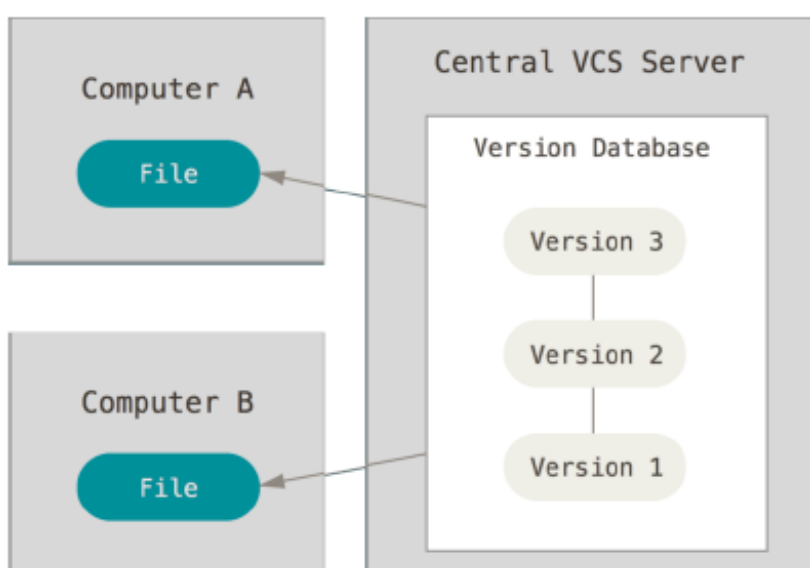
本地版本控制顾名思义，就在自己的电脑上进行版本控制，通过复制项目的整个目录以及给项目标注更改项和更改时间，在电脑本地存储所有的版本。具体框图如下：



然而这种版本不够灵活，不能进行不同系统的协同开发，于是下一个集中式版本控制应运而生。

## 1.2 集中式版本控制

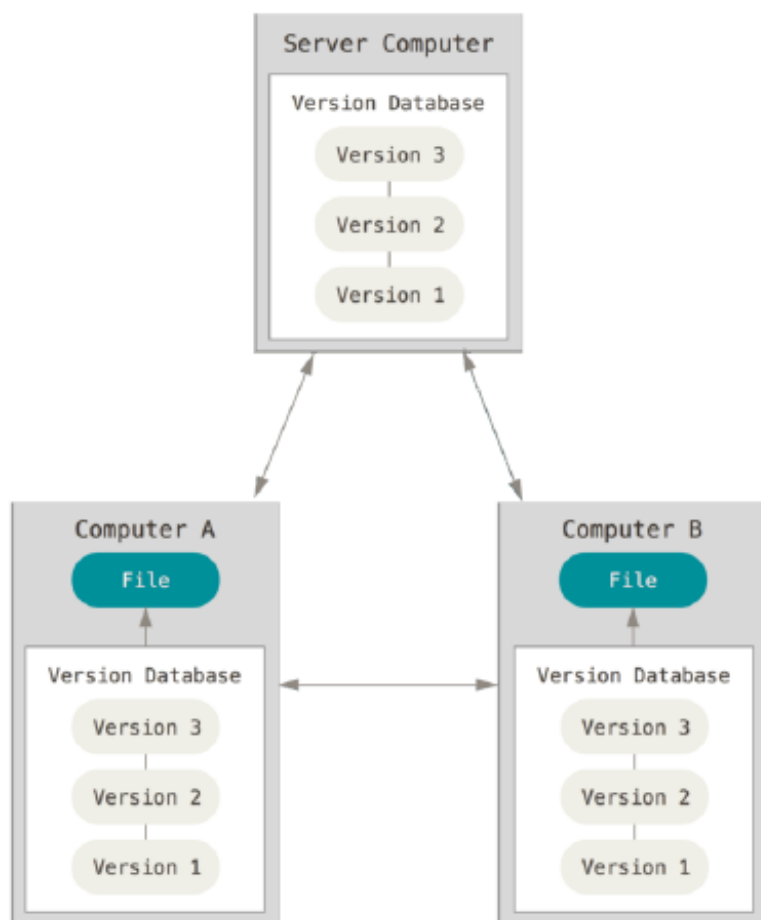
集中式版本控制则是将软件代码存在中央服务器中，开发者可以通过远程下载最新代码或者提交更新，多人协作更加方便。具体框图如下：



缺点是中央服务器万一崩溃或者数据库损坏，轻则宕机，重则失去所有代码和所有提交历史，无法恢复。这时候分布式版本控制就来发挥它的作用力。

## 1.3 分布式版本控制

分布式相比于集中式，就算服务器中的所有数据全部丢失，也可以通过本地的仓库进行恢复，因为每一次克隆，本地中都包含了软件的所有版本以及提交历史。具体框图如下：



## 2. Git安装

由于笔者没有接触过mac，所以只写window和linux下的安装方法。

**linux(基于Dedbian) 下：**

```
1. $ sudo apt-get install git
```


**window下：**


点击[链接](#)下载git，默认运行安装即可，另外提一句，git里可以指定默认编辑器，个人比


较喜欢VScode，所以比较推荐VScode啦！

### 3. 创建GitHub账号

进入[官网](#)后，点击sign up进行注册即可，会有邮箱认证填写经历什么的，这个很简单，不多说了。注册完了可以点击sign in进行登陆，浏览器记住账号即可。

 **Step 1:**  
Create personal account

 **Step 2:**  
Choose your plan

 **Step 3:**  
Tailor your experience

Create your personal account

**Username**  
  
This will be your username. You can add the name of your organization later.

**Email address**  
  
We'll occasionally send updates about your account to this inbox. We'll never share your email address with anyone.

**Password**  
  
Use at least one lowercase letter, one numeral, and seven characters.

**You'll love GitHub**

Unlimited collaborators

Unlimited public repositories

✓ Great communication

✓ Frictionless development

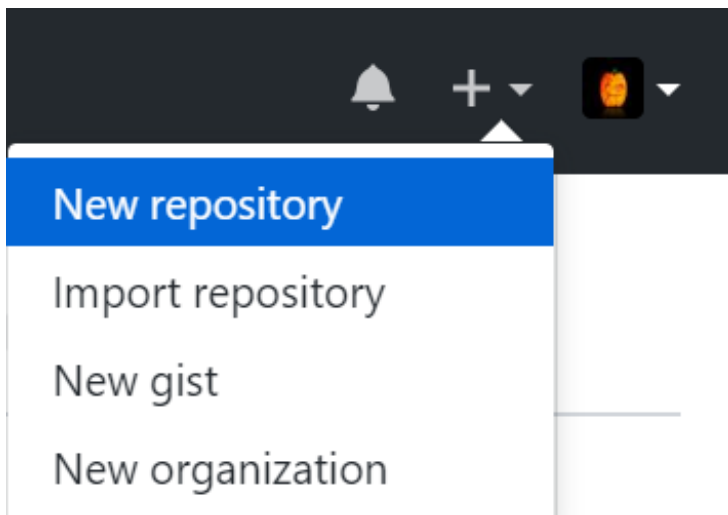
✓ Open source community

### 4. 第一个仓库

仓库的英文是repository，一般简说repo，代码存储在repo中，安装完git和创建完github账号，我们就可以创建我们自己的第一个repo啦！

#### 4.1 创建仓库

进入自己的github主页，点击自己图像旁边的加号，然后点击New repository，如图所示





然后在填写相关仓库的一些信息，然后点击creat repository

## Create a new repository

A repository contains all the files for your project, including the revision history.

Owner



Repository name

 FangYang970206 ▾ / playground 

Great repository names are short and memorable. Need inspiration? How about [super-duper-octo-couscous](#).

Description (optional)

write some notes for git

- ☒  **Public**  
Anyone can see this repository. You choose who can commit.
- ☐  **Private**  
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾ | Add a license: **None** ▾ 

Create repository

## 4.2 ssh key生成

git的远程管理是基于SSH的，所以需要进行SSH的配置，这样你才能访问自己的仓库。首先，在bash（window桌面右键有git bash，linux则直接终端进行即可）中设置Git的User name和email（注册名字和邮箱）：

```
1. $ git config --global user.name "FangYang970206"
2. $ git config --global user.email "15270989505@163.com"
```

然后，我们可以看看自己电脑里有没有ssh密钥，linux下是在 `/home/.ssh`，window是在 `C:\Users\Username\.ssh`，有则备份删除，然后在终端中运行



```
1. ssh-keygen -t rsa -C "15270989505@163.com"
```

按3个回车，密码为空，得到了两个文件：`id_rsa`和`id_rsa.pub`，然后打开`id_rsa.pub`，复制里面的内容，最后面的计算机名字不要复制，然后打开<https://github.com>，点击自己头像中的Setting，然后选择SSH and GPG keys，点击New SSH key，title随便写，下面的key粘贴刚才复制的内容，最后点击Add SSH key，成功SSH and GPG keys就会有SSH key的显示，如笔者界面所示(window和ubuntu各一个)

### SSH keys

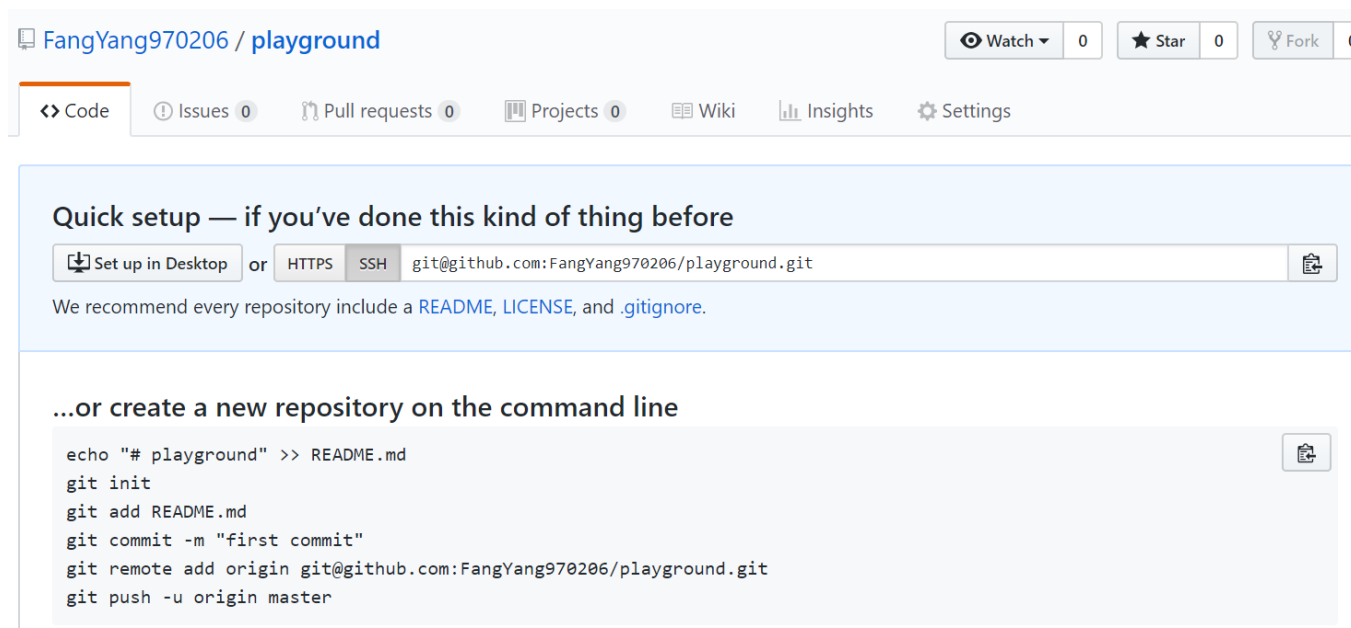
[New SSH key](#)

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

	<b>fangyang_ubuntu</b> [Redacted Key]	<a href="#">Delete</a>
SSH	Added on 13 Jul 2018 Last used within the last 2 months — Read/write	
	<b>fangyang_newcomputer</b> [Redacted Key]	<a href="#">Delete</a>
SSH	Added on 29 Jul 2018 Last used within the last week — Read/write	

## 4.3 git简单 workflow

点击我们刚刚创建的仓库，可以看到如下界面



首先我们在桌面创建一个文件夹，名字也取playground好了，我们可以安装上图的一个小教程来初步试试（仓库地址不同，请使用自己的地址哦）

```
1. echo "# playground" >> README.md
2. git init
3. git add README.md
4. git commit -m "first commit"
5. git remote add origin git@github.com:FangYang970206/playground.git
6. git push -u origin master
```

运行完上面的命令，我们重新打开我们刚才创建的仓库，你就会发现已经有所变化，我们已经把README.md文件上传到了我们的仓库中。

## 5. 使用git管理仓库

通过上面的一个小事例，没接触过git可能有比较多的疑问，下面我们来一步步进行讲解。

### 5.1 上节回顾

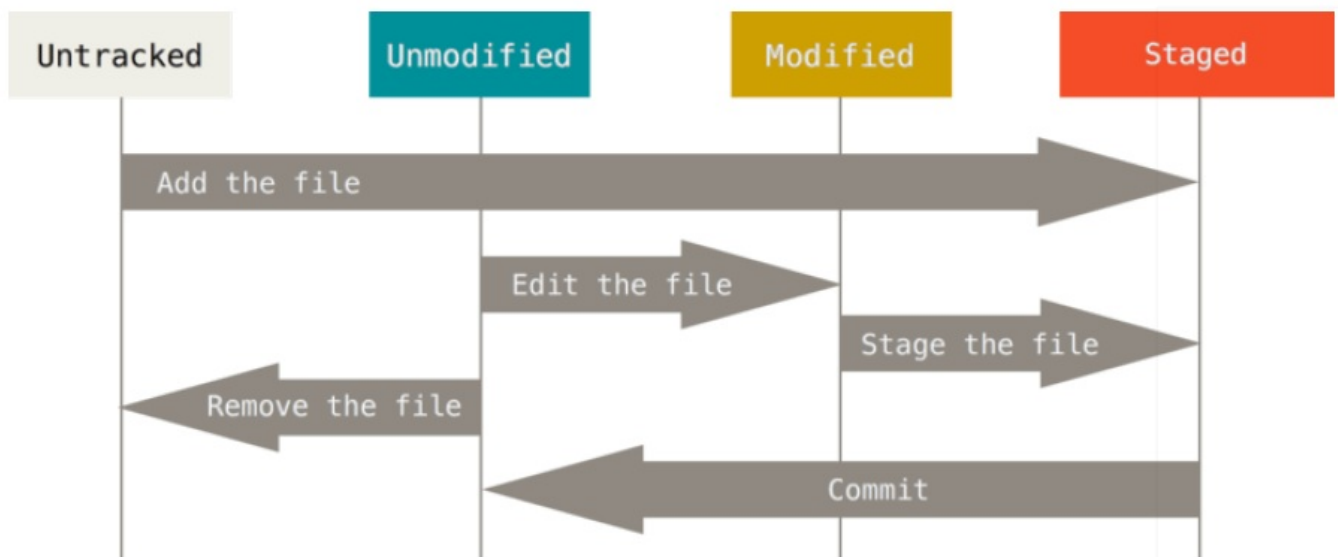
- 第一句 `echo "# playground" >> README.md` 不用多说，就是将使用echo命令将 `# playground` 写入 `README.md` 中



