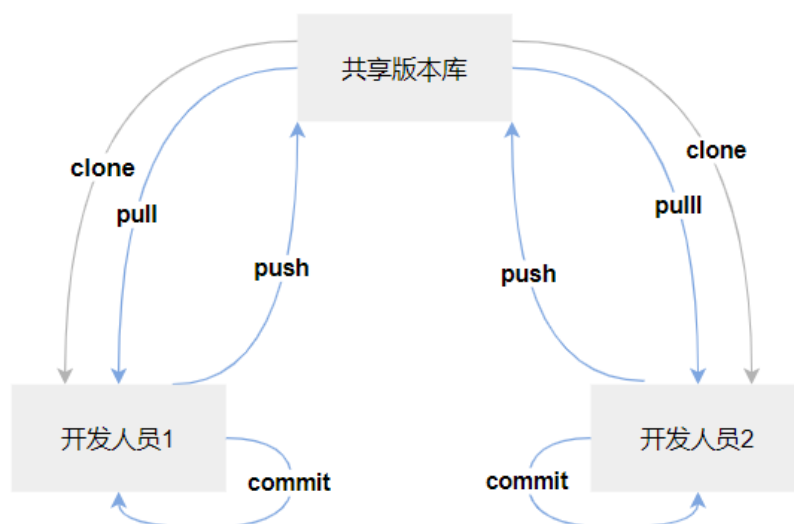


---

# Git 使用指北

## 一、Git 简介

Git 是分布式版本控制系统，那么它可以没有中央服务器的，每个人的电脑就是一个完整的版本库，这样，工作的时候就不需要联网了，因为版本都是在自己的电脑上。既然每个人的电脑都有一个完整的版本库，那多个人如何协作呢？比如说自己在电脑上改了文，其他人也在电脑上改了文，这时，你们两之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。下图就是分布式版本控制工具管理方式



---

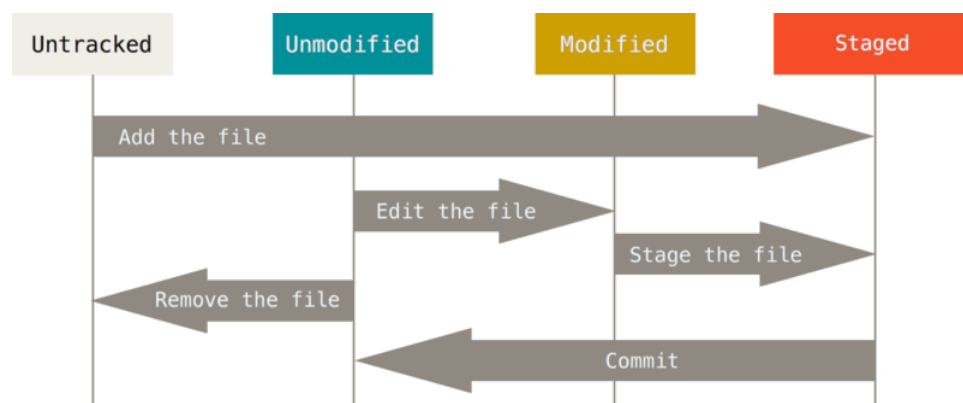
## 二、Git 基本概念

本小结涉及 Git 使用中，git 工作区域概念，git 关注文件的状态和 git 的一些常用命令，旨在对 git 有基本认知的了解。

### 2.1 四个工作区域

Git 本地有四个工作区域：工作目录（Working Directory）、暂存区（Stage/Index）、资源库（Repository 或 Git Directory）、git 仓库（Remote Directory）。文件在这四个区域之间的转换关系如下：

### 2.2 四个文件状态



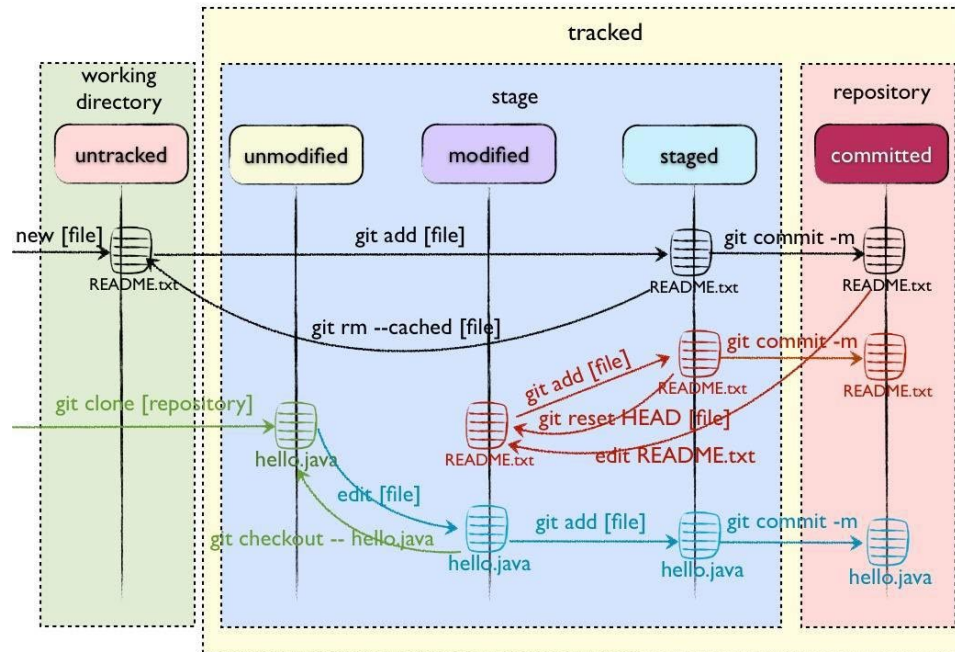
**Untracked:** 未跟踪，此文件在文件夹中，但并没有加入到 git 库，不参与版本控制。通过 `git add` 状态变为 Staged.

