廖雪峰Git教程学习笔记

概要

最近学习廖雪峰老师的git教程，现整理成文档，方便以后查看

一、Git 和 SVN 的区别

	类型	描述
Git	分布式	本地有镜像,无网络时也可以提交到本地镜像,待到有网络时再push到服务器
SVN	集中式	无网络不可以提交, 和 Git 的主要区别是历史版本维护的位置

二、Git 安装 :

Git 下载地址 (Linux/Unix, Mac, Windows 等相关平台)

```
### linux安装git
$ sudo apt-get install git
### 或者下载安装包，依次执行：./config, make, sudo make install

### 在 windows下更新git 版本
$ git update-git-for-windows
```

三、本地仓库操作

> 注意：以下所有命令都是在 Git Bash 中运行，不是 cmd

1. 查看Git 版本号

```
$ git --version                    查看 git 的版本
```

2. git config

```
### 配置所有 Git 仓库的 用户名 和 email
$ git config --global user.name "Your Name"
$ git config --global user.email "youremail@example.com"

### 配置当前 Git 仓库的 用户名 和 email
$ git config user.name "Your Name"
$ git config user.email "youremail@example.com"

### 查看全局配置的 用户名 和 email
$ git config --global user.name
$ git config --global user.email

### 查看当前仓库配置的 用户名 和 email
$ git config user.name
$ git config user.email

# Git 是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址
# git config 命令的 --global 参数，用了这个参数，表示你这台机器上所有的 Git 仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址(不加 --global)。

```

```
$ git config --list     查看所有配置
$ git config --list --show-origin     查看所有的配置以及它们所在的文件
$ git config --global color.ui true     让Git显示颜色，会让命令输出看起来更醒目
```

3. 初始化本地仓库:

```
$ git init              把当前目录初始化为 git 仓库
```

4. 添加文件到仓库

```
$ git add <file>                    如: git add readme.txt
$ git commit -m "description"       如: git commit -m "add readme.txt"

$ git add <file1> <file2>
$ git add .

# 添加文件到仓库分两步:
# 1. add 添加该文件到仓库。添加同种类型的文件,可以使用通配符 * (记得加引号)，如: git add "*.txt"  命令就是添加所有 .txt 文件
# 2. commit 提交该文件到仓库, description 为你对本次提交的描述说明,
  注意: git commit 如果没有-m 及描述，会进入 vi
```

5. 查看仓库当前状态

```
$ git status
```

6. 查看修改内容

```
# git diff     查看工作区(work dict)和暂存区(stage)的区别
# git diff HEAD    查看工作区(work dict)和上一次commit后的区别
# git diff --cached     查看暂存区(stage)和上一次commit后的区别
# git diff HEAD -- <file>     同 git diff HEAD <file>

如: git diff readme.txt     表示查看 readme.txt 在工作区和暂存区的不同，有什么修改
```

7. 查看提交日志

```
$ git log
$ git log --oneline     #美化输出信息,每个记录显示为一行,显示 commit_id 前几位数
$ git log --pretty=oneline     #美化输出信息,每个记录显示为一行,显示完整的commit_id
$ git log --graph --pretty=format:'%h -%d %s (%cr)' --abbrev-commit --
$ git log --graph --pretty=oneline --abbrev-commit

# 显示从最近到最远的提交日志
# 日志输出一大串类似 3628164...882e1e0 的是commit_id (版本号),和 SVN 不一样，Git 的 commit_id 不是 1，2，3…… 递增的数字，而是一个 SHA1 计算出来的一个非常大的数字，用十六进制表示，因为 Git 是分布式的版本控制系统，当多人在同一个版本库里工作，如果大家都用 1，2，3……作为版本号，那肯定就冲突了
# 最后一个会打印出提交的时间等，(HEAD -> master)指向的是当前的版本
# 退出查看 log 日志,输入字母 q (英文状态)
```