- 第二句 `git init` 是初始化本地仓库，会在当前目录产生 `.git` 文件夹，这是保存着所有git操作所需要的文件，是本地进行git的第一步（远程克隆仓库不需要这一步）
- 第三句 `git add README.md` 是将文件放入暂存区（stage），暂存区后面再说。
- 第四句 `git commit -m "first commit"` 是记录这次的更改，`-m` 后的字符串则是更改详情，在你的github仓库中，你也会看到 `README.md` 后面跟着 `"first commit"` 这句话。
- 第五句 `git remote add origin git@github.com:FangYang970206/playground.git` 是用来添加远程仓库的信息到本地，并用一个简短的引用来表示url，命令具体是 `git remote add <shortname> <url>`，也就是我们可以在接下来的git操作中，用 `origin` 来代表整个 `url`，你也可以取你自己感兴趣的简短名字
- 第六句 `git push -u origin master` 是将本地代码上传到github服务器，这句话要拆成两部分解释，第一部分是 `git push`，是上传命令，那上传到哪里呢？第二部分 `-u origin master` 则是指定上传位置，上传到 `origin` 的 `master` 分支，这里的 `-u` 是设定默认主机，也就是下次你要是也上传到 `origin` 的 `master` 分支，就直接 `git push` 就可以了

## 5.2 文件跟踪

仓库中的文件状态无非两种，一种是未被git跟踪（untracked），另一种是被git跟踪（tracked），对于从远程服务器中克隆出的仓库，默认全部文件都进行跟踪，而本地自己新建的仓库，则需要通过 `git add` 命令将未被git跟踪的文件变为被git跟踪的文件。而被git跟踪的文件有三种状态，分别是未修改（unmodified）、已修改（modified）和暂存区（staged）。以5.1节的为例，我们先新建了README.md文件，这个文件处于未跟踪状态，然后初始化仓库后，我们通过 `git add` 命令将未跟踪状态的文件转到跟踪状态，并将文件加入到暂存区。然后通过 `git commit` 命令将暂存区状态转成已修改文件。下图形象地表示了文件地状态转换。

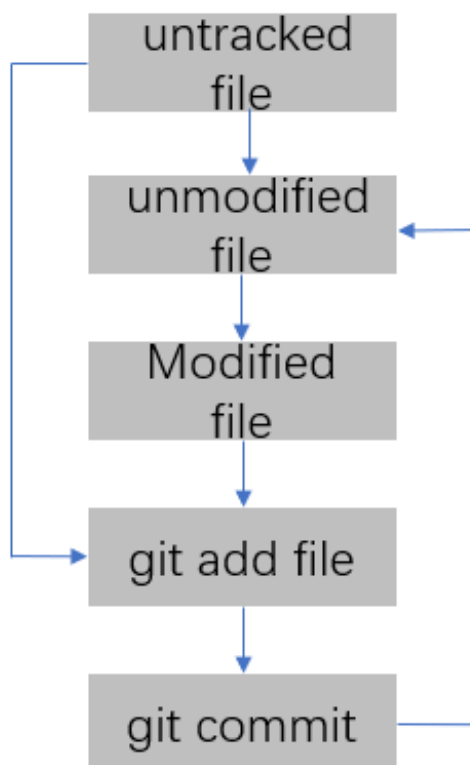


当你将文件转成跟踪状态时，文件如果没有人为移除，将一直处于跟踪状态，从未修改、已修改和暂存区三个状态反复转换，永不丢失。

## 5.3 git常用命令

### 5.3.0 git add和 git commit

上面的 `git add` 和 `git commit` 是最重要的两个命令，它是整个版本控制中最常用的两个命令，本地版本控制流程如下：



下面介绍一些 `git add` 常用命令

```
1.  git add <file>           #添加单个文件
2.  git add <dir_name>       #添加整个文件夹
3.  git add *.py             #添加所有py文件
4.  git add .                #提交被修改的和新建的文件，但不包括被删除的文件
5.  git add -A               #提交所有变化
6.  git add -h               #git add帮助命令
```

然后介绍一下 `git commit` 常用命令

```
1.  git commit -m <message> #提交暂存区更改
2.  git commit -am <message> #越过暂存区,不需要git add, 直接提交所有tracked文件
3.  git commit --amend       #追加提交,不引入新的commit,追加到前一次commit,commit信息还可修改
4.  git commit -h            #更多commit命令
```

这里的 `git commit --amend` 很有用，有时候我们提交时忘记了把一些文件加到暂存区一起提交，可又不想引入新的提交记录这次信息，就可以使用 `git commit --amend`，如下所示：

```
1.  $ git commit -m 'initial commit'
2.  $ git add forgotten_file
3.  $ git commit --amend
```

执行 `git commit --amend` 会打开编辑器（一般是vim），你可以修改提交说明（i进入插入模式修改），然后保存退出即可（Esc,然后shift+:,打wq,最后enter）

### 5.3.1 git status

知道文件的状态，我们通过一些实例来学习一下。在playground仓库新建 `hello.py`，然后加入下面一行：

```
1.  print("hello everyone")
```

`git status` 命令会显示当前仓库的文件状态，在终端中输入 `git status`，显示如下内容：

```
1.  $ git status
2.  On branch master
```

```
3. Your branch is up to date with 'origin/master'.
4.
5. Untracked files:
6.   (use "git add <file>..." to include in what will be committed)
7.
8.     hello.py
9.
10. nothing added to commit but untracked files present (use "git add" to track)
```

可以看到我们刚刚新建的 `hello.py` 处于 `untracked files`。然后我们在终端中输入 `git add hello.py`，然后我们再次输入 `git status`，会出现下面内容：

```
1. $ git add hello.py
2. warning: LF will be replaced by CRLF in hello.py.
3. The file will have its original line endings in your working directory
4.
5. $ git status
6. On branch master
7. Your branch is up to date with 'origin/master'.
8.
9. Changes to be committed:
10.   (use "git reset HEAD <file>..." to unstage)
11.
12.     new file:   hello.py
```

可以看到 `hello.py` 的状态变为 `Changes to be committed`，这意味着 `hello.py` 进入了暂存区，我们再用 `git commit` 提交这次更改，然后再用 `git status` 查看状态，结果如下：

```
1. $ git commit -m "add hello everyone"
2. [master 405cd1b] add hello everyone
3. 1 file changed, 1 insertion(+)
4. create mode 100644 hello.py
5.
6. $ git status
7. On branch master
8. Your branch is ahead of 'origin/master' by 1 commit.
9.   (use "git push" to publish your local commits)
10.
11. nothing to commit, working tree clean
```

可以看到显示 `nothing to commit`，文件状态处于未更改状态，因为我们完成了这一次版本的提交。最后我们可以使用 `git push` 上传我们这次的提交到远程github服务器上。最后可以看到github的仓库中多了我们刚才修改的 `hello.py`。

### 5.3.2 git diff

`git status` 命令的输出可能过于模糊，如果你想知道具体修改了什么地方，可以用 `git diff` 命令。它用来回答两个问题：当前做的哪些更新还没有暂存？有哪些更新已经暂存起来准备好了下次提交？代码修改运行出错，有时候可以用 `git diff`，可以看到自己或他人新加入了那些行，有助于修复bug和多人合作。我们在 `hello.py` 文件再加入一行：

```
1.  print("hello fang")
```

然后在终端中输入 `git diff`，结果如下：

```
1.  $ git diff
2.  warning: LF will be replaced by CRLF in hello.py.
3.  The file will have its original line endings in your working directory
4.  diff --git a/hello.py b/hello.py
5.  index 1ad4063..d469b07 100644
6.  --- a/hello.py
7.  +++ b/hello.py
8.  @@ -1,2 @@
9.   print("hello everyone")
10. +print("hello yang")
```

+号代表新添加的行，然后我们再加入一行：

```
1.  print("hello fang")
```

然后再调用 `git diff` 命令，结果如下：

```
1.  $ git diff
2.  diff --git a/hello.py b/hello.py
3.  index 97d6fb2..2595b9c 100644
4.  --- a/hello.py
```

```
5.    +++ b/hello.py
6.    @@ -1 +1,3 @@
7.     print("hello everyone")
8.     +print("hello yang")
9.     +print("hello fang")
```

可以看到我们又多加了一行，我们将文件状态转到暂存区(使用 `git add hello.py`)看看，然后运行 `git diff`，可以看到是没有任何输出的，因为 `git diff` 本身只显示尚未暂存的改动。如果想看暂存前后的变化，则需要使用 `git diff --cached` 命令，则可以看到：

```
1.    $ git diff --cached
2.    diff --git a/hello.py b/hello.py
3.    index 97d6fb2..2595b9c 100644
4.    --- a/hello.py
5.    +++ b/hello.py
6.    @@ -1 +1,3 @@
7.     print("hello everyone")
8.     +print("hello yang")
9.     +print("hello fang")
```

这里提一句，有时候我们运行 `git status`，我们会看到一个文件显示两种状态，比如我们上面加了一行 `print("hello FY")`，然后运行 `git status`，可以看到如下结果：

```
1.    $ git status
2.    On branch master
3.    Your branch is ahead of 'origin/master' by 1 commit.
4.    (use "git push" to publish your local commits)
5.
6.    Changes to be committed:
7.    (use "git reset HEAD <file>..." to unstage)
8.
9.            modified:   hello.py
10.
11.    Changes not staged for commit:
12.    (use "git add <file>..." to update what will be committed)
13.    (use "git checkout -- <file>..." to discard changes in working
14.    directory)
15.
16.            modified:   hello.py
```

出现这种结果的原因是因为我们的文件确实存在两种状态，一种是我们之前加入到暂存区，一种是我们刚刚修改添加的，两者是可以共存的。

### 5.3.3 git log

`git log` 命令会列出每个提交的SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。我们已经提交了许多次了，在bash中输入 `git log`，出现以下内容：

```
1.  $ git log
2.  commit a9b674c88eba0043f84aec4668215358f99c5572 (HEAD -> master)
3.  Author: FangYang970206 <15270989505@163.com>
4.  Date:   Mon Aug 20 16:04:09 2018 +0800
5.
6.      add some info
7.
8.  commit 2ae267c61261b6041d16133cca56f4c8155d73fa
9.  Author: FangYang970206 <15270989505@163.com>
10. Date:   Mon Aug 20 10:02:57 2018 +0800
11.
12.      fix one error
13.
14. commit 405cd1bd4b9e0d19f1698c7f7cb8f77184424040 (origin/master, origin
15. /HEAD)
16. Author: FangYang970206 <15270989505@163.com>
17. Date:   Sun Aug 19 21:49:29 2018 +0800
18.
19.      add hello everyone
20.
21. commit edb1d60a14b25be099205a62a7c469083dc1338a
22. Author: FangYang970206 <15270989505@163.com>
23. Date:   Fri Aug 17 17:41:25 2018 +0800
24.
25.      first commit
```

SHA-1校验和（也叫hash，哈希）这里提一句，在git中，文件的存储是通过文件内容计算哈希值（40位十六进制）进行索引的，所以不可能在git不知情的情况下修改文件，确保记录完备，提交更改会产生哈希值记录此次更改。

`git log` 其他常用命令：

1. `git log -n`            #n是整数，返回最近n次提交历史
2. `git log -p -n`        #用来显示每次提交的内容差异，-n则返回最近n次提交的差异

3. `git log --stat` #列出所有被修改过的文件、有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。
4. `git log --pretty=format` #按照指定格式展示历史
5. `git log --oneline --decorate` #显示分支的指向情况，见分支的使用
6. `git log --oneline --decorate --graph --all` #输出提交历史、各个分支的指向以及项目的分支分叉情况。

个人觉得 `git log --pretty=format` 非常有趣，可以很简洁展示历史，举例来说，我们在终端运行 `git log --pretty=format:"%h %s"`，有如下结果：

```
1. $ git log --pretty=format:"%h %s"
2. a9b674c add some info
3. 2ae267c fix one error
4. 405cd1b add hello everyone
5. edb1d60 first commit
```

第一项是简短哈希，第二项是提交说明，非常直观。常见的format如下。

**Table 1.** `git log --pretty=format` 常用的选项

选项	说明
<code>%H</code>	提交对象（commit）的完整哈希字符串
<code>%h</code>	提交对象的简短哈希字符串
<code>%T</code>	树对象（tree）的完整哈希字符串
<code>%t</code>	树对象的简短哈希字符串
<code>%P</code>	父对象（parent）的完整哈希字符串
<code>%p</code>	父对象的简短哈希字符串
<code>%an</code>	作者（author）的名字
<code>%ae</code>	作者的电子邮件地址
<code>%ad</code>	作者修订日期（可以用 <code>--date=</code> 选项定制格式）
<code>%ar</code>	作者修订日期，按多久以前的方式显示
<code>%cn</code>	提交者（committer）的名字
<code>%ce</code>	提交者的电子邮件地址
<code>%cd</code>	提交日期
<code>%cr</code>	提交日期，按多久以前的方式显示
<code>%s</code>	提交说明

### 5.3.4 git rm和git mv

`git rm` 命令用来删除文件，对文件不进行版本管理。`git mv` 命令可以对文件重命名和移动。



使用 `git rm` :

```
1. $ git rm README.md
2. rm 'README.md'
3. $ git status
4. On branch master
5. Your branch is ahead of 'origin/master' by 2 commits.
6.   (use "git push" to publish your local commits)
7.
8. Changes to be committed:
9.   (use "git reset HEAD <file>..." to unstage)
10.
11.       deleted:    README.md
12. $ git commit -m "delete README.md"
13. [master b976c0a] delete README.md
14. 1 file changed, 1 deletion(-)
15. delete mode 100644 README.md
```

