

# Git学习

---

## 参考链接:

### Git学习

- 集中式与分布式版本控制系统

- 安装Git

- 版本库管理

  - 初始化仓库

  - 添加文件

  - 查看状态

  - 版本回退

- 工作区与暂存区

  - 管理修改

  - 撤销修改

  - 删除文件

- 远程仓库

  - 设置远程仓库SSH Key

  - 添加远程仓库

  - 从远程仓库克隆

- 分支管理

  - 解决冲突

  - 分支策略

  - bug分支

  - 多人协作

  - rebase

  - tag标签

- GitHub使用

- Gitee使用

- 配置Git

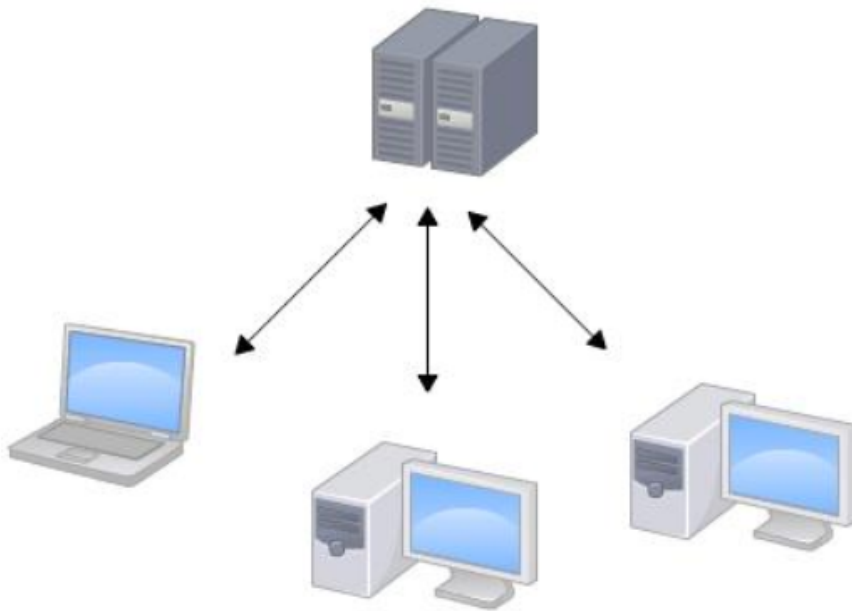
  - 忽略特殊文件

  - 配置文件

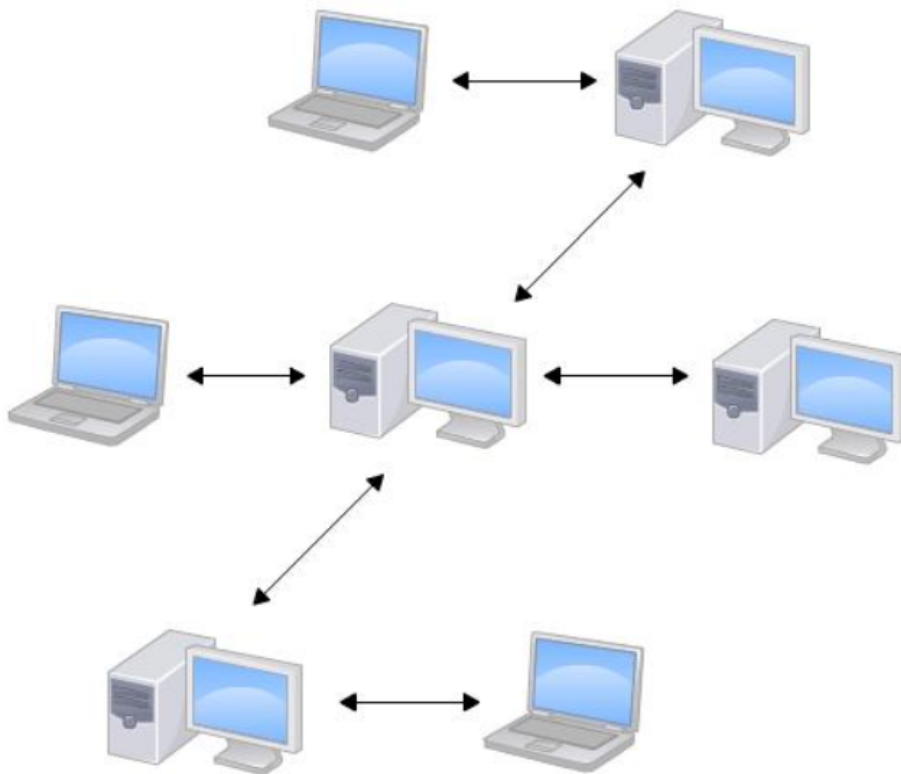
## 集中式与分布式版本控制系统

---

集中式：有一个中央服务器，版本库放在中央服务器中。从中央服务器中获取最新的版本，进行修改后，再推送给中央服务器。最大的缺点是需要联网才能工作。而且中央服务器挂了就全挂了。