8. 版本回退

```
$ git reset --hard HEAD^
$ git reset --hard <commit_id>

# HEAD     表示当前版本，也就是最新的提交
# HEAD^    上一个版本
# HEAD^^   上上一个版本
# HEAD~100    往上100个版本

# 回退到指定 commit_id   commit_id 只需要前几位就行
# git reset            移除所有暂存区的修改，但不会修改工作区
# git reset --hard     移除所有暂存区的修改，并强制删除所有工作区的修改
# git reset HEAD       同 git reset
```

9. git revert

```
$ git revert -n commit-id
#     反做commit-id对应的内容，然后重新commit一个信息，不会影响其他的commit内容
#     使用-n是应用revert后，需要重新提交一个commit信息，然后再推送。如果不使用-n，指令后会弹出编辑器用于编辑提交信息

$ git revert -n commit-idA..commit-idB
#     反做commit-idA到commit-idB之间的所有commit

$ git revert --abort
#     合并冲突后退出：当前的操作会回到指令执行之前的样子，相当于啥也没有干，回到原始的状态

$ git revert --quit
#     合并后退出，但是保留变化：该指令会保留指令执行后的车祸现场

$ git add .
$ git commit -m "提交的信息"
#     合并后解决冲突，继续操作：如果遇到冲突可以修改冲突，然后重新提交相关信息


### Git reset和git revert的区别
```

```
# git reset 是回滚到对应的commit-id，相当于是删除了commit-id以后的所有的提交，并且不会产生新的commit-id记录，如果要推送到远程服务器的话，需要强制推送-f
# git revert 是反做撤销其中的commit-id，然后重新生成一个commit-id。本身不会对其他的提交commit-id产生影响，如果要推送到远程服务器的话，就是普通的操作git push就好了

# 举例：
# 如果已经有A -> B -> C，想回到B：

# 方法一：reset到B，丢失C：
#   A -> B
# 方法二：再提交一个revert反向修改，变成B现场（相当于撤销C的修改）：
#   A -> B -> C -> B'
#   C还在，但是两个B的内容相同，commit id不同

# 根据需求，也许C就是瞎提交错了（比如把密码提交上去了），必须reset
# 如果C就是修改，现在又要改回来，将来可能再改成C，那你就revert
```

10. 查看命令历史

```
$ git reflog
# 假如我们依次提交了三个版本 a->b->c,然后昨天我们从版本 c 回退到了版本 b,今天我们又想要回到版本 c,此时就可以使用 reflog 命令来查找 c 版本的 commit_id,然后使用 reset 命令来进行版本回退
```

11. 撤销修改

```
### 从暂存区恢复工作区（丢弃工作区的修改）
$ git restore readme.txt
$ git resotre --worktree readme.txt
$ git checkout -- readme.txt （discarded）

###  从HEAD恢复暂存区（丢弃暂存区的修改）
$ git restore --staged readme.txt
$ git reset <file> / git reset HEAD <file>

###  从HEAD同时恢复工作区和暂存区
$ git restore --staged --worktree readme.txt
$ git restore --source=HEAD --staged --worktree readme.txt
$ git reset --hard HEAD readme.txt


               git add / git rm                        git commit
    工作区 ----------------------------> 暂存区 ----------------------------> HEAD

                git restore --worktree             git restore --staged
              or git restore                      or git reset
    工作区 <---------------------------- 暂存区 <---------------------------- HEAD

                          git reset --hard HEAD
    工作区              or git restore --staged --worktree
```

```
    暂存区 <-------------------------------------------------------------------
HEAD


# 举例测试：
#   比如文件 test 内容为 A，已提交到历史库，工作区将 test 改为 AA，然后 add 到暂存区，
工作区再将 test 改为 AAA。
#   运行 git restore test，会把工作区的 test 改为 AA。
#   运行 git restore --staged test，会把暂存区的 test 改为 A。
#   运行 git restore --source=HEAD test 会把工作区改为 A （从HEAD恢复工作区，git
status 会提示工作区和暂存区都有更新，不建议使用该 command）
#   运行 git restore --staged --worktree test，会把暂存区和工作区的test改为A
```

12. 删除文件

```
$ git rm <file>

# git rm <file> 相当于执行
- rm <file>
- git add <file>
```