**Unmodify:** 文件已经入库，未修改，即版本库中的文件快照内容与文件夹中完全一致。这种类型的文件有两种去处，如果它被修改，而变为 Modified. 如果使用 `git rm` 移出版本库，则成为 Untracked 文件

**Modified:** 文件已修改，仅仅是修改，并没有进行其他的操作。这个文件也有两个去处，通过 `git add` 可进入暂存 staged 状态，使用 `git checkout` 则丢弃修改过，返回到 unmodify 状态，这个 `git checkout` 即从库中取出文件，覆盖当前修改

**Staged:** 暂存状态. 执行 `git commit` 则将修改同步到库中, 这时库中的文件和本地文件又变为一致, 文件为 Unmodify 状态. 执行 `git reset HEAD filename` 取消暂存, 文件状态为 Modified

## 2.3 四种文件状态转换



## 2.4 常用命令

### 2.4.1 新建代码库

```
# 在当前目录新建一个 Git 代码库
git init
# 新建一个目录, 将其初始化为 Git 代码库
git init [project-name]
# 下载一个项目和它的整个代码历史
git clone [url]
```

### 2.3.2 查看文件状态

```
#查看指定文件状态
git status [filename]
#查看所有文件状态
git status
```

### 2.3.3 工作区<->暂存区

```
# 添加指定文件到暂存区
git add [file1] [file2] ...
# 添加指定目录到暂存区, 包括子目录
```

---

```
git add [dir]
# 添加当前目录的所有文件到暂存区
git add .
# 当我们需要删除暂存区或分支上的文件，同时工作区也不需要这个文件了，可以使用（△）
git rm file_path
# 当我们需要删除暂存区或分支上的文件，但本地又需要使用，这个时候直接 push 那边这个文件就没有，如果 push 之前重新 add 那么还是会有。
git rm --cached file_path
# 直接加文件名 从暂存区将文件恢复到工作区，如果工作区已经有该文件，则会选择覆盖
# 加了【分支名】 +文件名 则表示从分支名为所写的分支中拉取文件 并覆盖工作区里的文件
git checkout
```

### 2.3.4 工作区<->资源库（版本库\本地）

```
# 将暂存区-->资源库（版本库）
git commit -m '该次提交说明'
# 如果出现:将不必要的文件 commit 或者 上次提交觉得是错的 或者不想改变暂存区内容，只是想调整提交的信息
# 移除不必要的添加到暂存区的文件
git reset HEAD 文件名
# 去掉上一次的提交（会直接变成 add 之前状态）
git reset HEAD^
# 去掉上一次的提交（变成 add 之后，commit 之前状态）
git reset --soft HEAD^
```

### 2.3.5 远程操作

```
# 取回远程仓库的变化，并与本地分支合并
git pull
# 上传本地指定分支到远程仓库
git push
```

### 2.3.6 其他命令

```
# 显示当前的 Git 配置
git config --list
# 编辑 Git 配置文件
git config -e [--global]
# 初次 commit 之前，需要配置用户邮箱及用户名，使用以下命令：
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
# 调出 Git 的帮助文档
git --help
# 查看某个具体命令的帮助文档
git +命令 --help
# 查看 git 的版本
git --version
# 查看追踪的远程分支
git remote -v
# 查看本地仓库的分支详细信息
git branch -a -vv
```

## 三、分支和 commit

### 3.1 分支 commit 概念图

#### Master

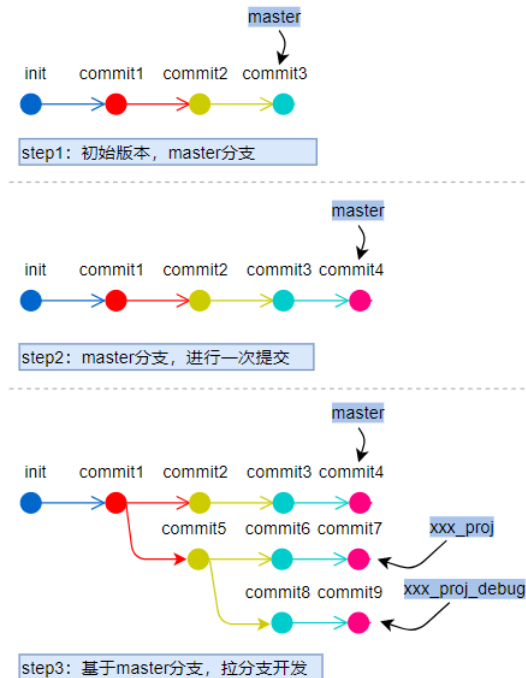
git分支:

- 1) 本质是指向提交对象的可变指针
- 2) 默认分支名是master
- 3) 每次提交时自动移动

#### Commit

一次提交、快照

- 1) 提交对象 (SHA-1哈希算法)
- 2) 指向修改的内容



### 3.2 提交

先提交三个文件:

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

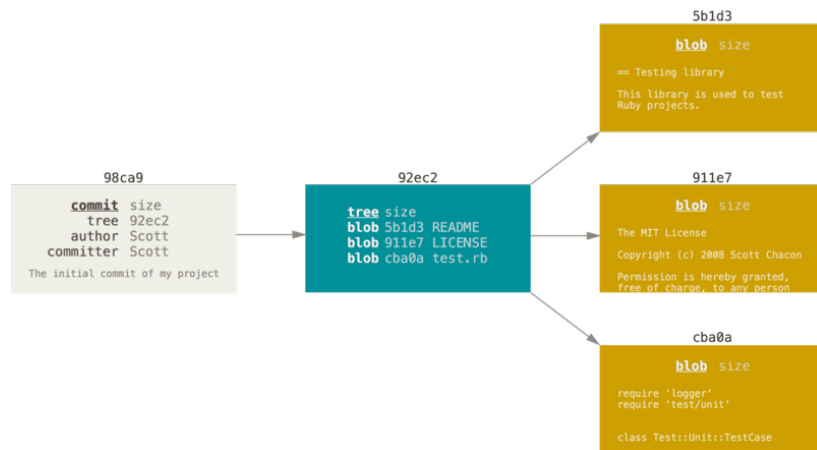
**commit、tree、blob 对象概念:**

Git 仓库中当前有五个对象

blob 对象: 保存着文件快照

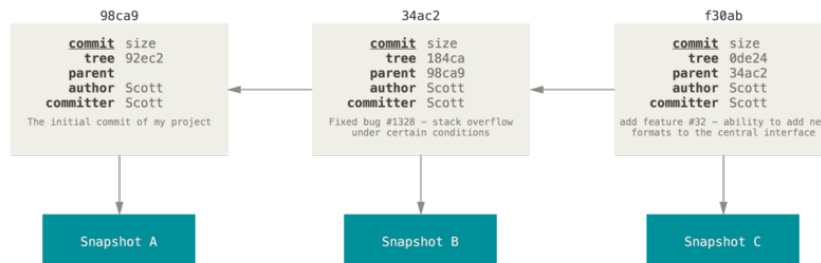
tree 对象: 记录着目录结构和 blob 对象索引

commit 对象: 指向着 tree 对象和所有提交信息



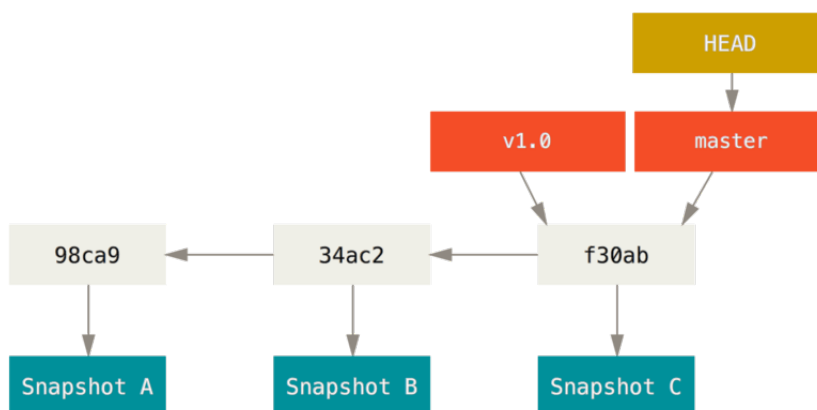
## 提交对象和树结构

做些修改后再次提交，那么这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。



## 提交对象及其父对象

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。master 分支会在每次提交时自动向前移动。



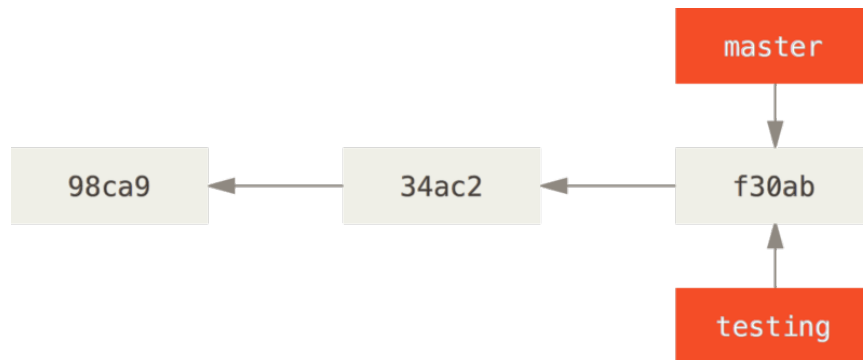
---

## 四、 分支操作

### 4.1 分支创建 `git branch`

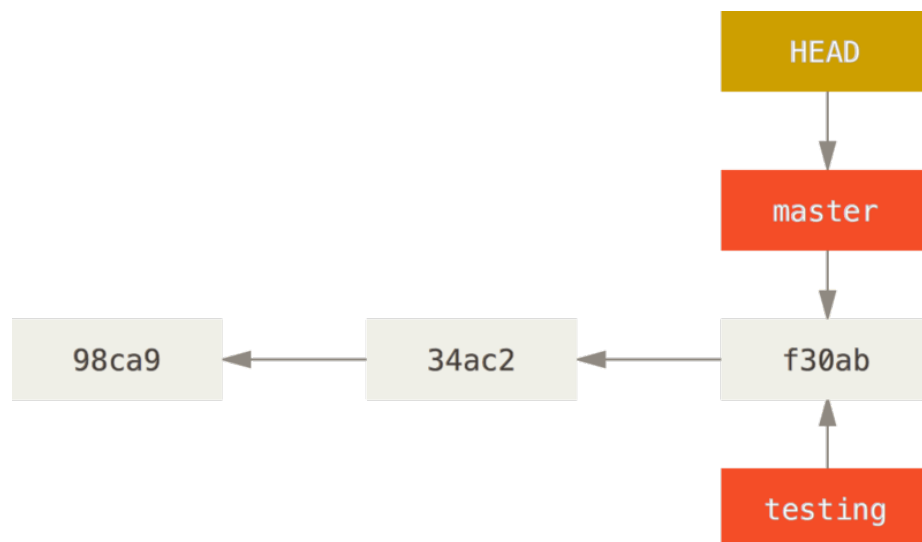
```
$ git branch testing
```

这会在当前所在的提交对象上创建一个指针。