通过 `git rm` 后, 可以在playground文件夹中看到README.md文件已经不见了。

常用的 `git rm` 命令有 :

```
1. git rm -r <dirname>      #递归删除文件夹中的文件
2. git rm \*.c              #删除所有后缀.c文件
3. git rm -f filename       #强制删除, 针对修改并放到暂存区的文件
4. git rm --cached README    #不删除文件, 仍然放在目录中, 但不进行跟踪
```

使用 `git mv` (先在playground文件夹新建dd文件夹) :

```
1. $ git mv hello.py dd/hello.py
2.
3. $ git status
4. On branch master
5. Your branch is ahead of 'origin/master' by 3 commits.
6.   (use "git push" to publish your local commits)
7.
8. Changes to be committed:
9.   (use "git reset HEAD <file>..." to unstage)
10.
11.       renamed:    hello.py -> dd/hello.py
12.
13. $ git mv dd/hello.py dd/hello.txt
```

```
14.
15. $ git status
16. On branch master
17. Your branch is ahead of 'origin/master' by 3 commits.
18.   (use "git push" to publish your local commits)
19.
20. Changes to be committed:
21.   (use "git reset HEAD <file>..." to unstage)
22.
23.       renamed:    hello.py -> dd/hello.txt
24.
25. $ git commit -m "rename hello.py -> hello.txt"
```

`git mv` 相当于以下三条命令：

```
1. $ mv README.md README
2. $ git rm README.md
3. $ git add README
```

当然，能有简洁的命令当然用简洁的好！

### 5.3.5 远程命令

这一节涉及到分支，建议跳过这一节，看完分支再过来。

`git clone` 是用来克隆远程仓库到本地；

`git remote` 是用来添加远程仓库，并为远程仓库添加名字缩写；

`git push` 是上传本地仓库到远程仓库中。

`git fetch` 是将远程仓库的更新下载到本地仓库中

`git pull` 是将远程仓库的更新下载到本地仓库中，并进行合并

#### 5.3.5.1 git clone

克隆可以通过https url下载或者ssh url，命令很简单

```
1. $ git clone url
```

#### 5.3.5.2 git remote

`git remote add origin git@github.com:FangYang970206/playground.git` 是用来添加远程仓库的信息到本地，并用一个简短的引用来表示url，命令具体

是 `git remote add <shortname> <url>`，也就是我们可以在接下来的git操作中，用 `origin` 来代表整个 `url`。  
常用命令：

```
1. $ git remote -v #显示所有远程仓库的简写和url
```

### 5.3.5.3 git push

`git push -u origin master` 是将本地代码上传到github服务器，这句话要拆成两部分解释，第一部分是 `git push`，是上传命令，那上传到哪里呢？第二部分 `-u origin master` 则是指定上传位置，上传到 `origin` 的 `master` 分支，这里的 `-u` 是设定默认主机，也就是下次你要是也上传到 `origin` 的 `master` 分支，就直接 `git push` 就可以了。

其他命令：

```
1. $ git push origin <source>:<destination> #来源和去向可不同
```

如果想把本地的foo分支推送到远程仓库中的bar分支，则可用这个命令进行指定。

### 5.3.5.3 git fetch

`git fetch` 命令默认是将github中获取最新的版本到本地分支，默认是获取最新的 `origin/master`，然后比较本地的 `master` 分支和 `origin/master` 分支的差别，进行差异合并。

更一般的命令：

```
1. $ git fetch origin <remote branch>:<local branch> #分支
```

### 5.3.5.4 git pull

`git pull` 命令是两个命令的合并

```
1. $ git pull origin <remote branch>:<local branch>
2. #equal
3. $ git fetch <remote branch>:<local branch>
4. $ git merge <remote branch> #on local branch
```

### 5.3.6 git reset和git revert

`git reset` 和 `git revert` 命令是用来撤销变更的，常用命令如下（关于HEAD的解释，请看分支一节）：

```
1. $ git reset                #取消所有暂存文件
2. $ git reset HEAD <file>    #取消某一暂存文件、
3. $ git reset HEAD~n         #回溯n个提交
4. $ git reset --hard HEAD~n  #强制回溯n个提交
5. $ git revert HEAD~n        #新的提交与HEAD~(n+1)的内容一模一样
```

使用 `git reset` 命令回溯到某个历史提交，不会保留后面的提交历史，而 `git revert` 命令则是创建与回溯历史一样的提交。

### 5.3.7 git别名

有一个小技巧可以使你的 Git 体验更简单、容易、熟悉：别名。可以通过 `git config` 文件来轻松地给每一个命令设置一个别名。以下是一些实例。

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

这样，就可以用 `git co` 代表 `git commit`，`git br` 代表 `git branch` 等等。取消别名使可使用如下命令：

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

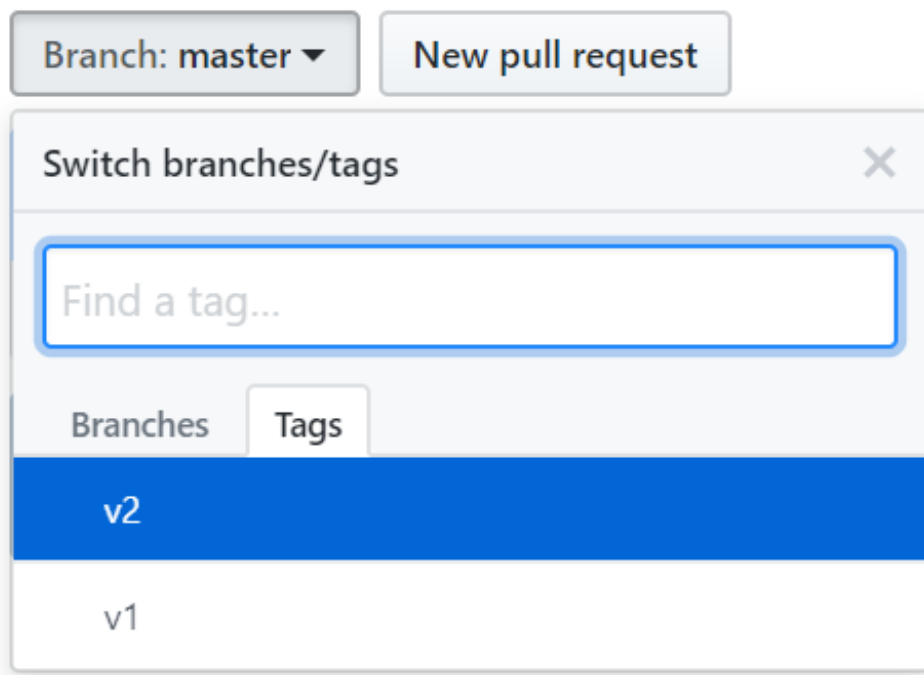
### 5.3.8 git tag

Git 可以给历史中的某一个提交打上标签，以示重要。比较有代表性的是人们会使用这个功能来标记发布结点。经常可以看某些软件库经常发x.x.x版本。

```
1. $ git log --pretty=format:"%h %s"
2. b976c0a delete README.md
3. b4d7987 add some info
4. 2ae267c fix one error
5. 405cd1b add hello everyone
6. edb1d60 first commit
7.
```

```
8. $ git tag v1 b976c0a
9.
10. $ git push origin v1
```

通过以上命令，就可以在远程仓库tag下有v1版本，如图所示（ps：我多tag了一个v2）：



### 5.3.9 .gitignore文件

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

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 所使用的简化了的正则表达式。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（这个例子要么匹配一个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（比如 [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。

.gitignore 文件的例子：

```
1.  # no .a files
2.  *.a
3.  # but do track lib.a, even though you're ignoring .a files above
4.  !lib.a
5.  # only ignore the TODO file in the current directory, not subdir/TODO
6.  /TODO
7.  # ignore all files in the build/ directory
8.  build/
9.  # ignore doc/notes.txt, but not doc/server/arch.txt
10. doc/*.txt
11. # ignore all .pdf files in the doc/ directory
12. doc/**/*.pdf
```

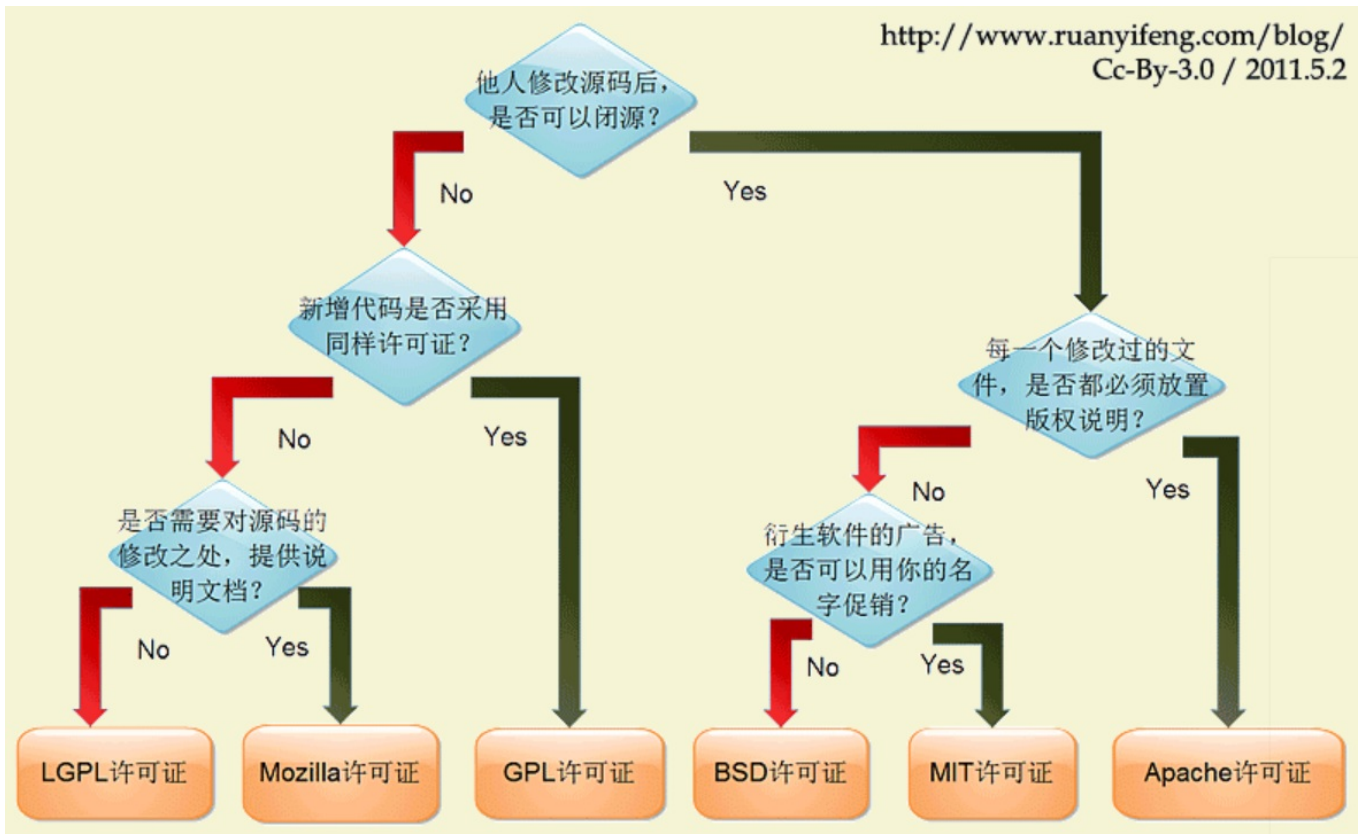
TIP: GitHub 有一个十分详细的针对数十种项目及语言的 .gitignore 文件列表，你可以在<https://github.com/github/gitignore> 找到它。

### 5.3.10 LICENSE文件

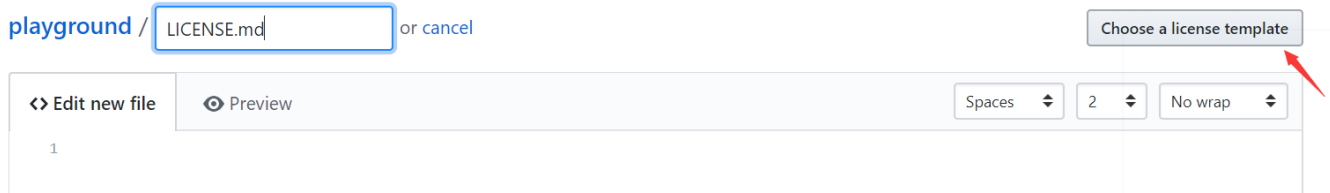
这一小节是最后加上的，可能会与后面的章节的LICENSE冲突，这点注意一下就好。

LICENSE文件是一种开源许可证，即授权条款。开源软件并非完全没有限制。最基本的限制，就是开源软件强迫任何使用和修改该软件的人承认发起人的著作权和所有参与人的贡献。任何人拥有可以自由复制、修改、使用这些源代码的权利，不得设置针对任何人或团体领域的限制；不得限制开源软件的商业使用等。而许可证就是这样一个保证这些限制的法律文件。

开源许可证有上百种，这里说说最流行的六种：GPL、BSD、MIT、Mozilla、Apache和LGPL如何做选择，阮一峰在如何选择开源许可证一文中给出了一张图，直观精确，就是下图：



那么如何给自己的仓库加上LICENSE呢？很简单，点击仓库中的creat new file，然后写LICENSE.md，选择choose a license template