13. .gitignore 设置忽略文件

```
# .gitignore，文件用于设置git更新时忽略的文件

# 某个文件被忽略，查看该文件被哪句忽略：
$ git check-ignore -v App.class
 .gitignore:3:*.class   App.class

.gitignore 中
# 排除所有.开头的隐藏文件：
    .*
# 排除所有.class文件：
    *.class
# 不排除 .gitignore 和 App.class:
    !.gitignore
    !App.class

### 把指定文件排除在.gitignore规则外的写法就是!+文件名
```

> GitHub提供的各种配置文件
>
> 在线生成.gitignore文件

四、相关名词理解：

1. 工作区 (Working Directory)：自己电脑里能看到的目录

2. 版本库 (Repository)：工作区有一个隐藏目录 .git，这个不算工作区，而是 Git 的版本库

> Git 的版本库里存了很多东西，其中最重要的就是称为 stage（或者叫index）的暂存区，还有 Git 为我们自动创建的第一个分支 master，以及指向 master 的一个指针叫 HEAD

五、远程仓库：

1. 创建 SSH Key

```
$ ssh-keygen -t rsa -C "youremail@example.com"
# 填写自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

# 多账号时，需要创建多个SSH Key：
$ ssh-keygen -t rsa -C 'xxxxx@company.com' -f ~/.ssh/gitee_id_rsa
Or
$ ssh-keygen -t rsa -C 'qd_zhangx@126.com'
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\12475/.ssh/id_rsa):
C:\Users\12475/.ssh/id_rsa_gitlab
```

> 如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有 id_rsa 和 id_rsa.pub 两个文件，这两个就是 SSH Key 的秘钥对，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。

2. 在 Github 添加 SSH Key

```
#### 登录 GitHub ，在 Settings 中找到 SSH 设置项，添加新的 SSH Key，设置任意 title，在 Key 文本框里粘贴 id_rsa.pub 文件的内容

# 复制Key用这种方式复制
$ cd ~/.ssh
$ cat id_rsa.pub

$ open ~/.ssh   (Mac 下打开存放 Github 生成的 ssh Key 文件夹)
$ pbcopy < ~/.ssh/id_rsa.pub  Mac 下拷贝生成的公钥内容
```

3. clone 远程库

```
$ git clone git@github.com:michaelliao/gitskills.git
# 本地没有仓库，从远程下载仓库，并关联
# GitHub 支持多种协议,上面是 ssh 协议,还有 https 协议

$ git clone -b <branch name> git@github.com:michaelliao/gitskills.git
# 克隆远程的一个分支到本地
```

> 注意：clone远程库后，默认情况下，只能看到master分支。

4. 关联远程仓库

```
$ git remote add origin git@github.com:michaelliao/learngit.git
# 本地库关联一个远程库 （注意改成自己的）
# 远程库的名字是origin，这是Git默认的叫法，也可以改成别的
# 远程库全名叫 git@github.com:michaelliao/learngit.git

### 本地关联多个远程库。如本地git库是learngit，关联3个远程服务器，并分别将这三个远程库起名 github, gitlab, gitee，一般来说，我们只关联一个远程库，就叫origin
$ git remote add github git@1.2.3.4:/user/learngit.git
$ git remote add gitlab git@2.3.4.5:/user/learngit.git
$ git remote add gitee git@3.4.5.6:/user/learngit.git

### clone远程库后，默认情况下，只能看到master分支。

### 在本地创建和远程分支对应的分支(默认进行关联)
$ git switch -c branch-name origin/branch-name  本地和远程分支的名称最好一致
### 建立本地分支和远程分支的关联
$ git branch --set-upstream-to origin/branch-name branch-name


# 多人协作的工作模式:
# 首先，用git push origin <branch-name> 推送本地的修改;
# 如果推送失败，则因为远程分支比本地的更新，需要先用git pull试图合并;
# 如果合并有冲突，则解决冲突，并在本地提交;
# 没有冲突或者解决掉冲突后，再用git push origin <branch-name>推送就能成功!

# 当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，用命令git branch --set-upstream-to origin/<branch-name> <branch-name>

# 如果远程和本地的版本有冲突，两种方向
#    从远程到本地，会需要先链接本地分支和远程分支，其次会要求合并冲突文件，之后才能成功更新至本地。
#    从本地到远程，指定如果版本冲突，需要先重复从远程到本地的操作，先整合了不同版本之后再推一次。
```