### 4.2 HEAD 指针

Git 怎么知道当前在那个分支上呢？引出了 HEAD 的特殊指针，它是一个指针，指向当前所在的本地分支。在本例中，你仍然在 master 分支上。因为 `git branch` 命令仅仅 创建 一个新分支，并不会自动切换到新分支中去。

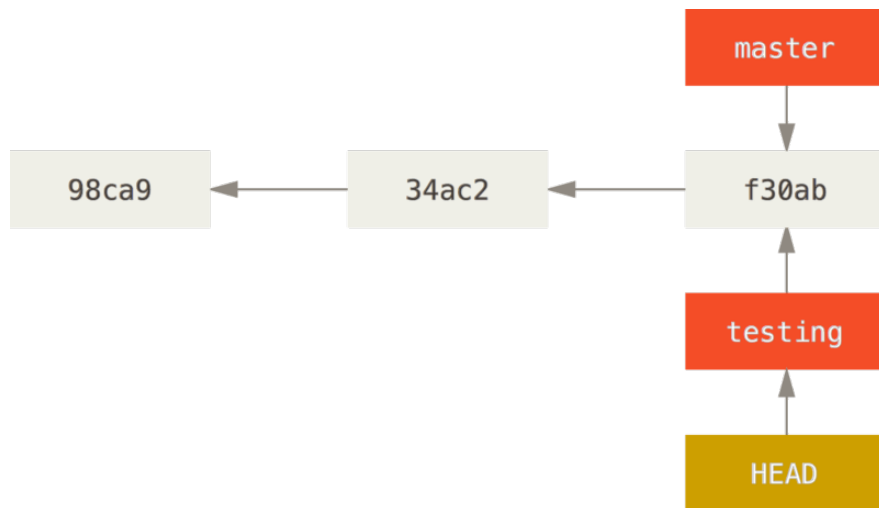


### 4.3 切换分支 `git checkout`

# 要切换到一个已存在的分支，你需要使用 `git checkout` 命令。我们现在切换到新创建的 `testing` 分支去：

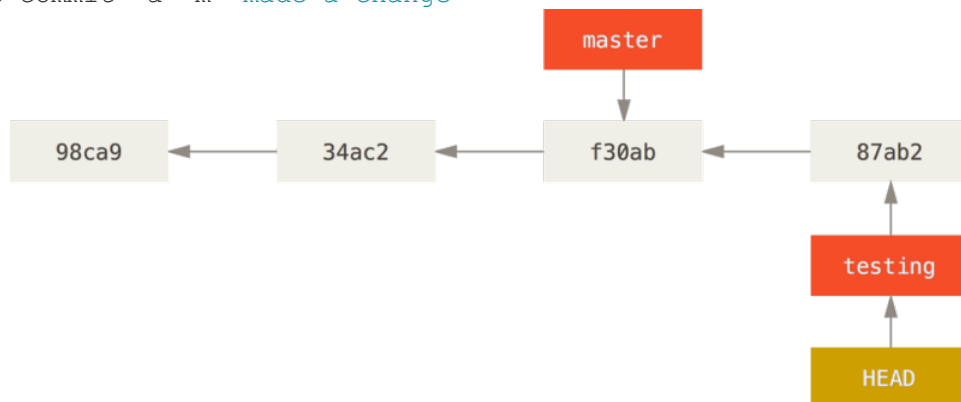
```
$ git checkout testing
```

# 这样 HEAD 就指向 `testing` 分支了。



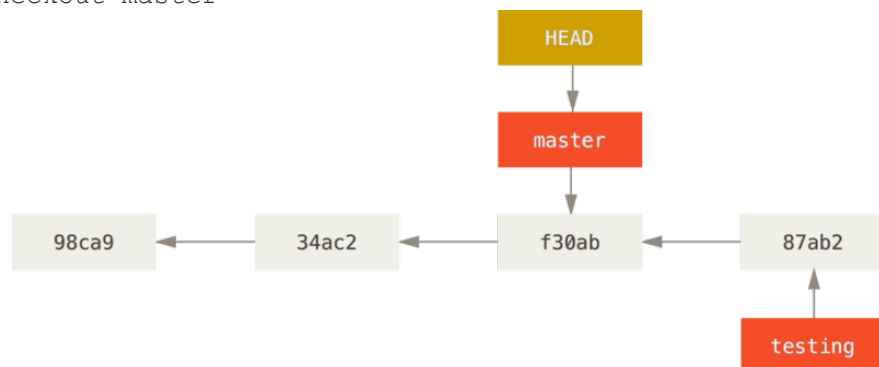
#### 4.4 HEAD 指向当前所在分支

```
# 现在不妨再提交一次:  
$ vim test.rb  
$ git add test.rb  
$ git commit -a -m 'made a change'
```



#### 4.4 HEAD 指向当前所在分支

```
# testing 分支向前移动了, 但是 master 分支却没有, 它仍然指向运行 git  
checkout 时所指的对象。这就有意思了, 现在我们切换回 master 分支看看  
$ git checkout master
```