分布式：每个人的电脑中都有完整的版本库，工作时不需要联网。理论上可以两两之间交换各自的修改，但是实际上通常有一台电脑充当中央服务器的角色，方便大家的交换。



集中式代表：CVS,SVN

分布式代表：Git

## 安装Git

Linux Debian系: `sudo apt install git`

windows:官网下载安装包安装即可

安装完成后，通过下面两个命令配置一下用户名和邮箱：

```
1 | git config --global user.name "name"
2 | git config --global user.email "email"
```

注意--global参数使得机器上所有仓库都会使用这个配置。

## 版本库管理

版本库又名仓库，英文repository。仓库里面的所有文件都被Git管理，可以对文件进行跟踪、备份、还原。

### 初始化仓库

```
1 | git init
```

在一个空目录中（当然非空目录也可以），输入命令 `git init` 即可。这时仓库目录下多出了一个.git目录，里面包含Git用来管理仓库的文件，不要手动修改。

### 添加文件

```
1 | git add filename1 filename2 filename3 ...
2 | git commit -m "comment"
```

### 查看状态

```
1 | git status
```

如果修改了文件，会返回 `Changes not staged for commit`

可以使用 `git diff filename` 查看文件修改前后的变化。

然后使用 `git add filename`，此时输入 `git status` 会返回 `changes to be committed`

然后使用 `git commit -m "comment"` 提交修改。此时输入 `git status` 会返回 `nothing to commit, working tree clean`

### 版本回退

每一次 `commit`，相当保存一次快照，当出现问题时，可以进行回退恢复，不至于从头开始。

```
1 | git log //显示最近的commit信息
2 | git log --pretty=oneline //一行显示一个commit信息
```

每commit一次，形成一个历史版本。`git log` 会按照从新到旧，返回commit的id，id是用SHA1计算出来的数字，用十六进制表示。HEAD表示当前版本，HEAD^ 表示上一个版本，HEAD^^ 表示上上个版本。上百个版本可以写作HEAD~100

版本回退：

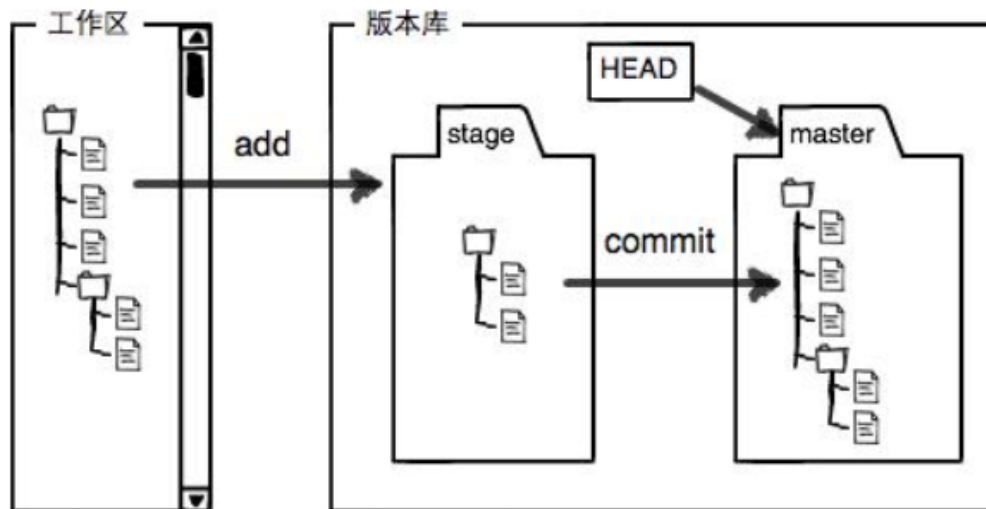
```
1 | git reset --hard HEAD^ //回退到上个版本
2 | git reset --hard commit_id //回退到某个历史版本。commit_id 可以写前几位，能够唯一区分即可
```

版本回退撤销：

```
1 git reflog //查看历史命令，找到回退前的版本id
2 git reset --hard commit id //回到回退前的版本
```

## 工作区与暂存区

工作区就是存放文件的目录，工作区隐藏目录.git就是版本库。版本库里面含有暂存区（stage或者index）以及默认生成的第一个分支master，以及指向master的指针HEAD。



add命令把文件修改提交到暂存区，commit命令一次性提交暂存区所有的文件修改。

## 管理修改

**Git跟踪的是修改，而非文件本身。**

执行操作：第一次修改 -> `git add` -> 第二次修改 -> `git commit`

第二次修改没被提交，需要add到暂存区后再commit。

可以用 `git diff HEAD -- filename` 命令查看工作区和版本库最新版本某文件的区别

## 撤销修改

场景1：直接丢弃工作区的修改时，用命令 `git checkout -- file`。此时

1. 修改之前没有add file到暂存区，会回到最近一次commit时的状态。
2. 修改file之前，已经add file到暂存区，会回到add file到暂存区的状态。