5. 查看关联的远程库

```
$ git remote        查看远程库信息
$ git remote -v     查看远程库详细信息
  origin  git@github.com:michaelliao/learngit.git (fetch)
  origin  git@github.com:michaelliao/learngit.git (push)
```

6. 删除与远程库的关联

```
$ git remote rm origin  解除了本地库和远程库origin的绑定关系，并不是物理上删除了远程库
```

7. 推送到远程仓库

```
$ git push <远程主机名> <本地分支名>:<远程分支名>

### 将本地的master分支推送到origin主机的master分支
$ git push origin master

### 如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。
$ git push origin

### 如果当前分支只有一个追踪分支，那么主机名都可以省略。
$ git push

$ git push -u origin master     第一次推送，使用-u参数，关联本地的master分支和远程的master分支

### 如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。
### 将本地的master分支推送到origin主机，同时指定origin为默认主机
### -u：手动建立追踪关系（tracking）
$ git push -u origin master

### 不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机
$ git push --all origin

### 强制推送
$ git push --force origin
$ git push --force-with-lease origin

### 删除指定的远程分支
$ git push origin :master //推送一个空的本地分支到远程分支
# 等同于
$ git push origin --delete master
```

> 加上了-u参数，Git 不但会把本地的 master 分支内容推送的远程新的 master 分支，还会把本地的 master 分支和远程的master分支关联起来

8. 从远程仓库拉取

```
$ git pull <远程主机名> <远程分支名>:<本地分支名>

### 要取回origin主机的next分支，与本地的master分支合并
$ git pull origin next:master

### 如果远程分支(next)要与当前分支合并，则冒号后面的部分可以省略
$ git pull origin next

### 如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名
$ git pull origin

### 如果当前分支只有一个追踪分支，连远程主机名都可以省略。
$ git pull

### 手动建立追踪关系（tracking）
### 指定master分支追踪origin/next分支
```

```
$ git branch --set-upstream-to origin/next master
 or
$ git branch -u origin/next master

### 如果合并需要采用rebase模式，可以使用--rebase选项。
$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>
```

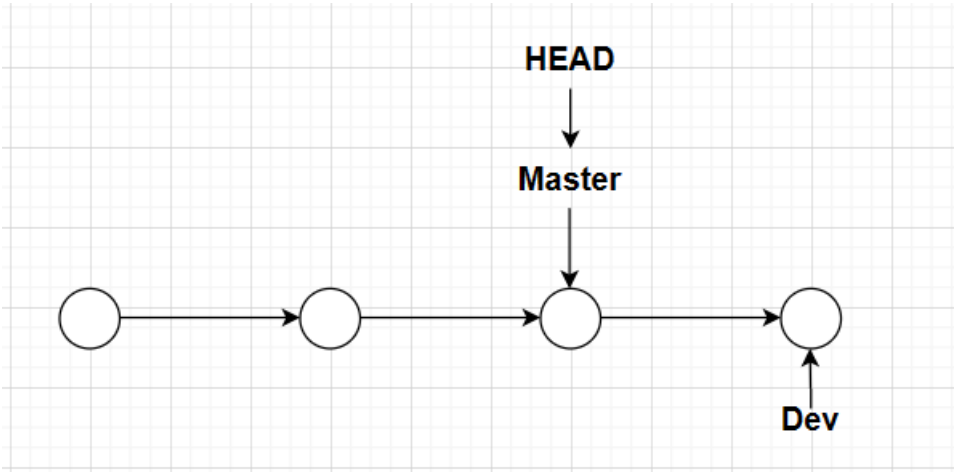
六、分支

```
# HEAD 指向当前分支
# master 指向 master 分支提交点

$ git branch        查看分支列表及当前分支（当前分支前有一个*号）

$ git branch dev    创建 dev 分支
$ git switch dev    切换到 dev 分支  (git checkout dev)
$ git switch -c dev    创建并切换到 dev 分支  (git checkout -b dev)
$ git switch -c dev origin/dev  创建远程 origin 的 dev 分支到本地并切换到该分支

$ git branch -d dev    删除本地 dev 分支
$ git branch -D dev    强制删除本地 dev 分支

$ git merge dev        合并 dev 分支到当前分支 (当有冲突的时候,需要先解决冲突)
$ git merge --no-ff -m "merge with no-ff" dev  合并 dev 分支到当前分支(禁用Fast forward 合并策略)
# 合并分支时，Git默认会用Fast forward模式，这种模式删除分支后，会丢掉分支信息，无法看出曾做过合并
# 加上--no-ff参数会禁用Fast forward模式，Git在merge时生成一个新的commit，合并后的历史有分支，从分支历史上可以看出分支信息

$ git cherry-pick <commit> 复制一个特定的提交到当前分支(当前分支的内容需要先 commit,然后冲突的文件需要解决冲突,然后 commit)

$ git log --graph  查看分支合并图
$ git log --graph --pretty=oneline --abbrev-commit
```