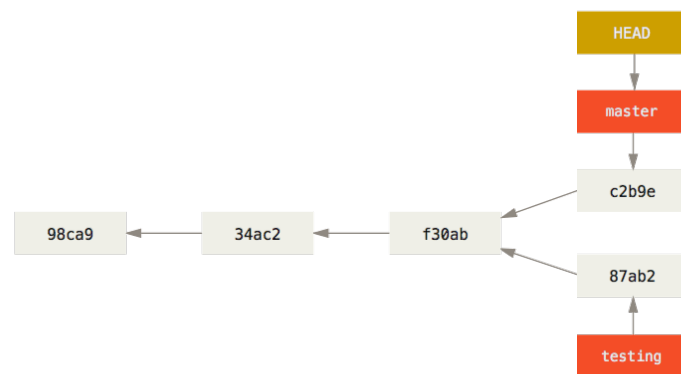


---

## 4.5 项目分叉

```
# 我们不妨再稍微做些修改并提交：  
$ vim test.rb  
$ git add test.rb  
$ git commit -a -m 'made other changes'
```

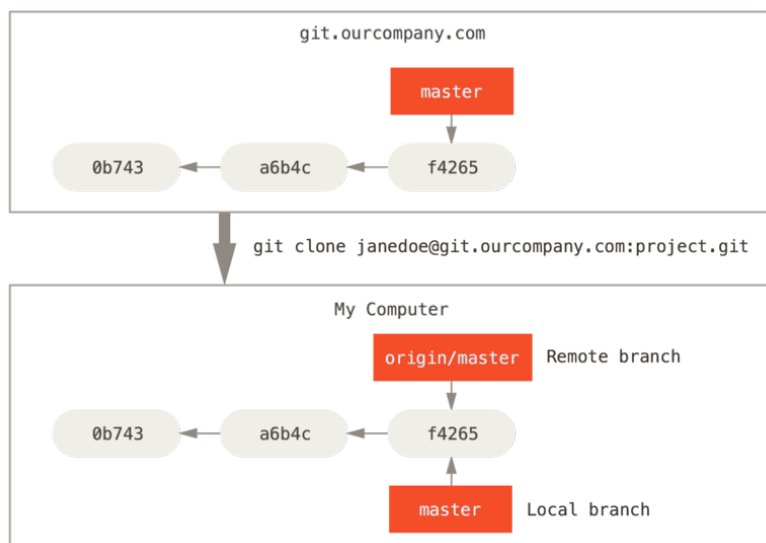
现在，这个项目的提交历史已经产生了分叉。因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 `master` 分支进行了另外一些工作。上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，并在时机成熟时将它们合并起来。而所有这些工作，你需要的命令只有 `branch`、`checkout` 和 `commit`。



## 五、远程分支

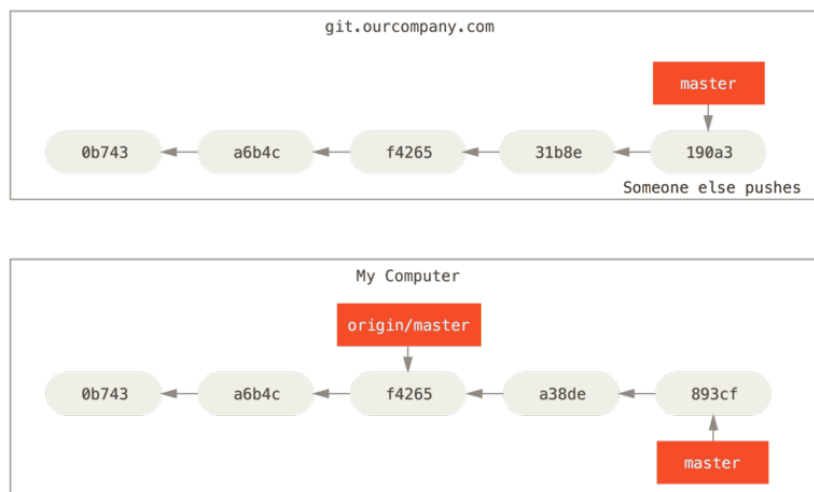
远程引用是对远程仓库的引用（指针），包括分支、标签等等。`git ls-remote` 来显式地获得远程引用的完整列表，或者通过 `git remote show` 获得远程分支的更多信息。

## 5.1 git clone 动作



## 5.2 克隆之后的服务器与本地仓库

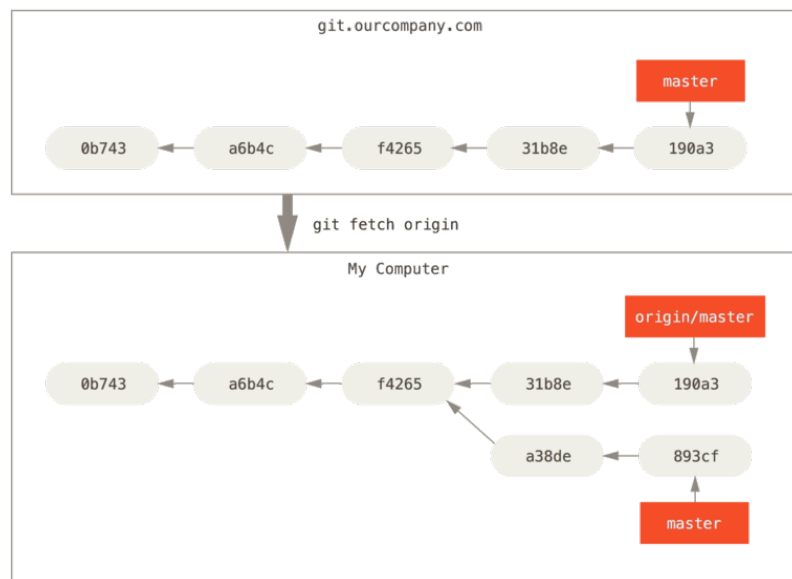
如果你在本地的 `master` 分支做了一些工作，在同一段时间内有其他人推送提交到 `git.ourcompany.com` 并且更新了它的 `master` 分支，这就是说你们的提交历史已走向不同的方向。即便这样，只要你保持不与 `origin` 服务器连接（并拉取数据），你的 `origin/master` 指针就不会移动。