然后选择MIT License，再点击Review and Submit，最后点击commit change就可以了。

Add a license to your project

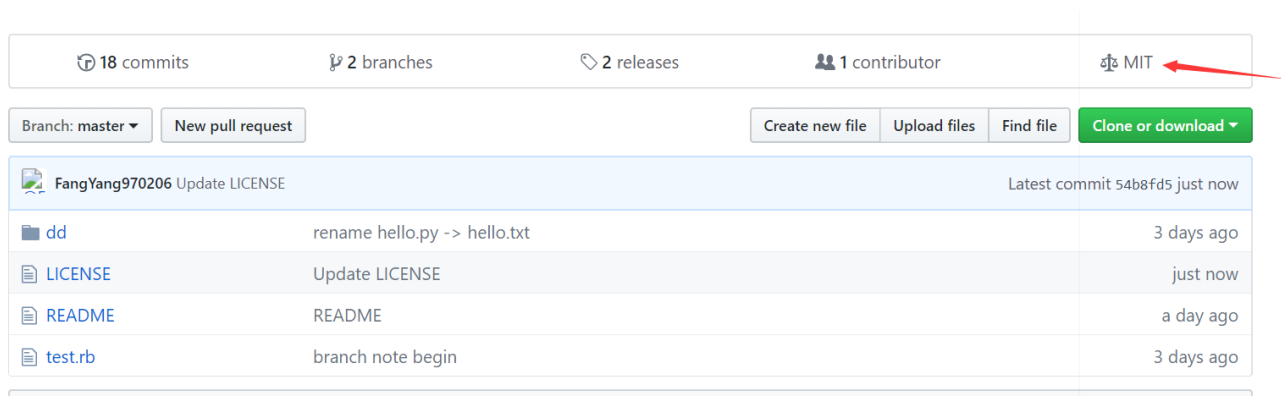
Apache License 2.0	<p>A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.</p> <table><thead><tr><th>Permissions</th><th>Limitations</th><th>Conditions</th></tr></thead><tbody><tr><td>✓ Commercial use</td><td>✗ Liability</td><td>① License and copyright notice</td></tr><tr><td>✓ Modification</td><td>✗ Warranty</td><td></td></tr><tr><td>✓ Distribution</td><td></td><td></td></tr><tr><td>✓ Private use</td><td></td><td></td></tr></tbody></table> <p>This is not legal advice. <a href="#">Learn more about repository licenses.</a></p>	Permissions	Limitations	Conditions	✓ Commercial use	✗ Liability	① License and copyright notice	✓ Modification	✗ Warranty		✓ Distribution			✓ Private use			<p>To adopt MIT License, enter your details. You'll have a chance to review before committing a <i>LICENSE</i> file to a new branch or the root of your project.</p> <p>Year ① 2018</p> <p>Full name ① Yang Fang (JoJo)</p> <p><a href="#">Review and submit</a></p>
Permissions		Limitations	Conditions														
✓ Commercial use		✗ Liability	① License and copyright notice														
✓ Modification		✗ Warranty															
✓ Distribution																	
✓ Private use																	
GNU General Public License v3.0																	
MIT License																	
BSD 2-Clause "Simplified" License																	
BSD 3-Clause "New" or "Revised" License																	
Eclipse Public License 2.0																	
GNU Affero General Public License v3.0																	
GNU General Public License v2.0																	

MIT License

Copyright (c) 2018 Yang Fang (JoJo)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to

我们就可以在自己仓库中看到MIT协议了



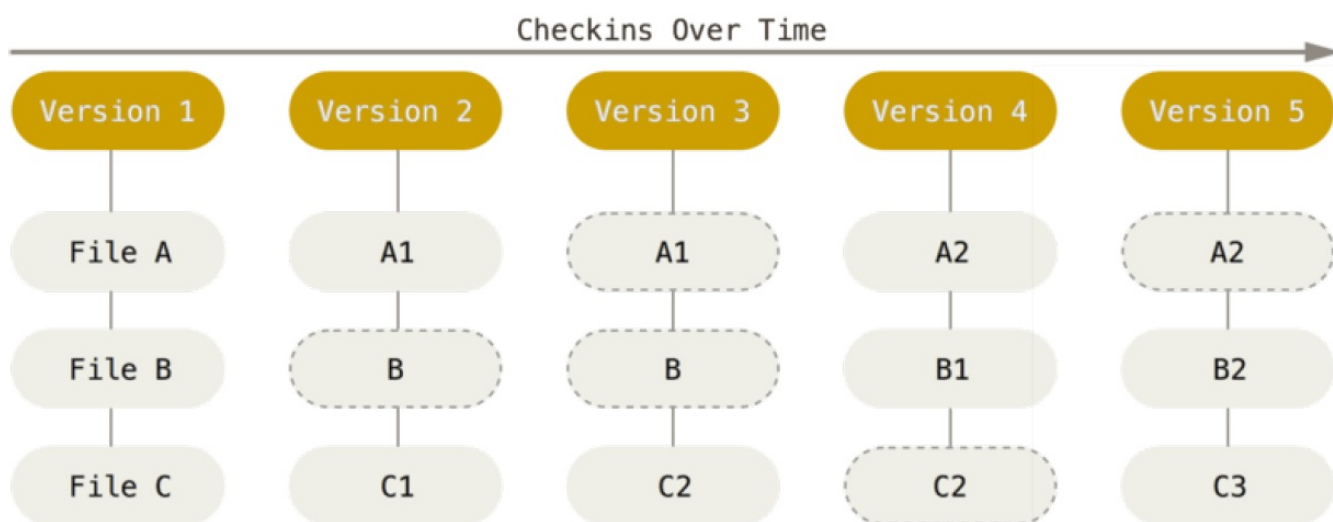
## 6. 分支的使用

有人称分支是git的必杀技，正是因为这一特性，git从众多版本管理系统中脱颖而出。git鼓励多次使用分支和合并。精通分支，将对你的版本管理十分便捷和高效。进行分支之前，先讲git是如何保存数据的。

Note: 为了节省工作量，分支中有的图是截的书上的图，图中的校验和会与实际的校验和不同，这点注意一下就好。

### 6.1 git保存数据方式与分支简介

与一些版本控制软件不同，Git保存的不是文件的变化或者差异，而是一系列不同时刻的文件快照。如下图所示



在进行提交操作时，Git 会保存一个提交对象（commit object）。知道了 Git 保存数据的方式，我们可以很自然的想到，该提交对象会包含一个指向暂存内容快照的指针。但不仅仅是这



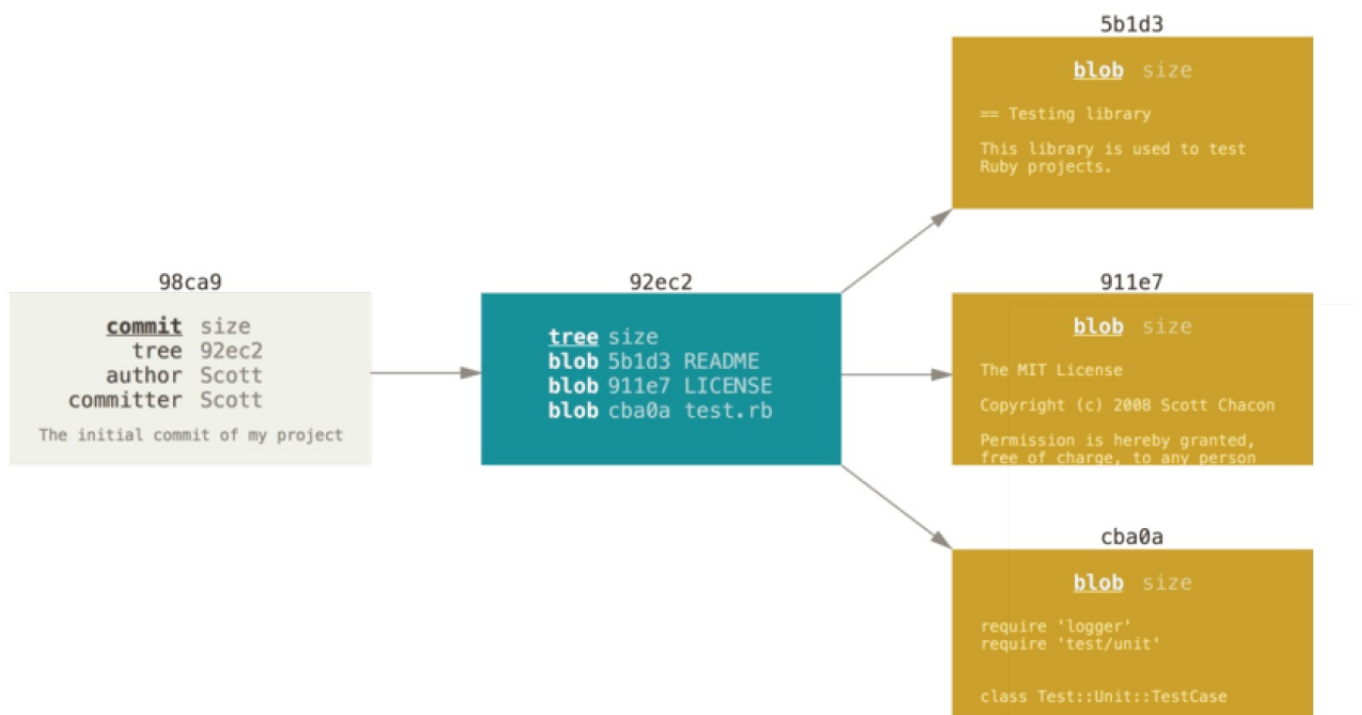
样，该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。

首次提交产生的提交对象没有父对象，普通提交操作产生的提交对象有一个父对象，而由多个分支合并产生的提交对象有多个父对象。

为了更加形象地说明，我们假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件(文件需要自己动手在playground目录新建)。暂存操作会为每一个文件计算校验和，然后会把当前版本的文件快照保存到Git仓库中（Git使用blob对象来保存它们），最终将校验和加入到暂存区域等待提交：

```
1. $ git add README test.rb LICENSE
2. $ git commit -m 'branch note begin'
```

现在，Git仓库中有五个对象（忽略dd文件夹）：三个blob对象（保存着文件快照）、一个树对象（记录着目录结构和blob对象索引）以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）。具体结构如下图：



做些修改后再次提交（两次）：

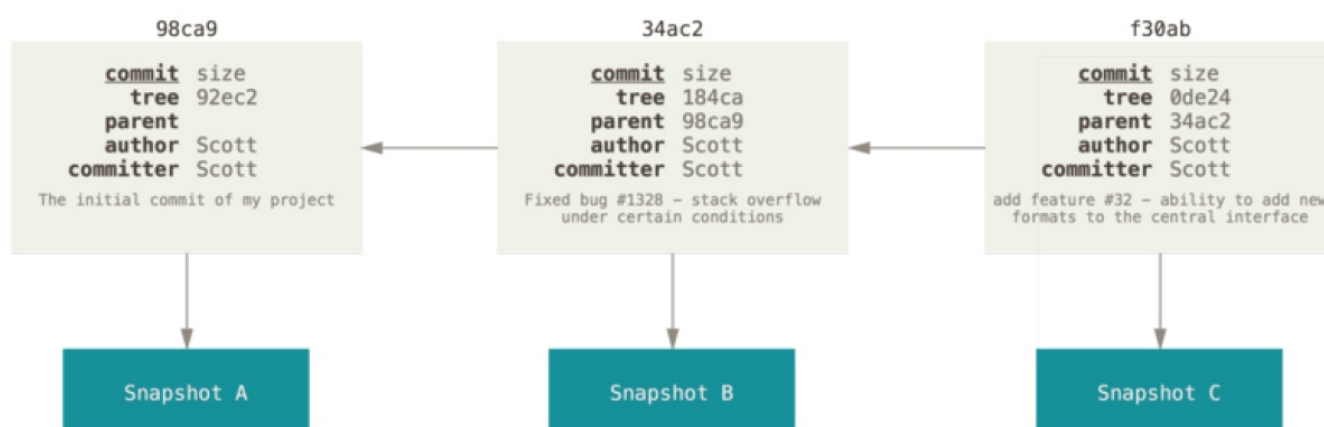
```
1. $ echo "print('1')" >> README
2.
3. $ git commit -am "add print('1') into README"
```

```

4.  warning: LF will be replaced by CRLF in README.
5.  The file will have its original line endings in your working directory
6.  [master 326dd0b] add print(1) into README
7.    1 file changed, 1 insertion(+)
8.
9.  $ echo "print("2")" >> README
10.
11. $ git commit -am "add print("2") into README"
12. warning: LF will be replaced by CRLF in README.
13. The file will have its original line endings in your working directory
14. [master ebc9b45] add print(2) into README
15.    1 file changed, 1 insertion(+)

```

两次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。见下图：



Git的分支，其实本质上仅仅是指向提交对象的可变指针。Git的默认分支名字是master。在多次提交操作之后，你其实已经有一个指向最后那个提交对象的master分支。它会在每次的提交操作中自动向前移动。

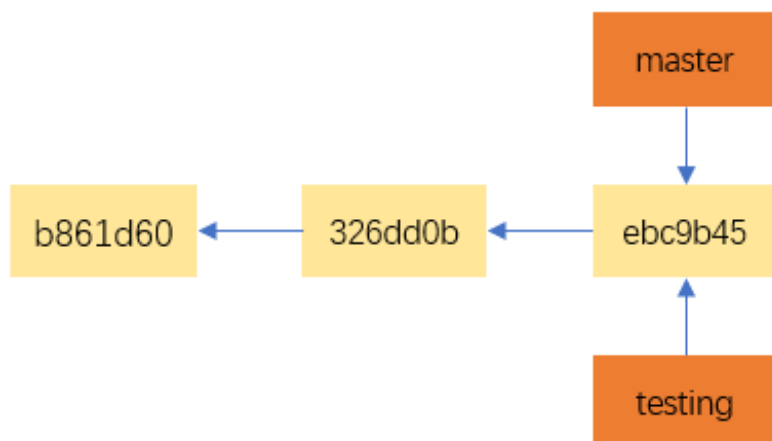
- Git的“master”分支并不是一个特殊分支。它就跟其它分支完全没有区别。之所以几乎每一个仓库都有 master 分支,是因为git init命令默认创建它，并且大多数人都懒得去改动它。

## 6.2 创建分支

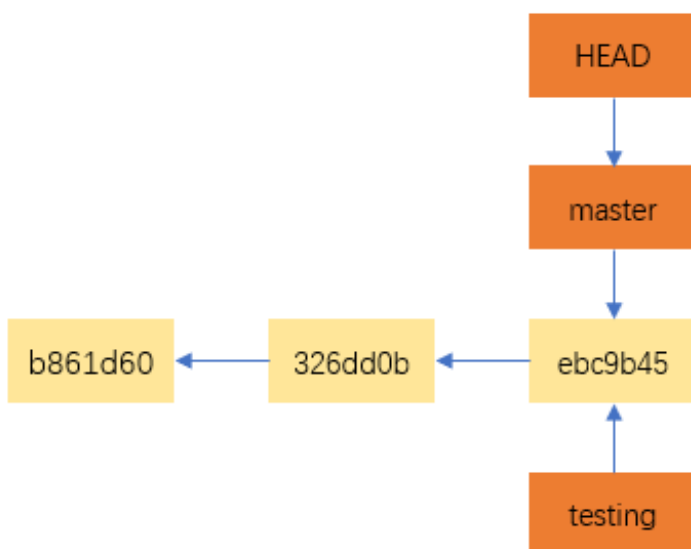
使用 `git branch` 命令可以很简单地创建分支，比如创建一个testing分支。

```
1. $ git branch testing
```