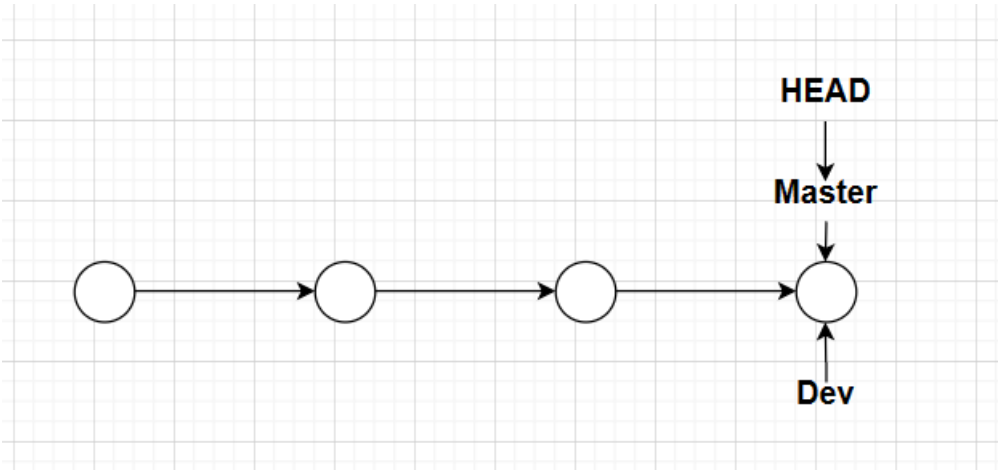
> 合并分支，只会对当前分支进行更新。如当前为 master 分支，git merge dev，只对 master 分支上的内容进行更新，dev 分支的内容不会改变。
>
> 因为工作区和暂存区是所有分支共享，这意味着不同分支间会影响。所以在切换分支前，保证当前工作区和暂存区修改都已提交。或将未提交的内容 stash
>
> 实际项目中如何使用Git做分支管理
>
> 对于所有分支而言，工作区和暂存区是公共的
>
> git 的 merge 与 no-ff merge 测试