## 5.3 本地与远程的工作可以分叉

如果要与给定的远程仓库同步数据，运行 `git fetch` 命令（在本例中为 `git fetch origin`）。这个命令查找“`origin`”是哪一个服务器（在本例中，它是

git.ourcompany.com)，从中抓取本地没有的数据，并且更新本地数据库，移动 origin/master 指针到更新之后的位置。



## 5.4 向远程分支推送

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。如果希望和别人一起在名为 serverfix 的分支上工作，你可以像推送第一个分支那样推送它。运行 git push：

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
* [new branch] serverfix -> serverfix
```

## 5.5 同步远程分支 git fetch

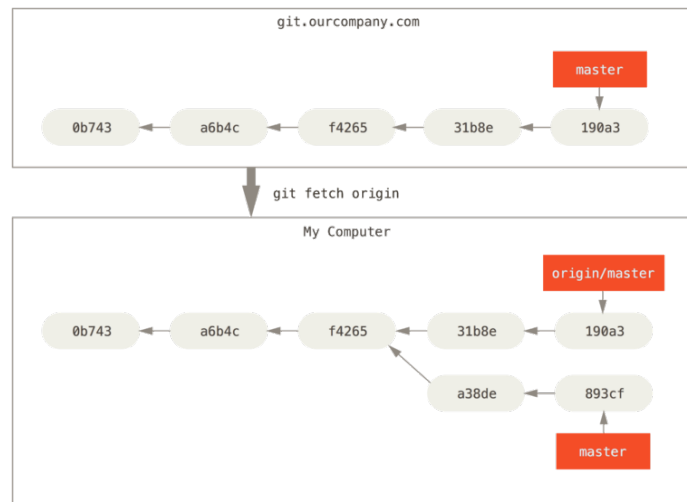
```
$ git fetch origin
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
```

---

```
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
* [new branch]      serverfix    -> origin/serverfix
```

这里特别说明 **FETCH\_HEAD** 指针，`git fetch` 命令同步远程分支之后，会在 `git` 中生成一个 **FETCH\_HEAD** 指针，该指针指向当前分支追踪的同步的远程分支最新节点。

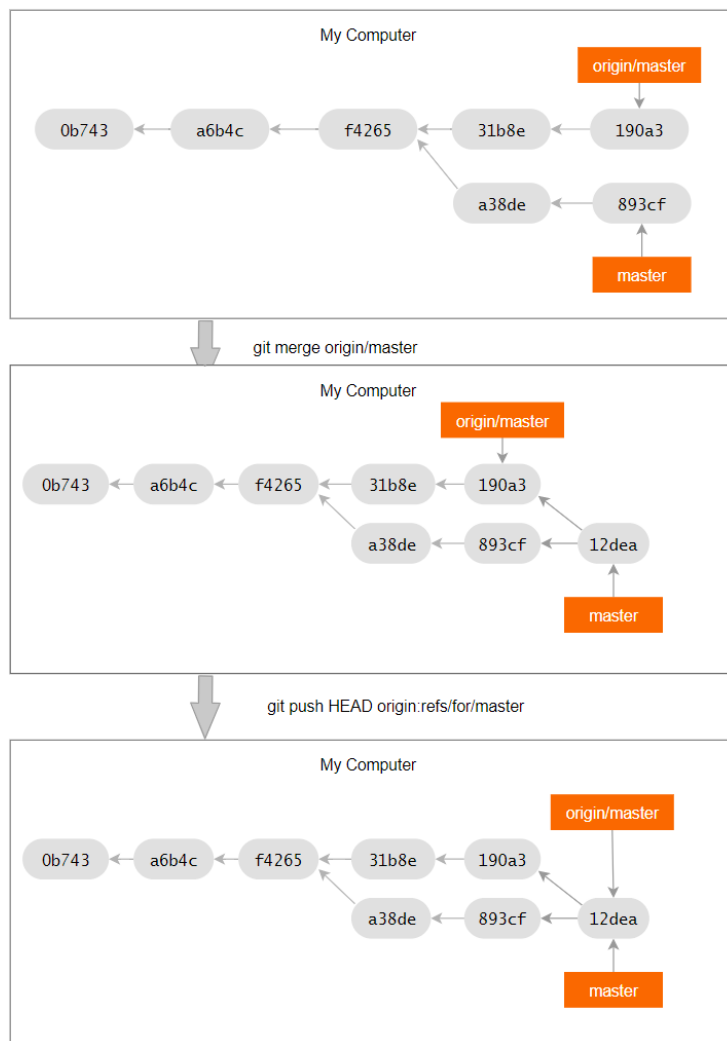
## 5.5 合并分支 git merge



# 将两个分支合成一个分支

`git merge origin/master`

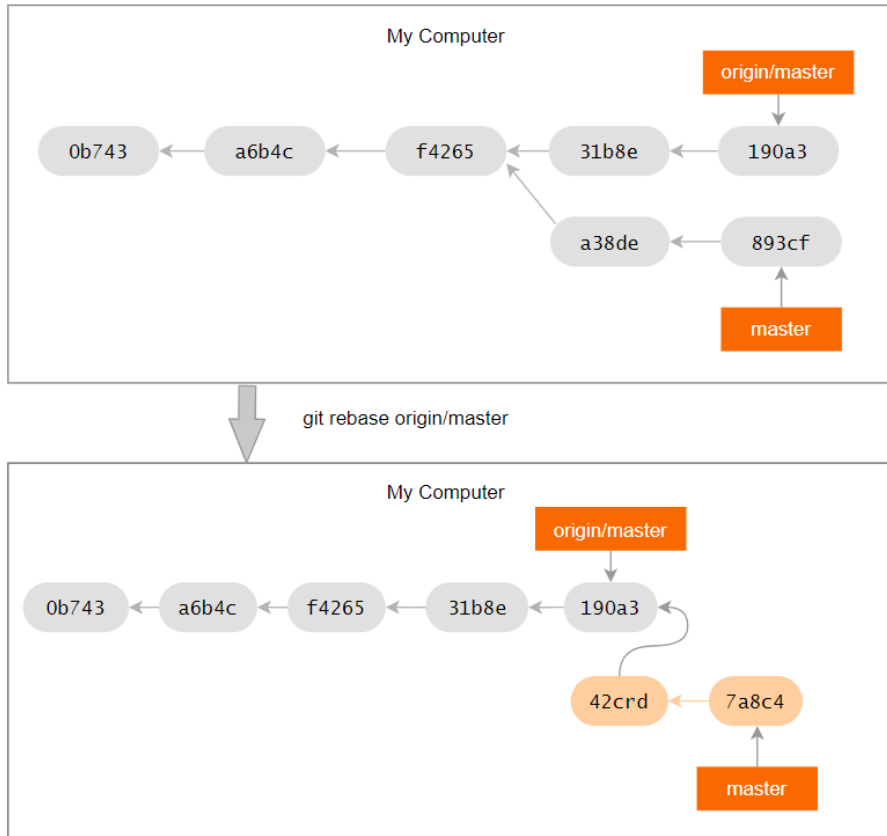
# 表示将远程分支 `origin/master` 合并到当前本地分支，生成一个新的节点。



## 4.5 合入分支 git rebase