这会在当前所在的提交对象上创建一个指针。如下图所示



那么，Git又是怎么知道当前在哪一个分支上呢？很简单，它有一个名为HEAD的特殊指针。这个特殊指针就是用来指定当前所在的分支。



可以使用 `git log` 命令查看各个分支当前所指的提交对象。

```
1. $ git log --oneline --decorate -3 #只看倒数3个
2. ebc9b45 (HEAD -> master, testing) add print(2) into README
3. 326dd0b add print(1) into README
4. b861d60 branch note begin
```

可以看到testing，master和现在指向master的HEAD都指向最后的提交。

## 6.3 切换分支

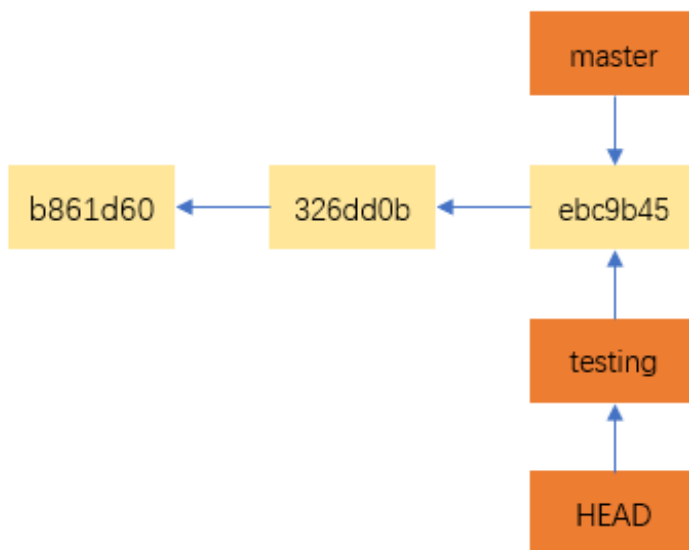
创建好了分支，可以用 `git checkout` 命令进行切换

```
1. $ git checkout testing
2. Switched to branch 'testing'
3. $ git branch    #可以使用git branch查看当前分支（前面带*号）
4.     master
5.     * testing
```

有一个简单的命令可以快速创建新的分支并切换到新的分支：

```
1. $ git checkout -b new_branch
2. #等效于
3. $ git branch new_branch
4. $ git checkout new_branch
```

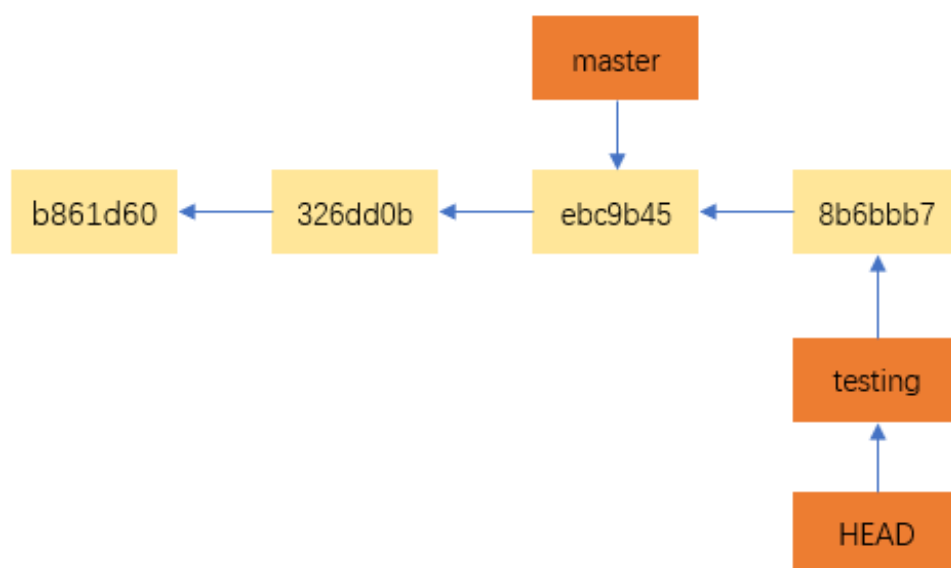
于是HEAD指针也发生了变化，如下图



在testing分支进行一次提交：

```
1. $ echo "print("3")" >> README
2.
3. $ git commit -am "add print("3") into README"
4. warning: LF will be replaced by CRLF in README.
5. The file will have its original line endings in your working directory
6. [testing 8b6bbb7] add print(3) into README
7. 1 file changed, 1 insertion(+)
```

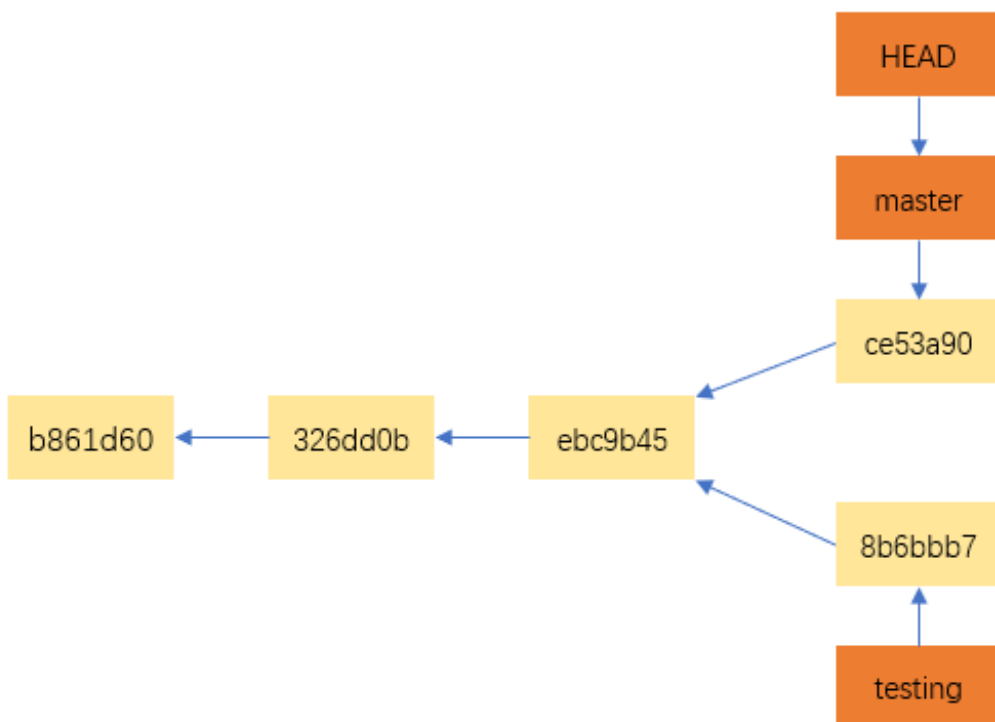
git的提交记录以及分支指向如下：



可以看到testing和HEAD都移动到前面了，而master没有移动。现在我们切换到master分支，对master进行一次提交。

```
1. $ echo "MIT LICENSE" >> LICENSE
2.
3. $ git commit -am "add MIT LICENSE"
4. warning: LF will be replaced by CRLF in LICENSE.
5. The file will have its original line endings in your working directory
6. [master ce53a90] add MIT LICENSE
7. 1 file changed, 1 insertion(+)
```

提交后，上图就变成了下图



可以使用 `git log` 命令查看提交历史

```
1. $ git log --oneline --decorate --graph --all -5 #只看倒数五个
2. * ce53a90 (HEAD -> master) add MIT LICENSE
3. | * 8b6bbb7 (testing) add print(3) into README
4. |/
5. * ebc9b45 add print(2) into README
6. * 326dd0b add print(1) into README
7. * b861d60 branch note begin
```

可以看到图中的看到结构与上图相同。

由于Git的分支实质上仅是包含所指对象校验和（长度为40的SHA-1值字符串）的文件，所以它的创建和销毁都异常高效，也就是添加或者删除41个字节的速度，能做到这一点的原因是因为git是以文件快照的形式保存文件，所以创建分支只需创建一个新的指针指向快照即可，而其他的一些版本管理软件往往需要将整个项目复制到另一个目录，这就比较慢了。

## 6.4 为什么要使用分支？

学了上面的东西，可能会想，为什么要使用分支？可以通过现实场景的问题来回答这个问题。

从个人角度，你正在开发一个网站，网站已经处于正常运行状态，我们想在网站中加入新的功能，这时候你有两种选择：一是直接在当前master上进行修改和测试，二是创建一个分支。选

择一可能会产生影响正常工作的代码，这是我們不想看到的。而选择二创建分支可以很方便地解决这个问题，分支不会影响当前工作的分支，可以很放心地进行开发和测试，最后对原工作分支进行合并即可。

从多人协作角度上看，这个更加直接，在一条流水线上不仅效率低，而且会产生很多混乱，比如不同人代码水平有限，有的还会编写错误的代码，使用分支可以让软件维护者很方便地查看不同的分支情况，选择合适地分支进行合并。

另外，分支的创建是非常快的，只需创建一个新的指针即可，切换分支也非常地快，这可以让我们很灵活而不受干扰地工作。

## 6.5 合并分支

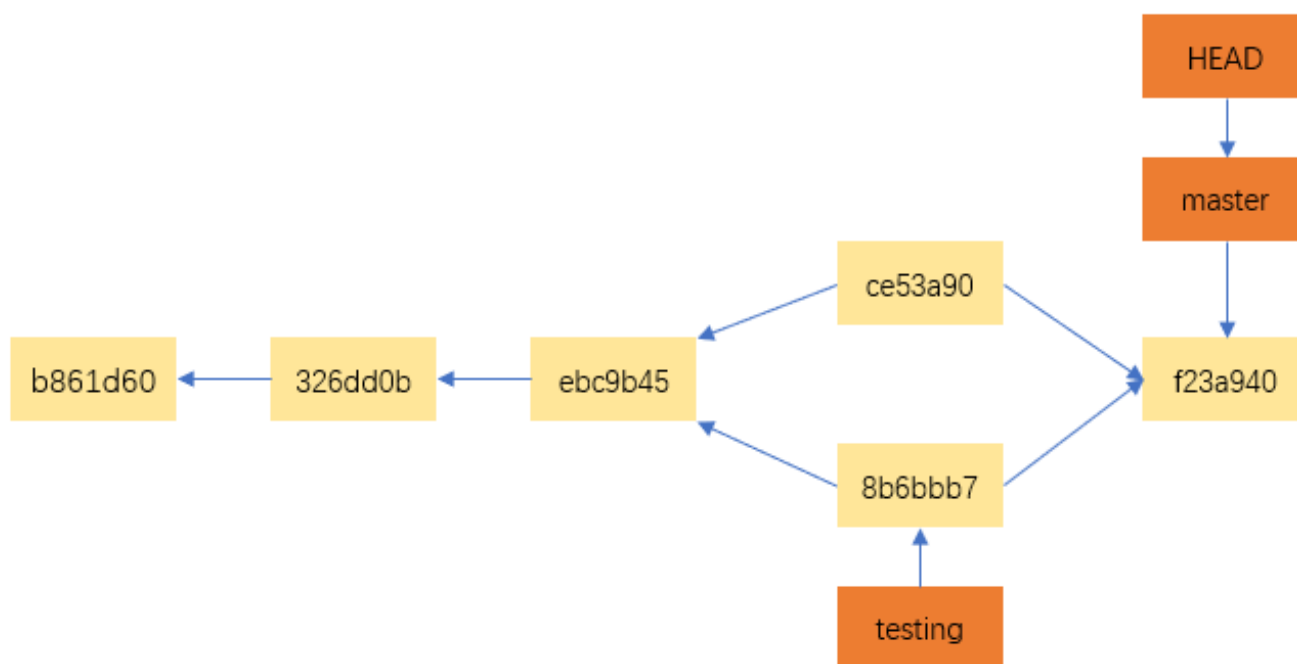
分支整合可以通过两种命令：一种是基于 `git merge` 命令，另一种是基于 `git rebase` 命令。

### 6.5.1 git merge——三方合并

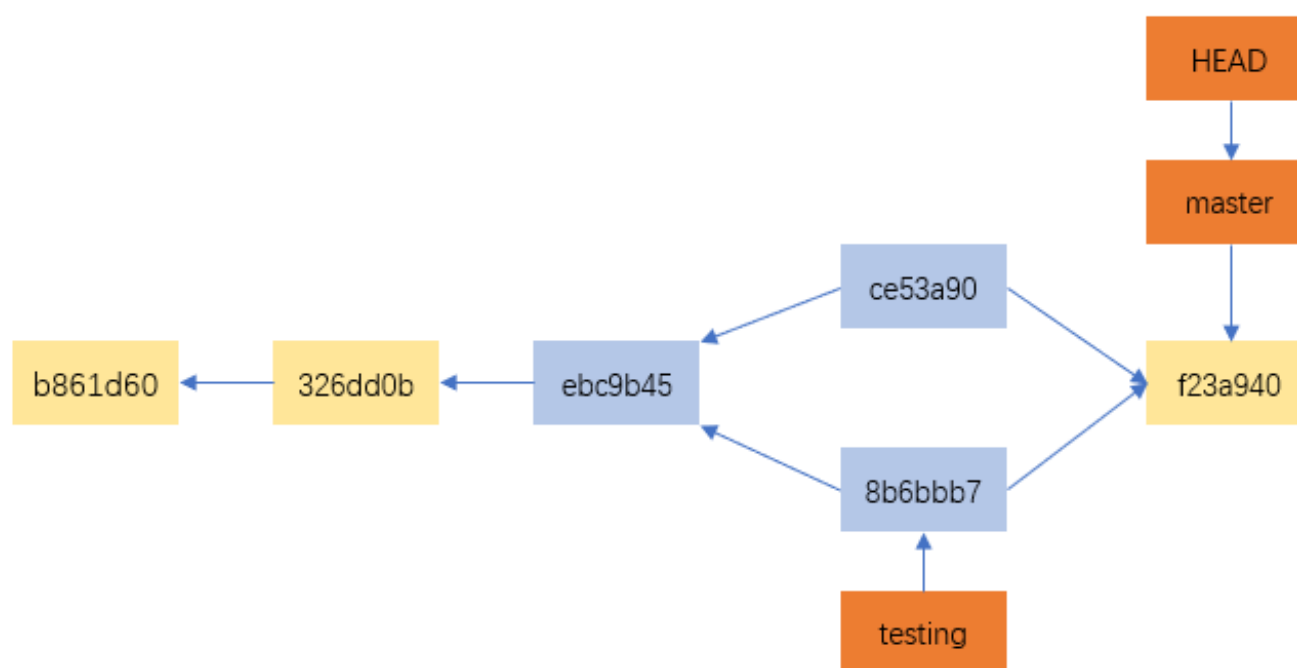
`git merge` 是一种保存分支结构的合并，并且是**三方合并**，通过实例来看吧。

```
1.  $ git merge testing #会跳出commit记录文件，默认退出即可
2.  Merge made by the 'recursive' strategy.
3.  README | 1 +
4.  1 file changed, 1 insertion(+)
5.
6.  $ git log --oneline --decorate --graph --all -6
7.  *    f23a940 (HEAD -> master) Merge branch 'testing'
8.  |\
9.  | * 8b6bbb7 (testing) add print(3) into README
10. * | ce53a90 add MIT LICENSE
11. | /
12. * ebc9b45 add print(2) into README
13. * 326dd0b add print(1) into README
14. * b861d60 branch note begin
```