分支合并图示：

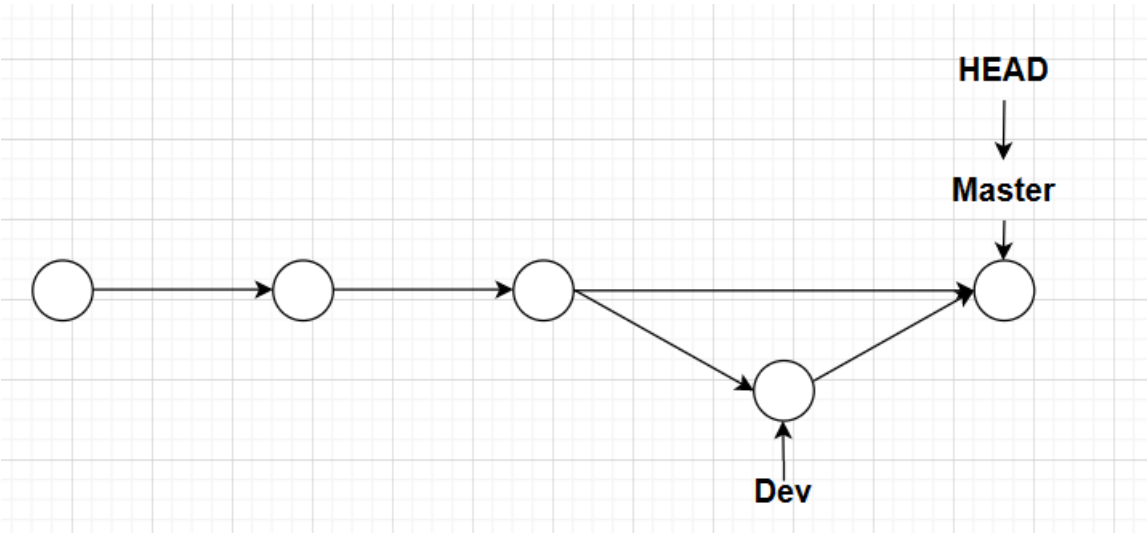
当前分支如下：



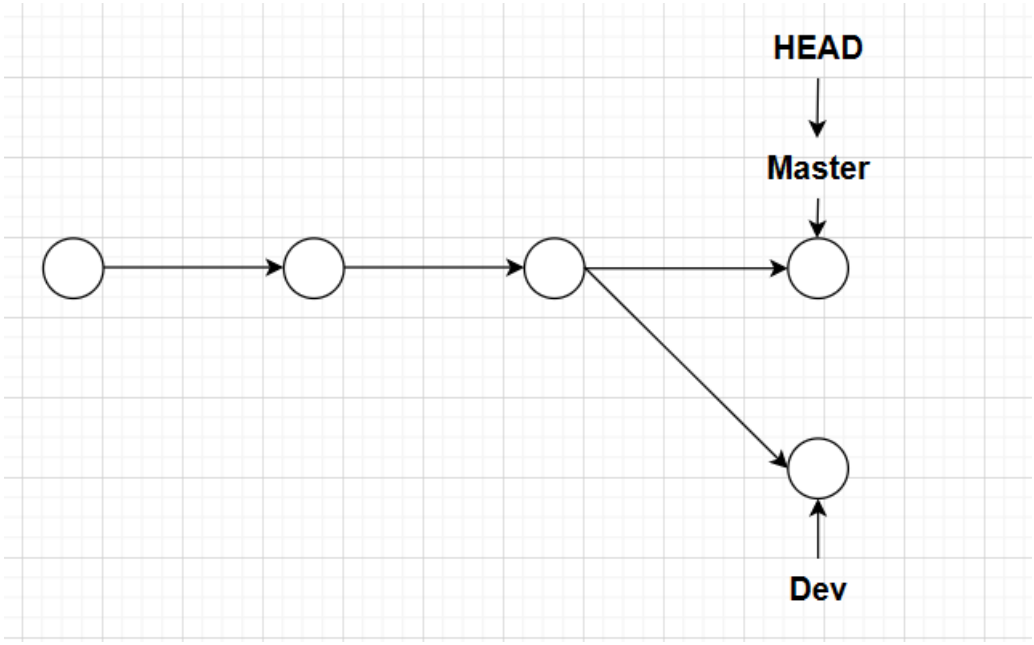
使用Fast forward 合并后，如下：



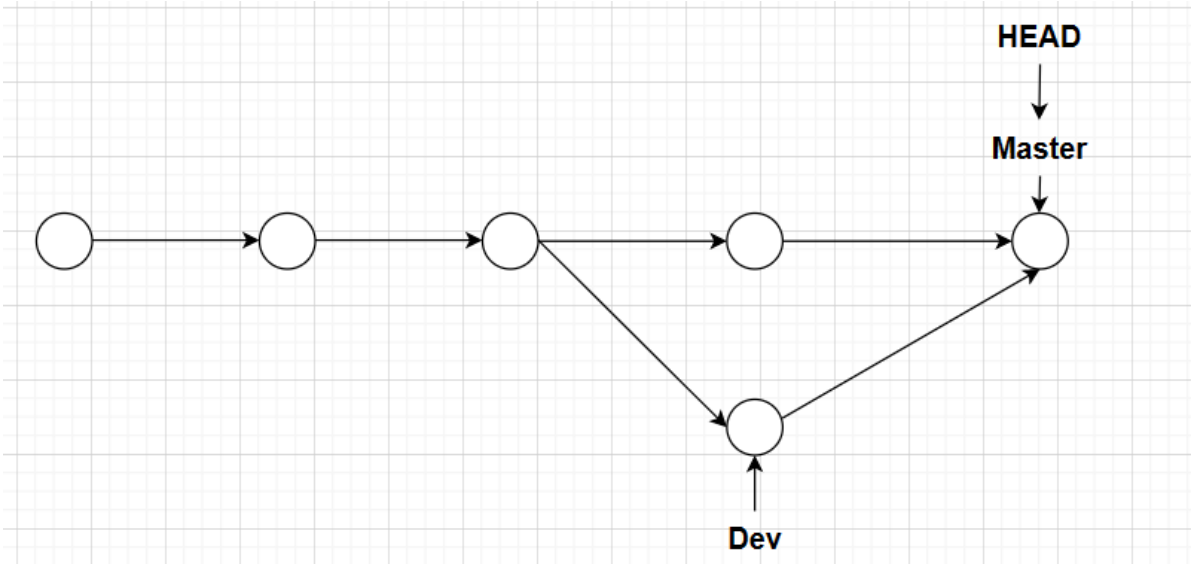