场景2：不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令 `git reset HEAD file`，将暂存区的修改撤销，就回到了场景1的1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退，不过前提是没有推送到远程库。

## 删除文件

删除也是修改，如果工作目录删除了文件，git status会显示修改没有加入暂存区。此时有两种情况

1. 确认删除，从版本库删除文件。

```
1 | git rm filename
2 | git commit //提交删除文件的修改
```

2. 删错了，从版本库恢复文件。（其实就是上面的[撤销修改](#)）

```
1 | git checkout -- filename
```

注意，恢复文件只是用版本库的文件进行恢复，恢复前没有提交的修改内容，就没了。

## 远程仓库

Git是分布式版本控制系统，同一个仓库可以分布到不同的机器上。通常情况下，有一台充当服务器的机器，大家从服务器克隆仓库到自己电脑上，然后在本地修改后再提交到服务器，也从服务器仓库拉取别人的提交。

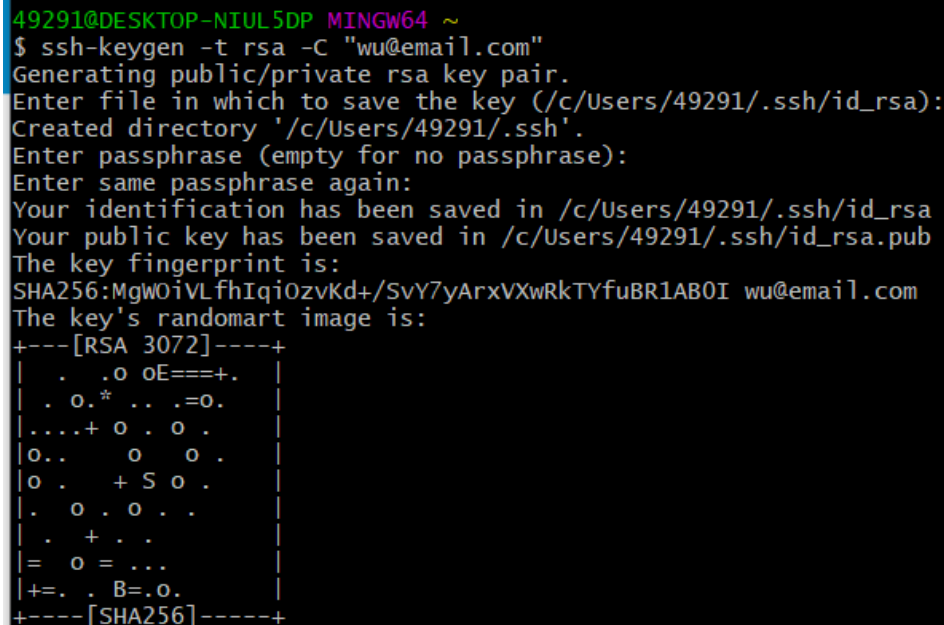
GitHub是最著名的Git仓库托管提供者，可以免费获得Git远程仓库。

## 设置远程仓库SSH Key

1. shell或者Git bash输入命令

```
ssh-keygen -t rsa -C "your_email"
```

出现下图，按提示操作即可。



```
49291@DESKTOP-NIUL5DP MINGW64 ~
$ ssh-keygen -t rsa -C "wu@email.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/49291/.ssh/id_rsa):
Created directory '/c/Users/49291/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/49291/.ssh/id_rsa
Your public key has been saved in /c/Users/49291/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:Mgw0iVLfhIqiOzvKd+/SvY7yArxVXwRkTYfuBR1ABOI wu@email.com
The key's randomart image is:
+---[RSA 3072]-----+
|  . .o oE==+. |
|  . o.* .. .o. |
|...+ o . o . |
|o.. o o . |
|o . + S o . |
| . o . o . . |
| . + . . |
|= o = ... |
|+=. . B=.o. |
+---[SHA256]-----+
```

此时在用户主目录或者上图自行输入的目录下，出现.ssh文件夹，里面包含id\_rsa私钥和id\_rsa.pub公钥。

2. 在GitHub账户的设置页面，选择SSH Key，填上title，然后在Key中粘贴id\_rsa.pub公钥内容。

## 添加远程仓库

```
1 | git remote add origin git@github.com:github_id/repository_name.git //关联远程
   | 仓库，远程仓库的名字设为origin，地址git@github.com:github_id/repository_name.git
2 | git push -u origin master //把当前分支master push到远程仓库origin，参数u在第一次
   | push时使用，将远程master分支与本地master分支关联，之后不再需要。
```