上面可以很直观地看出提交历史，用图形表示如下：



既然叫三方合并，是那三方呢？见下图



上图中浅蓝色方块就是三方，分别是当前分支，要合并的分支，以及这两者的共同祖先（这个由git自己决定），merge合并会根据当前分支与祖先的差异和要合并的分支与祖先的差异进行共同合并。

## 6.5.2 git rebase——变基

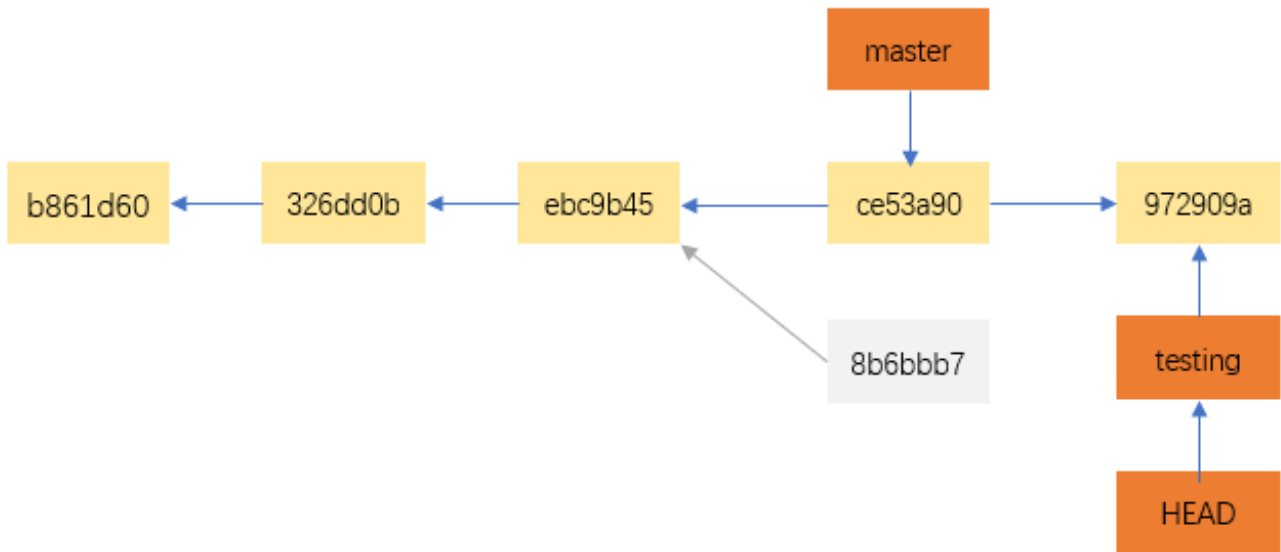
合并还有一种方法：那就是提取某一支（8b6bbb7）中引入的补丁和修改，然后在另一分支（ce53a90）的基础上应用一次。在Git中，这种操作就叫做**变基**。可以使用 rebase命令将



提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样。

```
1. $ git reset --hard ce53a90 #首先通过reset回溯到合并前的状态
2. HEAD is now at ce53a90 add MIT LICENSE
3.
4. $ git log --oneline --decorate --graph --all -5
5. * ce53a90 (HEAD -> master) add MIT LICENSE
6. | * 8b6bbb7 (testing) add print(3) into README
7. |/
8. * ebc9b45 add print(2) into README
9. * 326dd0b add print(1) into README
10. * b861d60 branch note begin
11.
12. $ git checkout testing #切换到要进行合并的分支
13.
14. $ git rebase master #使用rebase命令将testing合并到master
15. First, rewinding head to replay your work on top of it...
16. Applying: add print(3) into README
17.
18. $ git log --oneline --decorate --graph --all -5
19. * 972909a (HEAD -> testing) add print(3) into README
20. * ce53a90 (master) add MIT LICENSE
21. * ebc9b45 add print(2) into README
22. * 326dd0b add print(1) into README
23. * b861d60 branch note begin
```

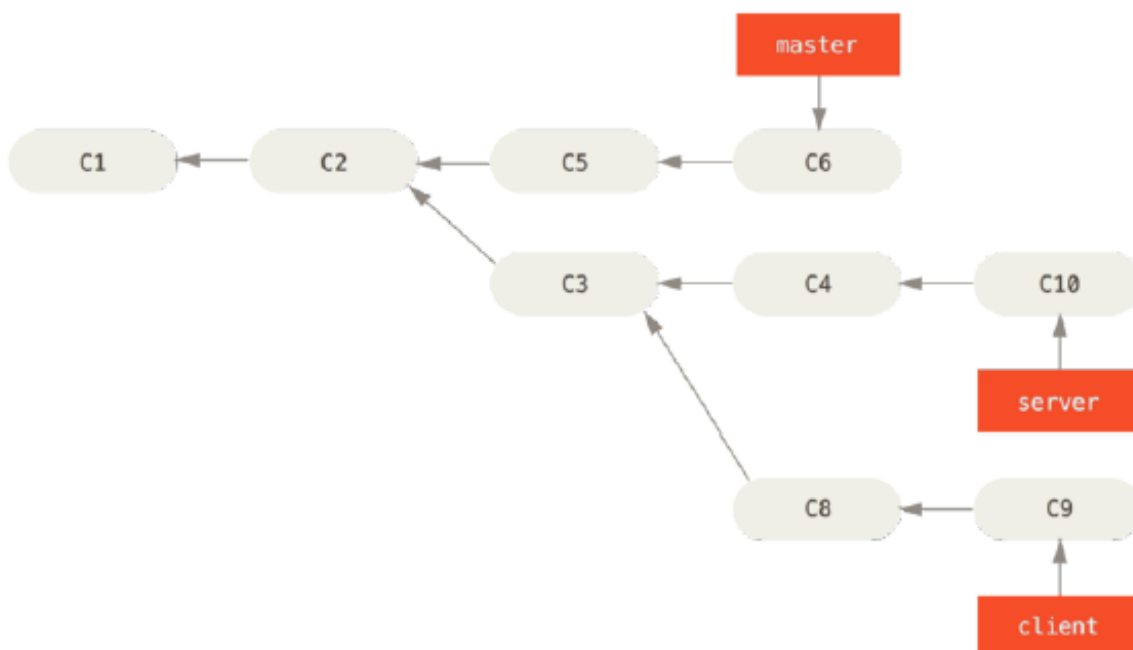
通过 `git log` 命令，整个历史可以看到没有想merge那样的分岔路，而是一条笔直的提交。rebase的原理是首先找到两个分支（即当前分支 testing、变基操作的目标基底分支master）的最近共同祖先，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存储为临时文件，然后将当前分支指向目标基底master,最后以此将之前另存为临时文件的修改依序应用。之前分支出去的提交就没有了延续，不会出现在提交历史中了。可以用新的图来表示这个过程。



无论是通过三方merge合并，还是通过rebase变基，最后的结果是一样的，唯一不同的是提交历史的区别，merge还会保存分支的历史，而rebase则不会，它的提交历史是没有分叉的直线，相对整洁。

#### 6.5.2.1 多重变基

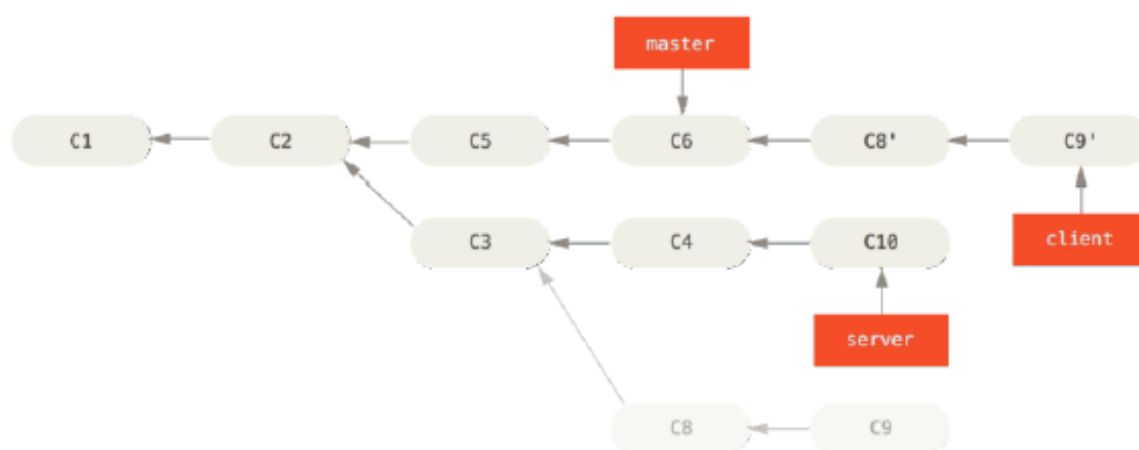
现在用commit id来简单表示校验和，你在主分支中的C2上创建了一个特性分支server，为服务端添加了一些功能，提交了C3和C4。然后从C3上创建了特性分支client，为客户端添加了一些功能，提交了C8和C9。最后，你回到server分支，又提交了C10。（ps:这里没有进行代码实践，有兴趣的朋友可以自己试试）



现在你希望将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：

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

以上命令的意思是：“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”。效果如下：



然后进行快速合并，

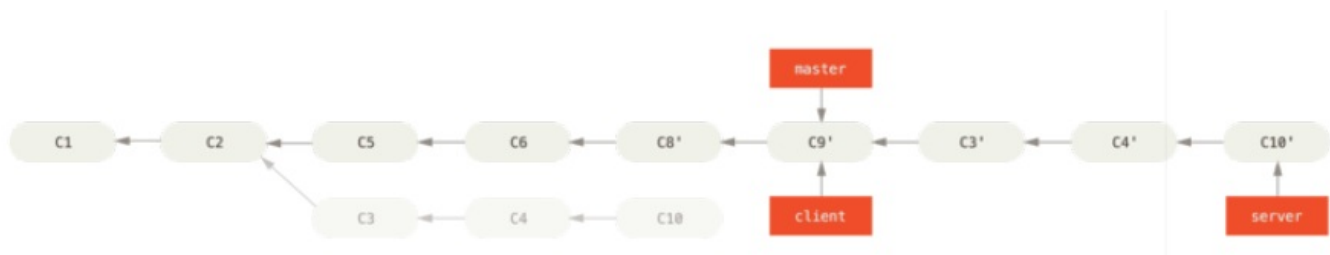
```
1. $ git checkout master
2. $ git merge client
```

接下来你决定将server分支中的修改也整合进来。使

用 `git rebase [basebranch] [topicbranch]` 命令可以直接将特性分支（即本例中的 server）变基到目标分支（即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。

```
1. $ git rebase master server
```

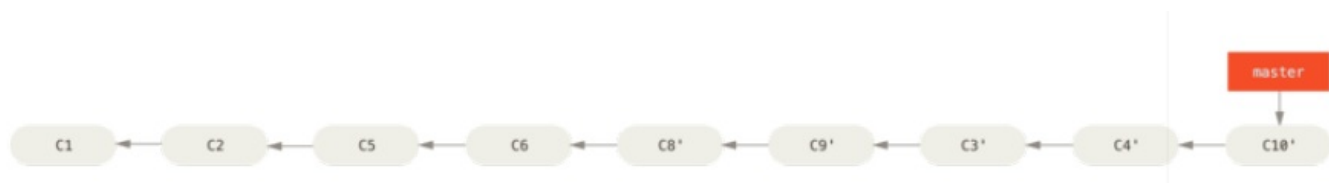
结果如下：



最后我们进行快速合并以及删除server，client分支。

1. `$ git checkout master`
2. `$ git merge server`
3. `$ git branch -d client`
4. `$ git branch -d client`

最终的提交历史：



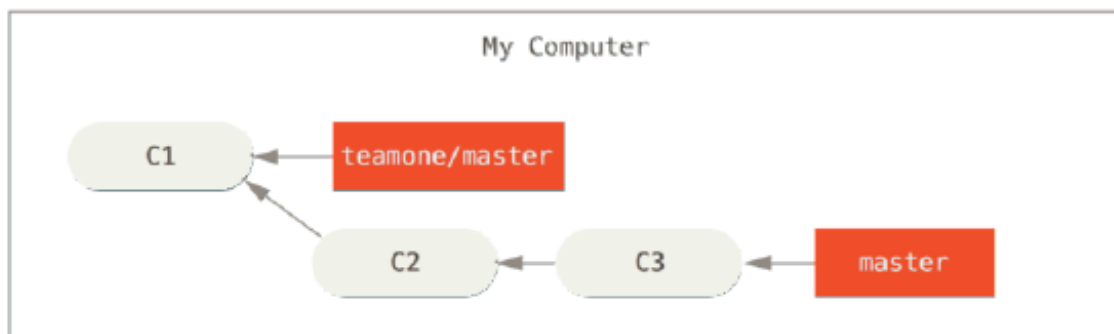
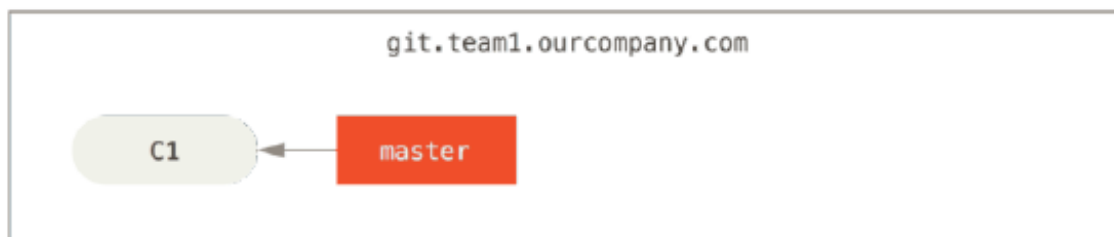
### 6.5.2.2 变基的风险