使用 --no-ff (不使用fast forward) 合并后如下（master 多了一个commit 提交）：



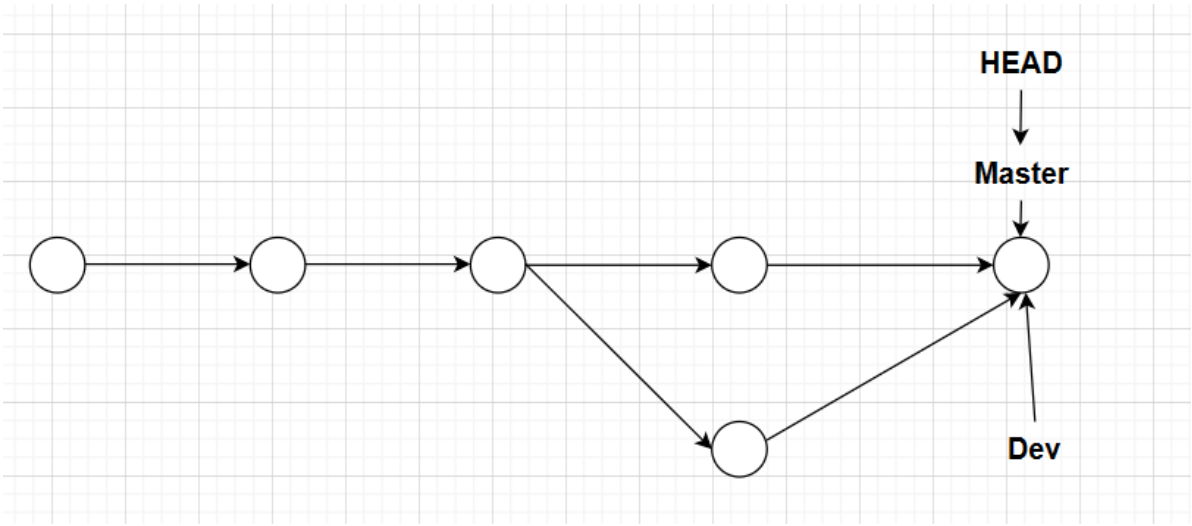
如果两个分支分开后，都有提交，此时合并可能存在冲突，需要首先解决冲突再合并。



解决冲突，并合并：



解决冲突后，master 还可以再合并到（更新）dev 分支。



分支合并小结:

领先分支合并到落后分支，会改变落后分支(常用)
落后分支合并到领先分支，会提示"Already up to data."
两分支分出后，若均有提交（即都有领先），合并时可能出现冲突，如果不冲突也能够合并

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。
解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

```
git merge
    //出现冲突
修改冲突文件
git add .
git commit -m "note"
git branch -d <name>
```

七、标签

```
$ git tag  查看所有的标签（注意不是按时间顺序列出，而是按字母顺序排序）
$ git show <tagname>  查看标签信息

$ git tag <tagname>  打标签(默认标签是打在最新提交的commit上) 如: git tag v1.0
$ git tag <tagname> <commit_id>  给对应的 commit_id 打标签
$ git tag -a <tagname> -m "标签说明信息" <commit_id>  创建带有说明的标签，用-a指定标签名，-m指定说明文字

$ git tag -d <tagname>  删除一个本地标签
$ git push origin :refs/tags/<tagname>  删除一个远程标签

$ git push origin <tagname>  推送一个本地标签到远程
$ git push origin --tags     推送全部尚未推送到远程的本地标签


### 删除远程标签,最好先删除本地标签,然后再删除远程标签, 如:删除标签 v0.9
$ git tag -d v0.9
$ git push origin :refs/tags/v0.9
```

> a) 默认创建的标签都只存储在本地，不会自动推送到远程
>
> b) 标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签。

八、stash

```
$ git stash  保存当前工作区和暂存区的修改状态，git status 查看是干净的
$ git stash save "comment"  保存现场，并添加备注信息
$ git stash list  查看保存现场的列表
$ git stash pop   恢复的同时把 stash 内容也删除
$ git stash apply  恢复现场，stash内容并不删除
$ git stash drop   删除 stash 内容
$ git stash apply stash@{0}  多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash
```

```
$ git stash list
 stash@{index}: WIP on [分支名]: [最近一次的commitID] [最近一次的提交信息]


# 注意: git stash不能将未被追踪的文件(untracked file)压栈,也就是从未被git add过的文件, 所以在git stash之前一定要用git status确认没有Untracked files


### 在stash中包含未跟踪的文件
$ git stash --include-untracked
 or
$ git stash -u

$ git stash -a //其中-a代表所有（追踪的&未追踪的）

# 通常在 dev 分支开发时,需要有紧急 bug 需要马上处理,保存现在修改的文件等,先修复 bug 后再回来继续工作的情况
```

> 为什么要用git stash

九、git rebase

> git rebase讲解

十、修改已经提交的commit 信息

```
修改最近一次提交的信息：
    $ git commit --amend
    类似 vim 修改，修改对应commit 信息，保存退出


修改以往n次提交的信息：

    $ git rebase -i HEAD~n (n是以往第n次的提交记录)

    类似vim 修改，将对应pick 改为 edit, 保存退出

    git commit --amend

    修改对应commit信息，保存退出

    git rebase --continue


如果不是最新版本，先pull 版本再修改，提交
    git pull
    按如上修改 commit信息

最新版本可直接 git push --force
```

十一、相关工具及网站

> Git教程-廖雪峰
>
> Git教程-易百教程
>
> Git官方文档
>
> git修改已经提交的commit信息
>
> git 教程 --git revert 命令
>
> 为什么要用git stash
>
> git rebase讲解
>
> GitHub提供的.gitignore配置文件
>
> 实际项目中如何使用Git做分支管理
>
> 对于所有分支而言，工作区和暂存区是公共的

十二、参与开源项目

基本步骤：

1. 将他人的开源仓库 fork 到自己的 Github 上；
2. 将该开源仓库从自己的 Github 上克隆到本地；git clone 项目地址
3. 修改该项目；
4. 推送一个 pull request 到他人开源仓库。（当然他人可选择接受或不接受）