## 从远程仓库克隆

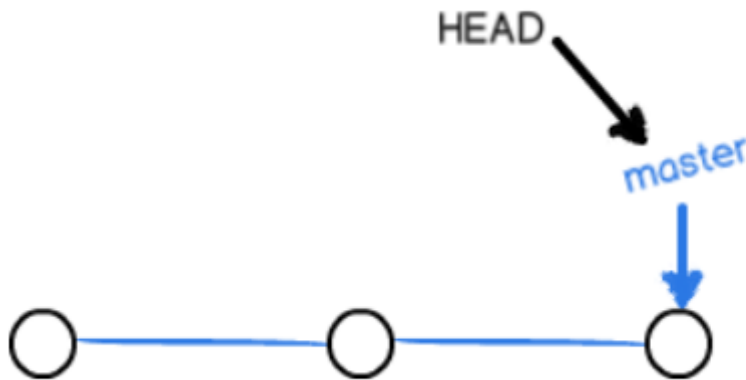
```
1 | git clone git@github.com:github_id/repository_name.git //SSH协议，速度更快。也  
   | 可以使用https://github.com/github_id/repository_name.git这样的HTTPS协议。
```

## 分支管理

通过创建分支，每个人在自己的分支上工作，互不干扰，开发完毕后再合并到主分支上。鼓励使用分支完成某个任务，合并后再删除分支，这样比直接在master上操作更安全。

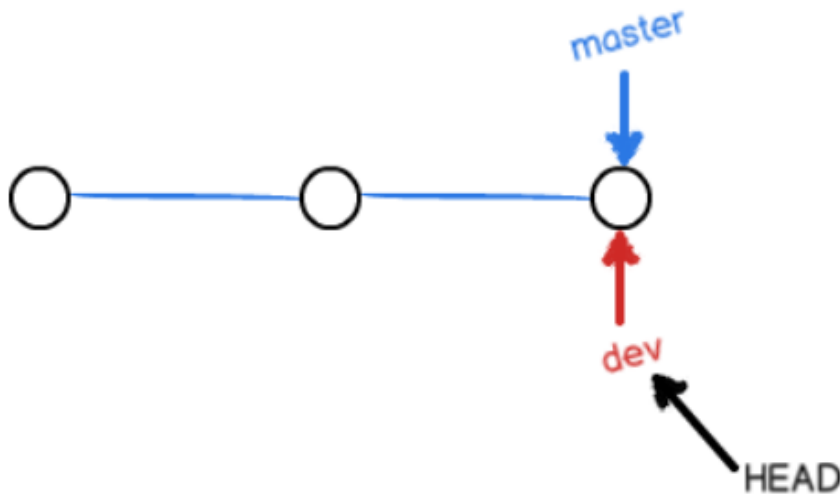
在Git里，有一个主分支，即 `master` 分支。`HEAD` 严格来说不是指向提交，而是指向 `master`，`master` 才是指向提交的，所以，**`HEAD` 指向的就是当前分支。**

一开始的时候，`master` 分支是一条线，Git用 `master` 指向最新的提交，再用 `HEAD` 指向 `master`，就能确定当前分支，以及当前分支的提交点：