奇妙的变基也并非完美无缺，要用它得遵守一条准则：**不要对在你的仓库外有副本的分支执行变基**。否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你

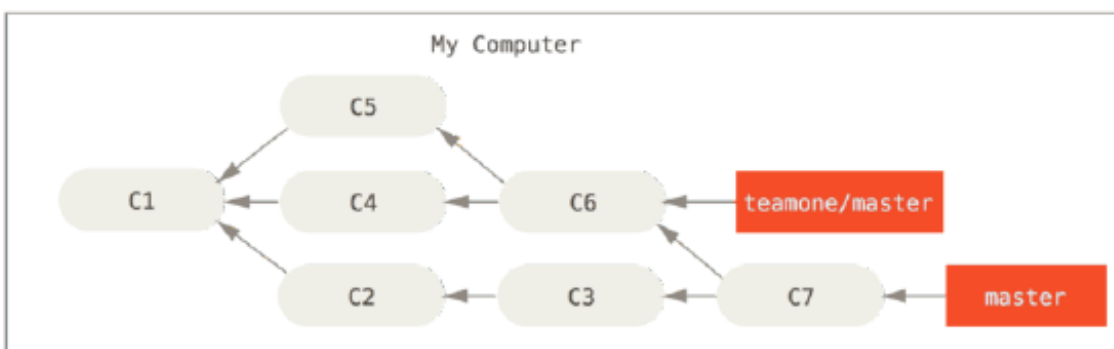
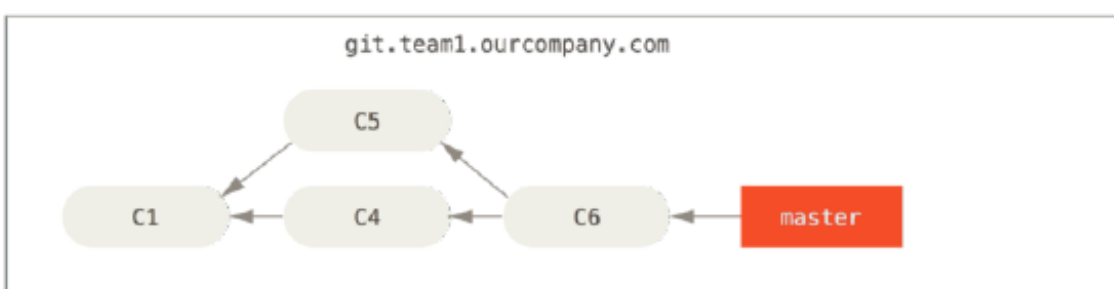
先简单说说这个意思，试想，A团队在本地进行了一次三方合并，然后push到远程服务器，B团队发现仓库有更改，通过 `git pull` 将新的提交拉到本地进行合并。可不久，A团队想对发到远程服务器的版本做做变基，把上次的三方合并修改成了变基，再次push到远程服务器。B团队发送远程服务器版本又更新，而且自己上一次pull下来的一些提交不见了（rebase丢弃掉了），只好再次进行合并，但发现没有，A团队两次推送没有更改最后的提交内容，也就是说B团队合并了两次相同的提交（历史混乱，B团队尴尬），不仅如此，A团队是想清理掉一些提交历史的，但B团队还保留那些历史，等B团队push到远程服务器时，A团队看到自己rebase去掉的历史又出现了（A团队尴尬）。

这样说不太容易理解，下面通过图形进行描述。

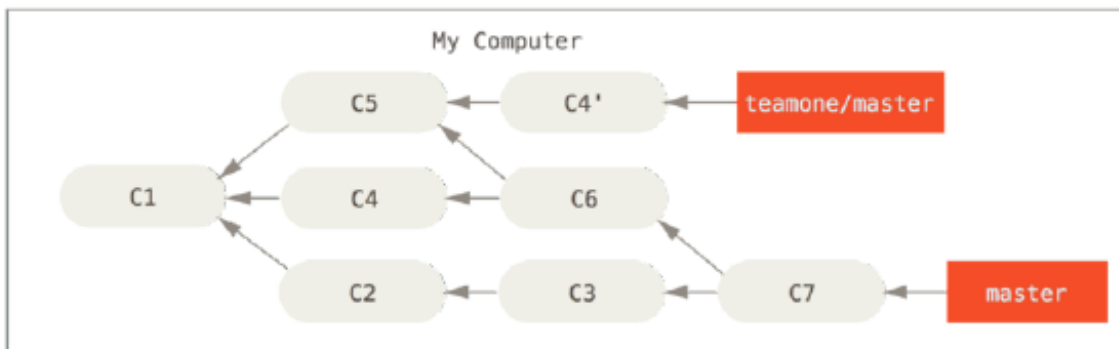
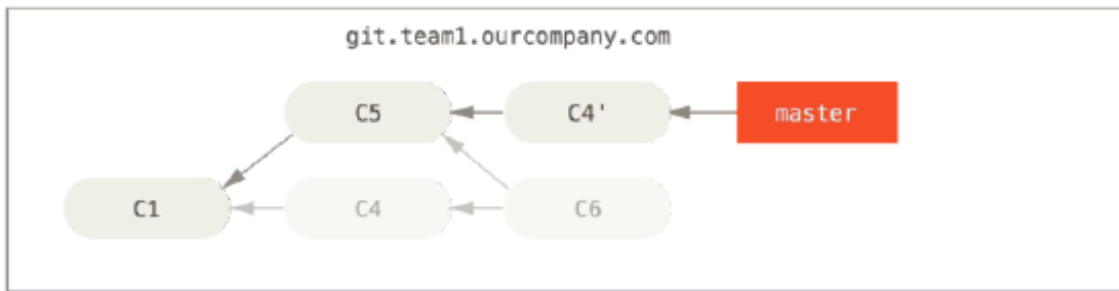
- 克隆一个仓库，然后在它的基础上进行了一些开发



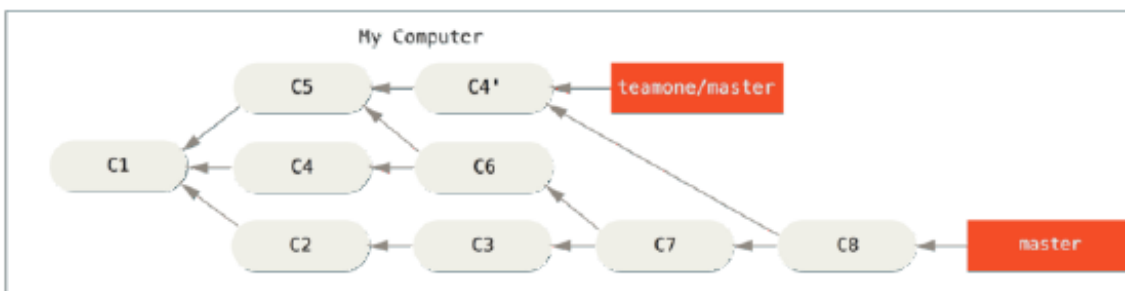
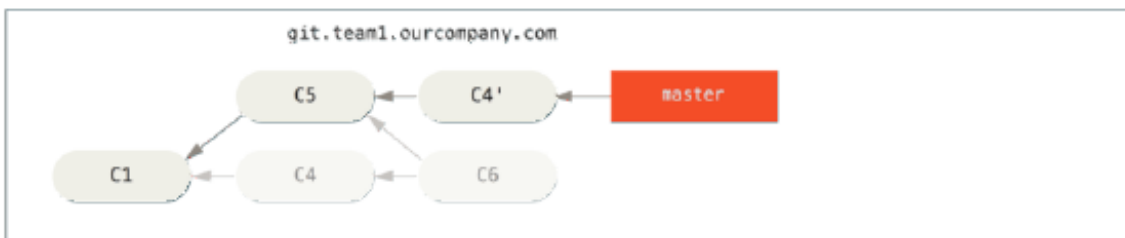
- 别人提交了一次合并，你抓取别人的提交，合并到自己的开发分支



- 有人推送了经过变基的提交，并丢弃了你的本地开发所基于的一些提交



- 你将相同的内容 (C6, C4')合并了两次



此时如果你执行 `git log` 命令，你会发现有两个提交的作者、日期、日志居然是一样的，这会令人感到混乱。此外，如果你将这一堆又推送到服务器上，你实际上是将那些已经被变基抛弃的提交又找了回来，这会令人感到更加混乱。很明显对方并不想在提交历史中看到C4和C6，因为之前就是他把这两个提交通过变基丢弃的。

**解决办法：**用变基解决变基，执行 `git rebase teamone/master`，Git将会

- 检查哪些提交是我们的分支上独有的 ( C2 , C3 , C4 , C6 , C7 )
- 检查其中哪些提交不是合并操作的结果 ( C2 , C3 , C4 )
- 检查哪些提交在对方覆盖更新时并没有被纳入目标分支 ( 只有 C2 和 C3 , 因为 C4 其实就是 C4' )
- 把查到的这些提交应用在 teamone/master 上面

当然这个办法有一个前提，那就是C4'和C4要几乎一样，否则变基无法识别。还有一个缓解疼痛的方法，同 `git pull --rebase` 替换 `git pull`，这个方法不会产生新的提交，也是变基。当然，最好的办法还是那条准则：**不要对在你的仓库外有副本的分支执行变基！**

## 6.6 合并冲突

当然在合并的过程中，可能会出现合并冲突的。合并冲突时，`git merge` 命令会显示是在哪个文件产生的冲突，我们来通过例子来试一试。

```

1.  $ git checkout master #接着上面的git rebase
2.  Switched to branch 'master'
3.  Your branch is ahead of 'origin/master' by 4 commits.
4.    (use "git push" to publish your local commits)
5.
6.  $ git merge testing
7.  Updating ce53a90..972909a
8.  Fast-forward
9.   README | 1 +
10.   1 file changed, 1 insertion(+)
11.
12. $ git branch -d testing
13. Deleted branch testing (was 972909a).
14.
15. $ git checkout -b newtesting
16. Switched to a new branch 'newtesting'
17.
18. $ echo "print("newtesting")" >> README
19.
20. $ git commit -am "Newtesting commit"
21. warning: LF will be replaced by CRLF in README.
22. The file will have its original line endings in your working directory
23. [newtesting 1aaf545] Newtesting commit

```

```

24. 1 file changed, 1 insertion(+)
25.
26. $ git checkout master
27. Switched to branch 'master'
28. Your branch is ahead of 'origin/master' by 5 commits.
29. (use "git push" to publish your local commits)
30.
31. $ echo "print("master")" >> README
32.
33. $ git commit -am "master commit"
34. warning: LF will be replaced by CRLF in README.
35. The file will have its original line endings in your working directory
36. [master 3883017] master commit
37. 1 file changed, 1 insertion(+)
38.
39. $ git merge newtesting
40. Auto-merging README
41. CONFLICT (content): Merge conflict in README
42. Automatic merge failed; fix conflicts and then commit the result.

```

可以看到最后出现了合并冲突，冲突出现在README文件中，我们需要打开文件看看冲突情况：

```

1. $ vi README
2. print(1)
3. print(2)
4. print(3)
5. <<<<<< HEAD
6. print(master)
7. =====
8. print(newtesting)
9. >>>>>> newtesting

```

为了解决冲突，你必须选择使用由=====分割的两部分中的一个，或者你也可以自行合并这些内容。我们选择print ( master )，从<<<<<<>>>>>>newtesting，除了print ( master ) 那句话，其他的都删除。

```

1. $ git add README
2.
3. $ git status
4. On branch master

```



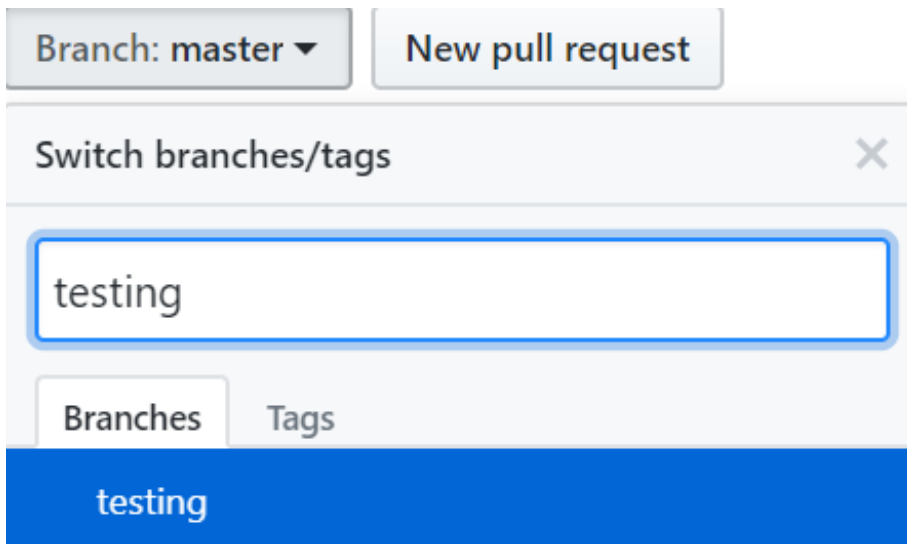
```
5. Your branch is ahead of 'origin/master' by 6 commits.
6. (use "git push" to publish your local commits)
7.
8. All conflicts fixed but you are still merging.
9. (use "git commit" to conclude merge)
10.
11. $ git commit -am "solve conflicts"
12. [master dc6b322] solve conflicts
13.
14. $ git log --oneline --decorate --graph --all -8
15. *   dc6b322 (HEAD -> master) solve conflicts
16. |\
17. | * 1aaf545 (newtesting) Newtesting commit
18. * | 3883017 master commit
19. | /
20. * 972909a add print(3) into README
21. * ce53a90 add MIT LICENSE
22. * ebc9b45 add print(2) into README
23. * 326dd0b add print(1) into README
24. * b861d60 branch note begin
```

