



git

Git学习文档

一、Git简介

1.git是什么？

git是一种分布式 版本控制系统，由Linux使用C语言开发

2.什么是版本控制系统

举个例子：我们在管理文件时有时会对文件做一些修改更新，无论是自己或者别人来更新这个文件，最后都需要在某一个时间来查看这个更新之后的文档，那么具体怎么做呢？

以前的样子：



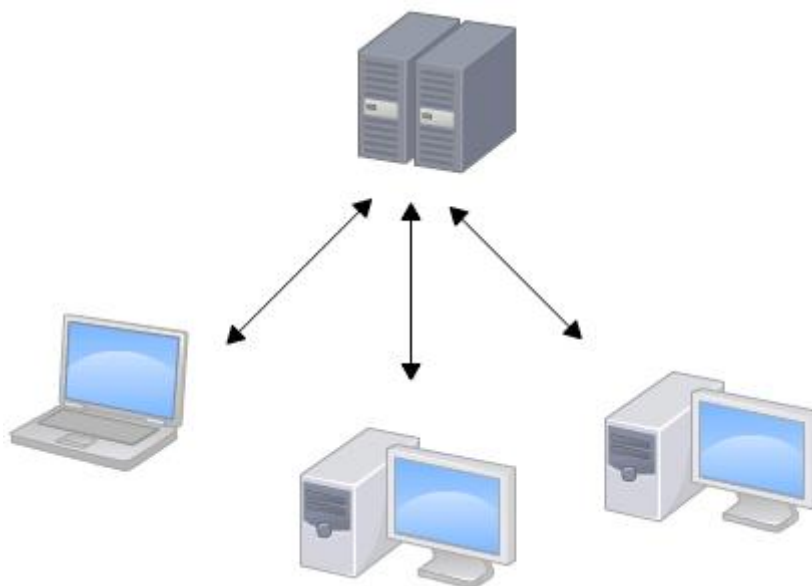
使用git之后样子：

版本	文件名	用户	说明	日期
1	demo.html	A	删除了xxx内容	5/20 08:20
2	demo.html	A	添加了xxx内容	5/20 15:14
3	demo.html & login.php	B	新增了login.php文件	6/1 20:05