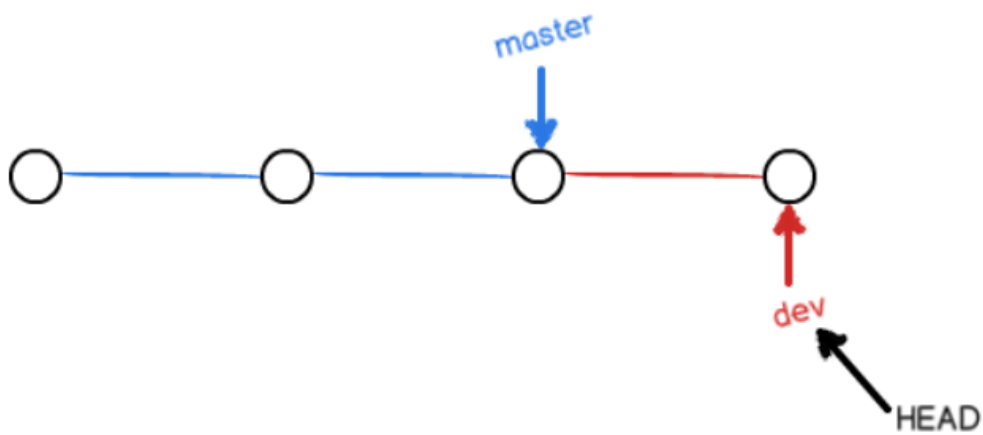


每次提交，`master` 分支都会向前移动一步，这样，随着你不断提交，`master` 分支的线也越来越长。

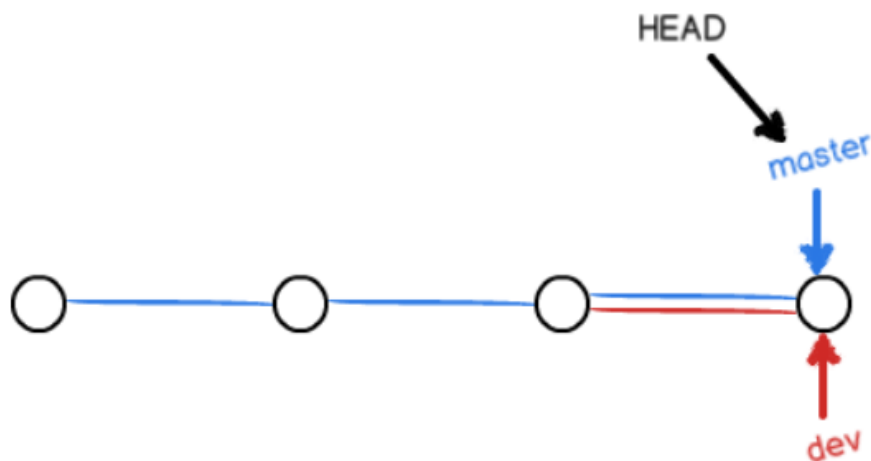
当我们创建新的分支，例如 `dev` 时，Git新建了一个指针叫 `dev`，指向 `master` 相同的提交，再把 `HEAD` 指向 `dev`，就表示当前分支在 `dev` 上：



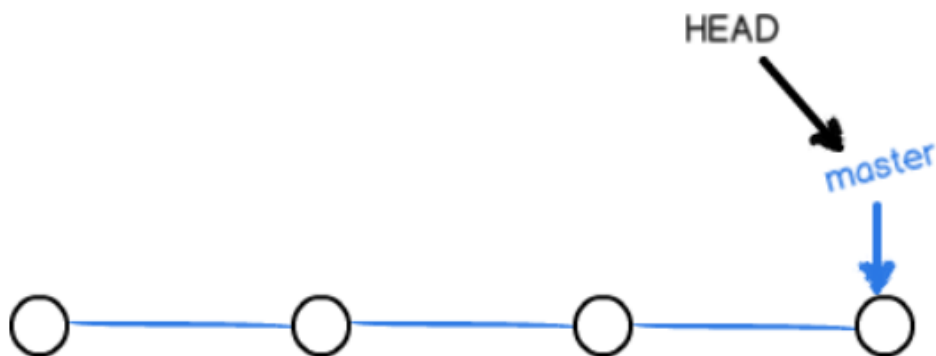
不过，从现在开始，对工作区的修改和提交就是针对 `dev` 分支了，比如新提交一次后，`dev` 指针往前移动一步，而 `master` 指针不变：



假如我们在 `dev` 上的工作完成了，就可以把 `dev` 合并到 `master` 上。Git怎么合并呢？最简单的方法，就是直接把 `master` 指向 `dev` 的当前提交，就完成了合并（Fast forward 模式）：