## 6.7 远程分支

还记得我们之前通过 `git push -u origin master` 命令推送我们地仓库吗？最后推送的地址是 `origin/master`，这个叫做远程分支。其实在本地有一个 `origin/master` 的指针，这个叫做远程跟踪分支，用来跟踪远程分支（最后一次沟通）的状态，这个指针所指向的位置不会随着本地操作而发生改变，而当使用 `git fetch`、`git pull` 等命令会随着远程仓库的状态而改变。而本地的master指针是会默认追踪 `origin/master`，这个追踪是从 `git clone` 或者 `git remote add` 那一刻起。

当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。本地的分支并不会自动与远程仓库同步-你必须显式地推送想要分享的分支。这样，你就可以把不愿意分享的内容放到私人分支上，而将需要和别人协作的内容推送到公开分支。下面通过实例进行学习。

首先我们点开在GitHub创建的仓库——playground，然后如图所示，创建一个新分支 `testing`，由于我已经创建，所以已经显示有testing分支。



我们在本地中使用 `git fetch` 命令，将刚刚创建的分支下载到本地。

```
1. $ git fetch
2. From github.com:FangYang970206/playground
3. * [new branch]      testing    -> origin/testing
```

可以看到多出新的分支testing(本地分支)跟踪origin/master(远程跟踪分支)。我们通过 `git checkout <branch_name>` 看到分支是否处于跟踪的状态。

```
1. $ git checkout master
2. Switched to branch 'master'
3. Your branch is up to date with 'origin/master'.
4.
5. $ git checkout testing
6. Switched to a new branch 'testing'
7. Branch 'testing' set up to track remote branch 'testing' from 'origin'
   .
```

现在分别对testing和master分支做一次提交并push

```
1. $ echo "print(\"testing\")" >> README
2.
3. $ git commit -am README
4. warning: LF will be replaced by CRLF in README.
5. The file will have its original line endings in your working directory
   .
6. [testing f2466c5] README
7. 1 file changed, 1 insertion(+)
```

```
8.
9. $ git push origin testing
10. Enumerating objects: 5, done.
11. Counting objects: 100% (5/5), done.
12. Delta compression using up to 8 threads.
13. Compressing objects: 100% (3/3), done.
14. Writing objects: 100% (3/3), 367 bytes | 183.00 KiB/s, done.
15. Total 3 (delta 0), reused 0 (delta 0)
16. To github.com:FangYang970206/playground.git
17.    dc6b322..f2466c5  testing -> testing
18.
19. $ git checkout master
20. Switched to branch 'master'
21. Your branch is up to date with 'origin/master'.
22.
23. $ echo "print("master1")" >> README
24.
25. $ git commit -am README
26. warning: LF will be replaced by CRLF in README.
27. The file will have its original line endings in your working directory
28. [master b849b25] README
29.  1 file changed, 1 insertion(+)
30.
31. $ git push origin master
32. Enumerating objects: 5, done.
33. Counting objects: 100% (5/5), done.
34. Delta compression using up to 8 threads.
35. Compressing objects: 100% (3/3), done.
36. Writing objects: 100% (3/3), 360 bytes | 360.00 KiB/s, done.
37. Total 3 (delta 0), reused 0 (delta 0)
38. To github.com:FangYang970206/playground.git
39.    dc6b322..b849b25  master -> master
```

还可以更改其他本地指针来跟踪远程跟踪分支，下面通过实例来学习

```
1. $ git checkout -b foo origin/master
2. Switched to a new branch 'foo'
3. Branch 'foo' set up to track remote branch 'master' from 'origin'.
4.
5. $ echo "# if" >> test.rb
6.
7. $ git commit -am "commit test.rb"
8. warning: LF will be replaced by CRLF in test.rb.
```

```
9. The file will have its original line endings in your working directory
.
10. [foo d3f4109] commit test.rb
11. 1 file changed, 1 insertion(+)
12.
13. $ git push origin HEAD:master
14. Enumerating objects: 5, done.
15. Counting objects: 100% (5/5), done.
16. Delta compression using up to 8 threads.
17. Compressing objects: 100% (2/2), done.
18. Writing objects: 100% (3/3), 260 bytes | 86.00 KiB/s, done.
19. Total 3 (delta 1), reused 0 (delta 0)
20. remote: Resolving deltas: 100% (1/1), completed with 1 local object.
21. To github.com:FangYang970206/playground.git
22. b849b25..d3f4109 HEAD -> master
23.
24. $ git checkout testing
25. Switched to branch 'testing'
26. Your branch is up to date with 'origin/testing'.
27.
28. $ git checkout -b testing1 origin/testing
29. Switched to a new branch 'testing1'
30. Branch 'testing1' set up to track remote branch 'testing' from
    'origin'.
31.
32. $ echo "print(\"testing1\")" >> README
33.
34. $ git commit -am "print testing1"
35. warning: LF will be replaced by CRLF in README.
36. The file will have its original line endings in your working directory
.
37. [testing1 2600339] print testing1
38. 1 file changed, 1 insertion(+)
39.
40. $ git push origin HEAD:testing
41. Enumerating objects: 5, done.
42. Counting objects: 100% (5/5), done.
43. Delta compression using up to 8 threads.
44. Compressing objects: 100% (3/3), done.
45. Writing objects: 100% (3/3), 373 bytes | 124.00 KiB/s, done.
46. Total 3 (delta 0), reused 0 (delta 0)
47. To github.com:FangYang970206/playground.git
48. f2466c5..2600339 HEAD -> testing
49.
50. $ git log --oneline --decorate --graph --all -9
```

```

51. * d3f4109 (origin/master, origin/HEAD, foo) commit test.rb
52. * b849b25 (master) README
53. | * 2600339 (HEAD -> testing1, origin/testing) print testing1
54. | * f2466c5 (testing) README
55. | /
56. * dc6b322 solve conflicts
57. | \
58. | * 1aaf545 Newtesting commit
59. * | 3883017 master commit
60. | /
61. * 972909a (newtesting) add print(3) into README
62. * ce53a90 add MIT LICENSE

```

可以看到之前跟踪远程跟踪分支的master和testing指针是没有移动，新建的foo和testing1取代了它们。这样要注意一点，使用不同于远程分支名的分支进行push，一定要指定当前分支，不然默认还是通过branch\_name推送到origin/branch\_name，出现Everything up-to-date，指定可以通过以上的HEAD ( source ) : testing ( target )

这里的取代方式可以有两种：一种是直接新建分支取代，也就是上面

的 `git checkout -b testing1(newbranch) origin/testing(Remote tracking branch)`，

还有一种是通过现在分支进行取代，可以通

过 `git branch -u <Remote tracking branch> <existed branch>` 命令，注意一点，不要

出现远程跟踪分支和跟踪的指针出现分叉，这样会导

致 `Your branch and 'origin/xxx' have diverged` 这个问题，详细可参考这个[链接](#)。

## 6.8 分支练习（强烈推荐）

强烈建议去<https://learngitbranching.js.org/>，这个网站有很多分支的练习，还有一部分是远程控制的练习，配合动画，非常适合正在学习git的朋友，相信可以让你的git本领上一个台阶。

## 7. pull request的使用

如果你发现你感兴趣的仓库的bug或者你想添加某个新功能到你感兴趣的仓库中（一般是先在Issue中提出想法或问题，沟通好后可以在对应编号上创建pull re，这时候你就可以pull request这个沟通利器了（一般是先在Issue中提出想法或问题，沟通好后可以在对应编号上创建pull request）。

这里介绍一下一个仓库，可供你进行pull request练习，仓库地址是<https://github.com/Data4Democracy/github-playground>，当然，也欢迎你对本文的仓库<https://github.com/FangYang970206/playground>进行pull request，由于对自己的仓库进行pull request，本质上就是本地合并再提交，所以没有必要，完全可以在本地进行，pull request是以协作开发为目的，为了方便，对<https://github.com/Data4Democracy/github-playground>进行pull request，看本篇文章的朋友就可以随意选择两者之一。

### pull request流程：

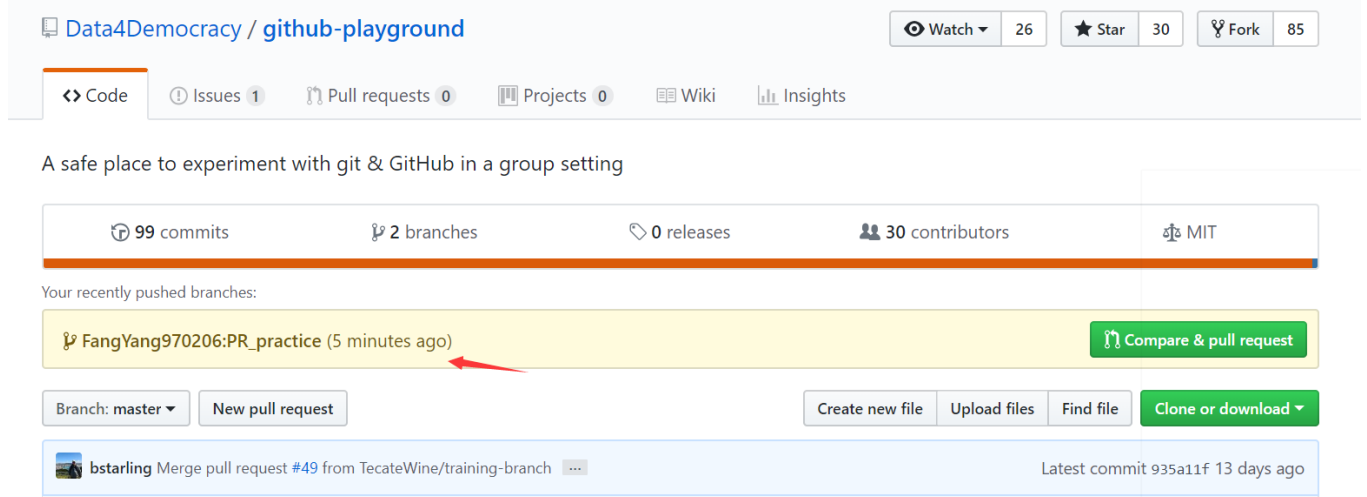
- 先fork你感兴趣的仓库到自己的仓库中（副本）
- 将副本仓库克隆到本地
- 从master分支中创建一个新分支
- 在分支中进行修改，以此改进项目
- 将分支推送到github仓库
- 创建一个pull request
- 讨论，根据实际情况继续修改
- 项目的拥有者合并或关闭你的合并请求

打开<https://github.com/Data4Democracy/github-playground>，点击Fork，等待副本仓库生成，复制下载url，使用 `git clone` 到本地，然后在master分支下创建PR\_practice分支，然后使用vi hello\_test.py对文件进行修改，在后面添加一句print("PR practice, thanks")，然后Ese:wq保存后，再进行提交修改，最后push到origin/PR\_practice分支就可以了，详细过程如下：

```
1.  $ git clone git@github.com:FangYang970206/github-playground.git
2.  Cloning into 'github-playground'...
3.  remote: Counting objects: 230, done.
4.  remote: Compressing objects: 100% (7/7), done.
5.  remote: Total 230 (delta 2), reused 4 (delta 1), pack-reused 221
6.  Receiving objects: 100% (230/230), 307.80 KiB | 324.00 KiB/s, done.
7.  Resolving deltas: 100% (85/85), done.
8.
9.  $ cd github-playground/
10.
11. $ git checkout -b PR_practice
```

```
12. Switched to a new branch 'PR_practice'
13.
14. $ vi hello_test.py
15.
16. $ git add hello_test.py
17.
18. $ git commit -m "add PR practice"
19. [PR_practice a874bdf] add PR practice
20. 1 file changed, 1 insertion(+)
21.
22. $ git push origin PR_practice
23. Enumerating objects: 5, done.
24. Counting objects: 100% (5/5), done.
25. Delta compression using up to 8 threads.
26. Compressing objects: 100% (2/2), done.
27. Writing objects: 100% (3/3), 309 bytes | 309.00 KiB/s, done.
28. Total 3 (delta 1), reused 0 (delta 0)
29. remote: Resolving deltas: 100% (1/1), completed with 1 local object.
30. To github.com:FangYang970206/github-playground.git
31. * [new branch] PR_practice -> PR_practice
```

我们再进入我们感兴趣的仓库地址，就会发现如下页面



Data4Democracy / github-playground

Watch 26 Star 30 Fork 85

Code Issues 1 Pull requests 0 Projects 0 Wiki Insights

A safe place to experiment with git & GitHub in a group setting

99 commits 2 branches 0 releases 30 contributors MIT

Your recently pushed branches:

FangYang970206:PR\_practice (5 minutes ago) Compare & pull request


Branch: master New pull request Create new file Upload files Find file Clone or download

bstarling Merge pull request #49 from TecateWine/training-branch Latest commit 935a11f 13 days ago

我们点击旁边的compare&pull request或者直接点击New pull request，写一下标题和commit，然后creat pull request。

# Open a pull request


Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



base fork: Data4Democracy/github-playg...  
base: master

head fork: FangYang970206/github-playg...  
compare: PR\_practice

✓ Able to merge. These branches can be automatically merged.



add PR practice

Write

Preview

AA B i “ <> 🔗 ☰ ☷ ☶ @ 📎 ↶

PR practice, thanks 😊

Attach files by dragging & dropping, [selecting them](#), or pasting from the clipboard.

☒ Allow edits from maintainers. [Learn more](#)

Create pull request

最后等待仓库拥有者审核，对这个pull request进行讨论，看是否要进行再修改等等。另外，每一个pull request都可以看files changed，可以看到有哪些行添加进去了，有哪些删除了，很是方便。

以上，就是一个pull request的流程，记得动手操作一遍。

## 8. 参考

最后，希望这篇文章能对看的朋友有所帮助，欢迎给这篇文章来个star。本文大量参考了[Pro Git](#)，建议读者可以去读一读这本git官网推荐的书籍。[git-github-intro](#)对git有一个不错大致简介。[learngitbranching](#)是一个非常不错的动手学习网站，推荐去动手学习，更多资源可以去参考[trygit](#)里面的内容。