```
git rebase origin/master
```

# rebase 会把你当前分支新增的 commit 放到公共分支的最后面，如下图，对比 origin/master，新增了两个 commit（因为是新增的 commit，哈希值跟 master 上的肯定不一样，会改变），所以将两个 commit 排列到 origin/master 之后。



---

## 六、回退操作 git reset 和 git revert

### 6.1 git reset(-hard、-soft、-mixed)

git reset 三个命令就是用于本地工作区域撤回动作。

# 将文件提交至本地仓库后撤回暂存区的操作

```
git reset --soft xxx
```

```
taobb@sileadinc:~/workdir/temp/git$ git ls
2c0457b 2020-07-03 (HEAD, master) add hello world file [taobb]
24c2f1b 2020-07-03 made a change [taobb]
taobb@sileadinc:~/workdir/temp/git$ git reset --soft HEAD^
taobb@sileadinc:~/workdir/temp/git$ git status .
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   hello.c
    new file:   world.c
```

# HEAD 指针表示当前分支的 commit 节点，HEAD^表示当前节点的前一个节点，图中 git reset --soft HEAD^表示回退一个 commit，同样的 HEAD^^ 表示回退两个节点。

# 将文件提交至本地仓库后撤回工作区的操作，--mix 可以省略

```
git reset (--mix) xxx
```

```
taobb@sileadinc:~/workdir/temp/git$ git ls
2c0457b 2020-07-03 (HEAD, master) add hello world file [taobb]
24c2f1b 2020-07-03 made a change [taobb]
taobb@sileadinc:~/workdir/temp/git$ git reset --mix HEAD^
taobb@sileadinc:~/workdir/temp/git$ git status .
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    hello.c
    world.c
```

#将文件提交至本地仓库后撤回删除所有修改

```
git reset --hard xxx
```

```
taobb@sileadinc:~/workdir/temp/git$ git ls
2c0457b 2020-07-03 (HEAD, master) add hello world file [taobb]
24c2f1b 2020-07-03 made a change [taobb]
taobb@sileadinc:~/workdir/temp/git$ git reset --hard HEAD^
HEAD is now at 24c2f1b made a change
taobb@sileadinc:~/workdir/temp/git$ git status .
On branch master
nothing to commit, working directory clean
```

注：git reset 是 git 中一个比较重要的命令，git reset --hard 常用于清空环境，工作区未提交(git commit)的修改若使用-hard 操作，将无法找回，谨慎使用。若修改内容已提交 commit，可通过 git reflog 命令找回 commit 号恢复。

---

## 6.2 git revert

`git revert` 是一种反做命令，用一笔提交 (commit) 回退某 commit 的操作。比如 `commit1` 提交了一个文件，`git revert commit1` 则表示使用 `commit2` 来回退 `commit1` 操作。