合并完分支后，甚至可以删除 `dev` 分支。删除 `dev` 分支就是把 `dev` 指针给删掉，删掉后，我们就剩下了一条 `master` 分支：

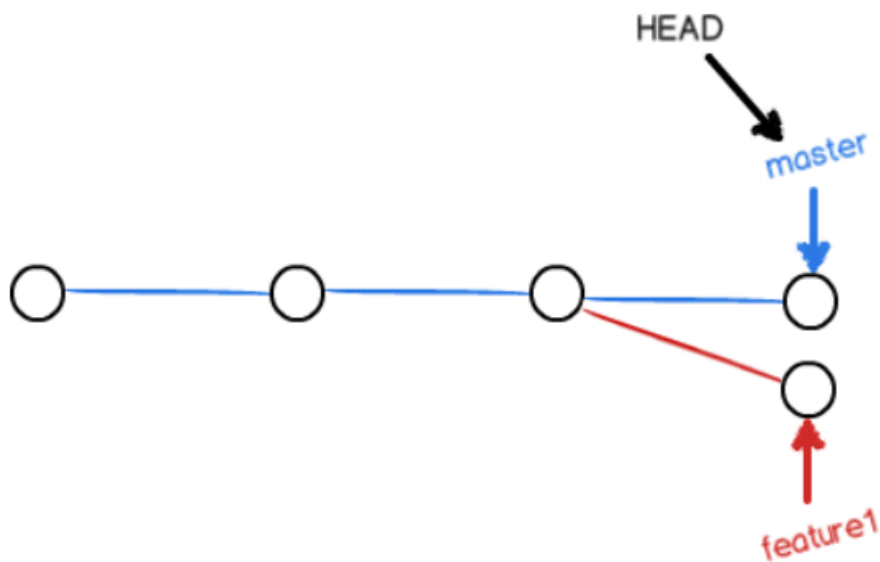


相关命令：

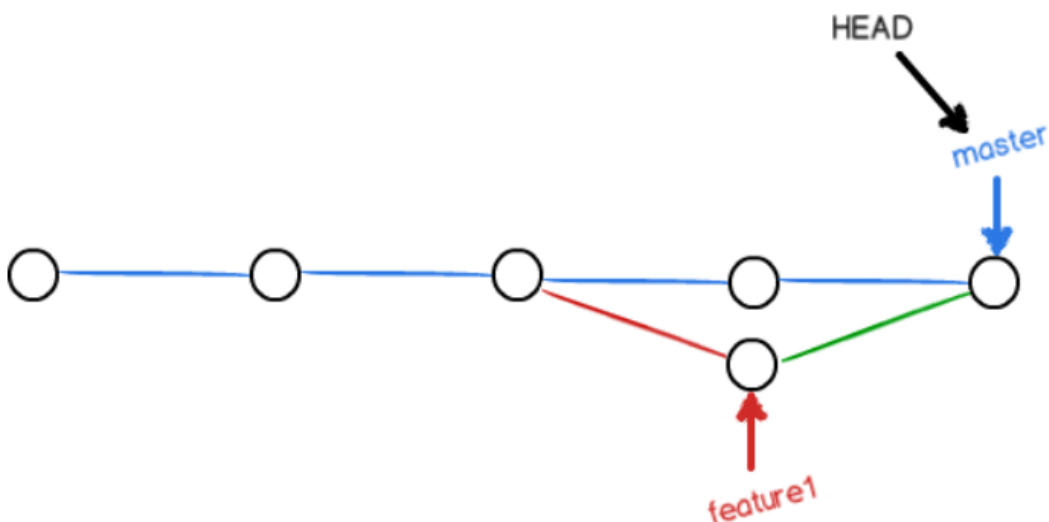
```
1 | git branch //查看分支
2 | git branch branch_name //创建分支
3 | git checkout branch_name //切换分支，或者git switch branch_name
4 | git checkout -b branch_name //创建并切换分支
5 | git merge branch_name //合并某分支到当前分支
6 | git branch -d branch_name //删除分支
```

## 解决冲突

如下情景：



这种情况下，Git无法进行快速合并，只能试图把各自的修改合并起来。合并过程可能产生冲突（如同时修改了某个文件的相同部分），这时需要手动编辑相关文件，在当前分支进行一次commit，解决冲突。冲突解决后，成功合并的结果如下。



可以使用 `git log --graph --pretty=oneline --abbrev-commit` 看分支合并情况。

```
$ git log --graph --pretty=oneline --abbrev-commit
*   cf810e4 (HEAD -> master) conflict fixed
| \
| * 14096d0 (feature1) AND simple
* | 5dc6824 & simple
| /
* b17d20e branch test
* d46f35e (origin/master) remove test.txt
* b84166e add test.txt
* 519219b git tracks changes
* e43a48b understand how stage works
* 1094adb append GPL
* e475afc add distributed
* eaadf4e wrote a readme file
```



## 分支策略

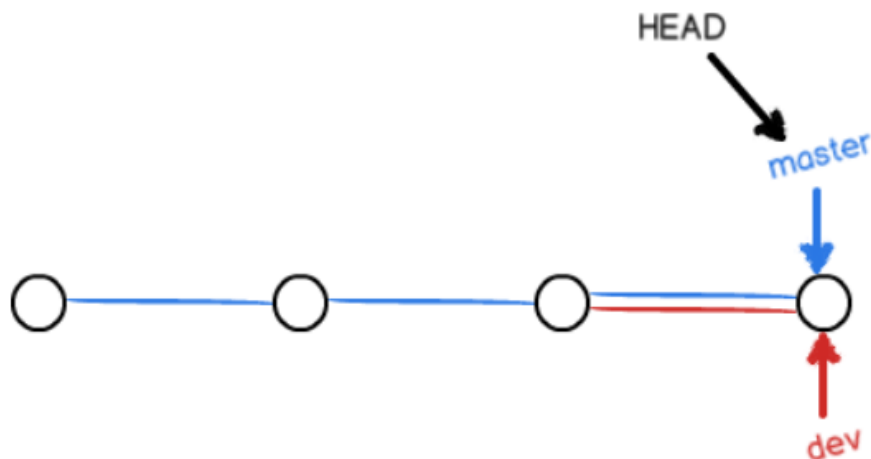
通常，合并分支时，如果可能，Git会用 `Fast forward` 模式，但这种模式下，删除分支后，会丢掉分支信息。

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

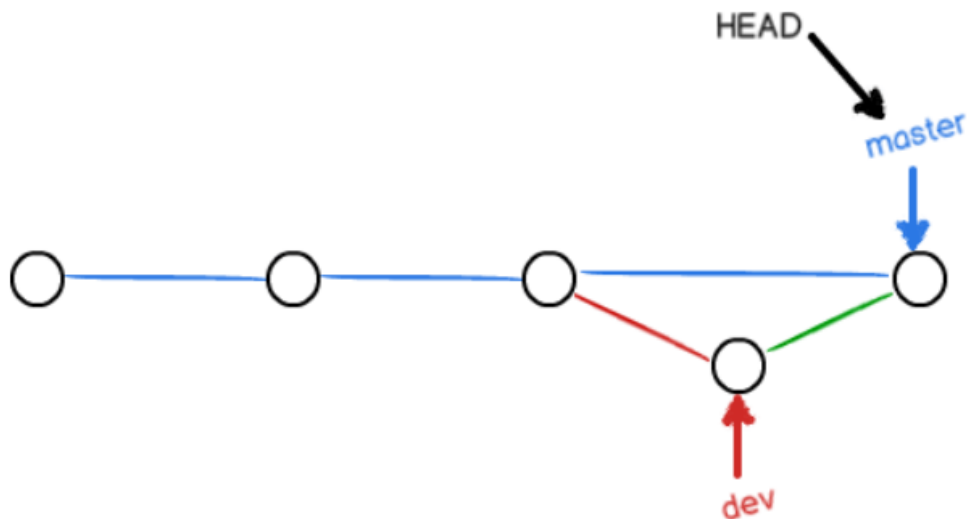
命令：

```
1 | git merge --no-ff -m "merge with no-ff" dev //提交一个commit, m参数表示描述
```

`Fast forward` 模式:



普通模式:



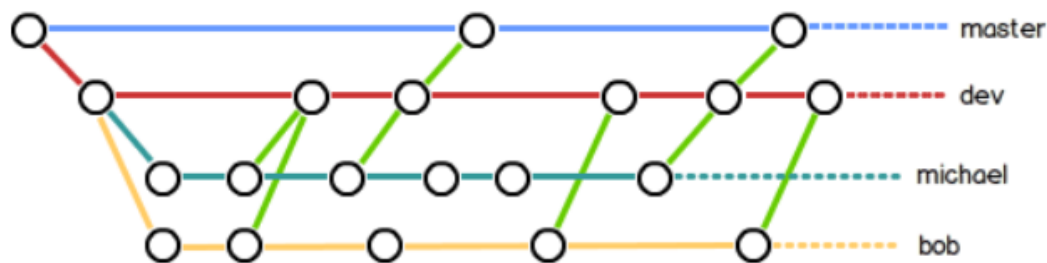
在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，`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；

那在哪干活呢？干活都在 `dev` 分支上，也就是说，`dev` 分支是不稳定的，到某个时候，比如1.0版本发布时，再把 `dev` 分支合并到 `master` 上，在 `master` 分支发布1.0版本；

每个人都有自己的分支，时不时地往 `dev` 分支上合并就可以。

结果看起来就是：



## bug分支

假设当前在dev分支工作，但是需要修复master分支的bug，此时将工作现场储存起来，等bug修复完成后再继续工作。

```
1 | git stash    //储存当前工作现场
2 | git stash list  //显示stash的分支
3 | git stash apply //恢复当前分支现场，但是stash的内容不删除
4 | git stash pop   //恢复当前分支，同时删除stash内容
```

若master的bug存在于dev分支中，通过cherry-pick命令来将修改bug的提交整合到dev分支中，而不是将整个master分支merge进来，避免重复劳动。

```
1 | git cherry-pick commit_id    //将某个提交整合进当前分支
```

## 多人协作

克隆远程仓库后，Git自动将本地master和远程master关联起来。

```
1 | git remote    //查看远程库
2 | git remote -v  //查看远程库详细信息
3 | git push origin master  //推送本地master所有提交到远程库对应的分支master
4 | git push origin dev    //推送本地dev所有提交到远程库对应的分支dev
5 | git branch --set-upstream-to=origin/branch_name branch_name //关联本地分支与远程分支
6 | git checkout -b branch_name origin/branch_name //创建和远程分支对应的本地分支
7 | git pull      //拉取远程仓库对应分支的最新提交
8 | git push      //推送本地分支提交到远程仓库对应分支
```