```
taobb@sileadinc:~/workdir/temp/git$ git ls
2c0457b 2020-07-03 (HEAD, master) add hello world file [taobb]
24c2f1b 2020-07-03 made a change [taobb]
taobb@sileadinc:~/workdir/temp/git$
taobb@sileadinc:~/workdir/temp/git$ git show 2c0457b
commit 2c0457bf0f754a1ba3f6c0101a03e2ac098fe9ac
Author: taobb <luke_tao@sileadinc.com>
Date: Fri Jul 3 17:06:46 2020 +0800

    add hello world file

diff --git a/hello.c b/hello.c
new file mode 100644
index 0000000..e69de29
diff --git a/world.c b/world.c
new file mode 100644
index 0000000..e69de29
taobb@sileadinc:~/workdir/temp/git$ git revert 2c0457b ←
[master 1267faf] Revert "add hello world file"
 2 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 hello.c
 delete mode 100644 world.c
taobb@sileadinc:~/workdir/temp/git$ git ls
1267faf 2020-07-03 (HEAD, master) Revert "add hello world file" [taobb]
2c0457b 2020-07-03 add hello world file [taobb]
24c2f1b 2020-07-03 made a change [taobb]
```



## 七、其他基本操作

### 7.1 git alias 使用

# 使用如下命令, 在~/.gitconfig 中增加别名。  
git config --global alias.ls --pretty=format:\ "%C(yellow)%h %C(blue)%ad %C(red)%d %C(reset)%s %C(green) [%cn]\ " --decorate --date=short

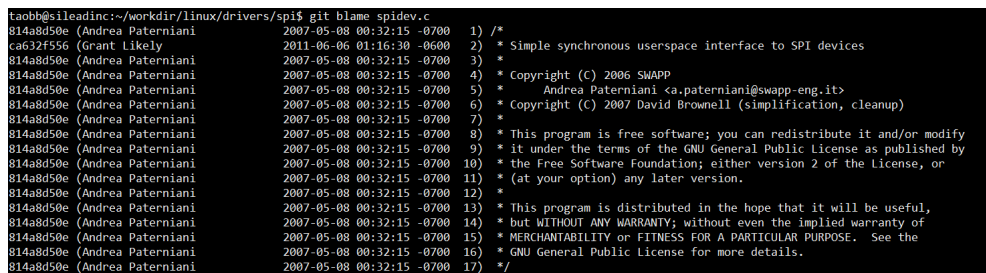
配置之后, 可以使用 git ls 命令更直观查看 git 版本更新。(git ls -10 表示前 10 次提交)



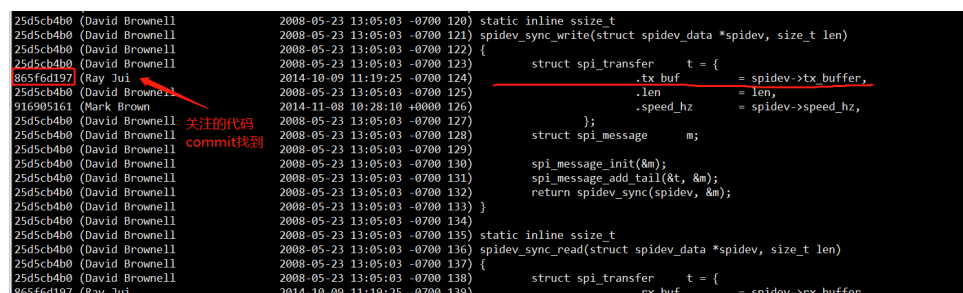
### 7.2 git blame 查找一行代码提交记录

- 1.git blame xxx.c
- 2.定位到某一行代码的 commit
- 3.git show commit

step1: git blame spidev.c



step2: 使用空格逐行查找或者直接输入行数定位制定位置



step3: git show xxx

```
taobb@sileadinc:~/workdir/linux/drivers/spi$ git show 865f6d197
commit 865f6d1974ddd9eff7e10820c4f9f7c7179d6659
Author: Ray Jui <rjui@broadcom.com>
Date: Thu Oct 9 11:19:25 2014 -0700

    spi: spidev: Use separate TX and RX bounce buffers

    By using separate TX and RX bounce buffers, we avoid potential cache
    flush and invalidation sequence issue that may be encountered when a
    single bounce buffer is shared between TX and RX

    Signed-off-by: Ray Jui <rjui@broadcom.com>
    Reviewed-by: JD (Jiandong) Zheng <jdzheng@broadcom.com>
    Signed-off-by: Mark Brown <broonie@kernel.org>

diff --git a/drivers/spi/spidev.c b/drivers/spi/spidev.c
index e3bc23b..e50039f 100644
--- a/drivers/spi/spidev.c
+++ b/drivers/spi/spidev.c
@@ -82,10 +82,11 @@ struct spidev_data {
     struct spi_device      *spi;
     struct list_head       device_entry;

-    /* buffer is NULL unless this device is open (users > 0) */
+    /* TX/RX buffers are NULL unless this device is open (users > 0) */
     struct mutex            buf_lock;
     unsigned               users;
-    u8                     *buffer;
+    u8                     *tx_buffer;
+    u8                     *rx_buffer;
};
```

## 7.3 git stash 暂存工作区修改

# 毫不夸张地说 git stash 是一个极度实用的 Git 操作，stash 可以把当前工作现场“保存”起来，等以后恢复现场后继续工作

```
git stash          //保存当前工作区内容
git stash pop      //取出来不保留
git stash apply    //取出来还保留，需要用 git stash drop 删除
git stash list
git stash clear
```

---

## 参考

引用 1: [【Git】\(1\)—工作区、暂存区、版本库、远程仓库 - 雨点的名字 - 博客园](#)

引用 2: [Git - 分支简介](#)

引用 3: [git reset 与 git revert - 简书](#)