推送本地提交到远程，若失败，则先pull，若pull有冲突，先处理冲突，再pull，再push。

## rebase

rebase操作可以把本地未push的分叉提交历史整理成直线，使得历史提交的变化看上去更直观。缺点是本地的分叉提交已经被修改过了。

```
1 | git rebase
```

rebase前

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

rebase后

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

Git把本地提交挪动了位置，放到了 f005ed4 (origin/master) set exit=1 之后。rebase操作前后，最终的提交内容是一致的，但是，我们本地的commit修改内容已经变化了，它们的修改不再基于 d1be385 init hello，而是基于 f005ed4 (origin/master) set exit=1，但最后的提交 7e61ed4 内容是一致的。

最后，通过push把本地分支推送到远程。

## tab标签

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的時刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针，所以，创建和删除标签都是瞬间完成的。

tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。引入标签就是为了方便人的记忆和使用。

标签总是和某个commit挂钩，所有出现这个commit的分支，都可以看到这个标签。

```
1 git tag tag_name //当前分支最新提交的commit上打上tag
2 git tag tag_name commit_id //当前分支的commit_id上打上tag
3 git tag -a tag_name -m "tag comment" //a制定tag名，m指定tag说明
4 git tag //查看所有标签
5 git tag show tag_name //查看标签信息
6 git tag -d tag_name //删除标签
7 git push origin tag_name //推送tag到远程仓库
8 git push origin --tags //一次性推送尚未推送的所有标签到远程仓库
9 git push origin :refs/tags/tag_name //删除远程标签
```

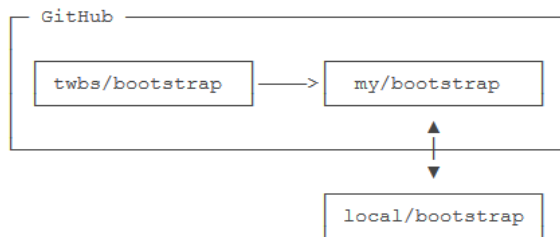
## GitHub使用

- 在GitHub上，可以任意Fork开源仓库；
- 自己拥有Fork后的仓库的读写权限，在自己fork的仓库工作。

- 工作完成后，可以推送pull request给官方仓库来贡献代码。

例如：

Bootstrap的官方仓库 `twbs/bootstrap`、你在GitHub上克隆的仓库 `my/bootstrap`，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：



## Gitee使用

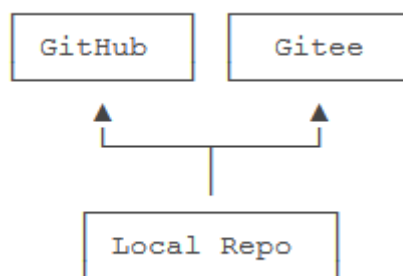
使用Gitee和使用GitHub类似，我们在Gitee上注册账号并登录后，需要先上传自己的SSH公钥。选择右上角用户头像 -> 菜单“修改资料”，然后选择“SSH公钥”，填写一个便于识别的标题，然后把用户主目录下的 `.ssh/id_rsa.pub` 文件的内容粘贴进去。

本地库关联多个远程库：

默认远程库名字是origin，关联多个远程库时，自行命名远程库即可。

例如：

```
1 git remote add github repo_url
2 git remote add gitee repo_urla
3 git push github master
4 git push gitee master
```



## 配置Git

### 忽略特殊文件

有些时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件啦，等等，每次 `git status` 都会显示 `Untracked files ...`。这时候，在工作区目录下创建一个特殊的 `.gitignore` 文件，把要忽略的文件名放进去就可以。然后把 `.gitignore` 文件提交到Git。

GitHub有一个模板，在里面找到需要的，组合一下就行。<https://github.com/github/gitignore>

忽略文件的原则是：

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。

例如：

```
1 # Windows:
2 Thumbs.db
3 ehthumbs.db
4 Desktop.ini
5
6 # Python:
7 *.py[cod]
8 *.so
9 *.egg
10 *.egg-info
11 dist
12 build
13
14 # My configurations:
15 db.ini
16 deploy_key_rsa
```

```
1 git add -f file //忽视忽略规则，强制添加文件
2 git check-ignore -v file //查看某文件被哪条规则忽略，可用来修改.gitignore文件
```

## 配置文件

每个仓库的Git配置文件都放在 `.git/config` 文件中。

例如：

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = git@github.com:michaelliao/learngit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[alias]
    last = log -1
```

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件 `.gitconfig` 中。

例如：

```
$ cat .gitconfig
[alias]
    co = checkout
    ci = commit
    br = branch
    st = status
[user]
    name = Your Name
    email = your@email.com
```

可以直接修改配置文件，或者使用 `git config` 命令。加上 `--global` 参数后，对当前用户起作用，否则只对当前仓库起作用。

例：

配置命令别名

```
1 | git config alias.st status //git st 表示git status
```