版本	demo.html	用户	修改了内容	6/3/2003
	文件名		说明	日期

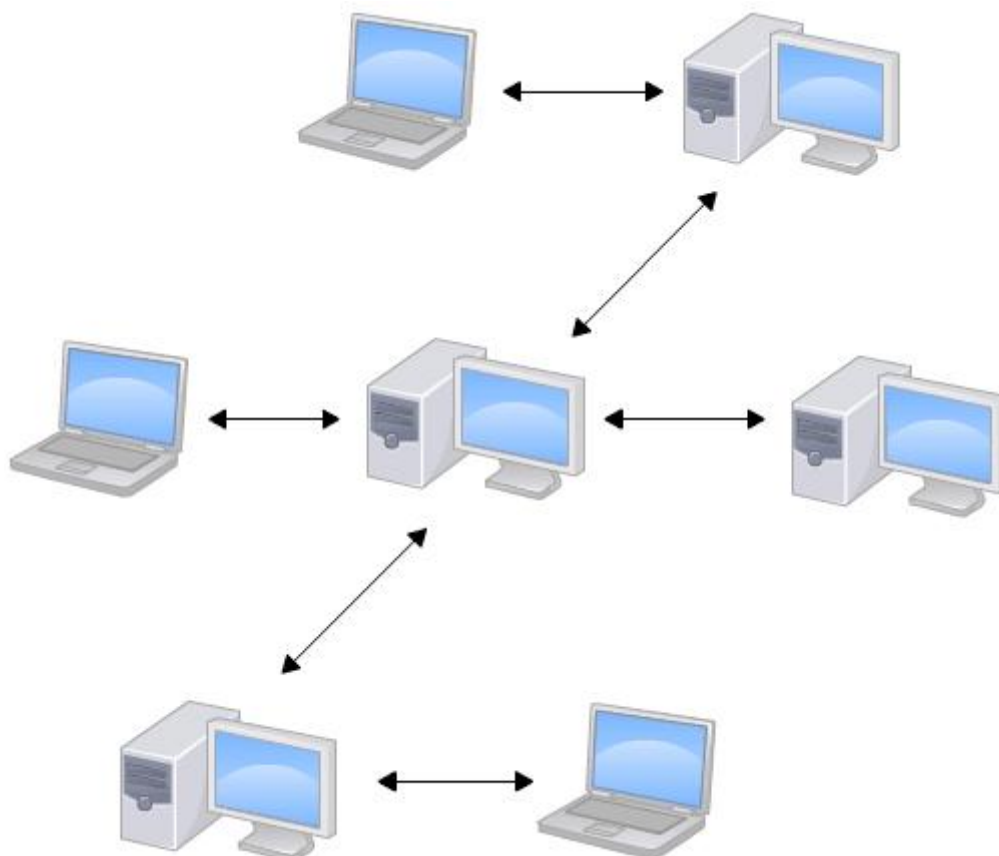
3.集中式 VS 分布式

- 集中式版本控制系统



定义：版本库是集中存放在中央式的版本控制系统，而工作的时候，都是用的自己的电脑。故要先从中央服务器取得文件的最新版本，工作完了再把文件推送给中央服务器。最大的特点就是**必须联网**才能工作，常见集中式版本控制系统：CVS、SVN等。

- 分布式版本控制系统



定义：分布式版本控制系统没有中央服务器，每个人的电脑就是一个完整的版本库，工作时就不需要联网，能够实现多人协作，即把修改的同一文件推送给对方完成修改整合，而且不用担心文件丢失损坏。在实际工作的时候通常也有一台充当“中央服务器”的电脑。Git就是常用的分布式版本控制系统，不仅不需要在联网条件下工作，还具备强大的分支管理。

4.创建版本库

1. 创建文件目录（有文件的目录也可以）
2. 将该目录变为Git可以管理的仓库（本地仓库）

```
1 $ git init
2 Initialized empty Git repository in F:/Git-Study/GitCourse-Liao Xuefeng
  version/learnGit/.git/
3
```

3. 添加文件：git add xxx.txt

注：可以添加多个文件

```
1 $ git add file1.txt
2 $ git add file2.txt file3.txt
3 $ git commit -m "add 3 files"
```

4. 使用 git commit 告诉Git，把文件提交到仓库

```
1 $ git commit -m "wrote a readme.txt"
```

注：-m 后面输入的是本次提交的说明，可以输入任意内容，但最好是见名知意的内容，方便以后自己或者别人查看

二、时光机穿梭

1.时光机穿梭

1. 当文件被修改之后

`git status` 命令可以时刻查看仓库当前状态，以下命令提示文件被修改但还没准备提交的修改

```
1 $ git status
2 On branch master
3 Changes not staged for commit:
4   (use "git add <file>..." to update what will be committed)
5   (use "git checkout -- <file>..." to discard changes in working
  directory)
6
7       modified:   readme.txt
8
9 no changes added to commit (use "git add" and/or "git commit -a")
10
```

2. 查看被修改的内容

`git diff`, diff 即 difference, 查看当前文件与之前文件的差异, ---为之前文件, +++为当前文件, 这样就能够清晰的看到修改的内容

```
1 $ git diff
2 diff --git a/readme.txt b/readme.txt
3 index 89253dd..be836b4 100644
4 --- a/readme.txt
5 +++ b/readme.txt
6 @@ -1,3 +1,2 @@
7 -Git is a version control system
8 -
9 -Git is free software
10 +Git is a distributed version control system
11 +Git is free software
12 \ No newline at end of file
13
```

3. 再次 `git add`, 查看状态

```
1 $ git add readme.txt // 无任何提示
2 $ git status
3 On branch master
4 Changes to be committed:
5   (use "git reset HEAD <file>..." to unstage)
6
7       modified:   readme.txt
8 // 表示将要被提交的修改包括 readme.txt, 可以进行下一步的提交
```

4. 再次 `git commit -m "xxx"`, 查看状态

```
1 $ git commit -m "add distributed"
2 [master 152646f] add distributed
3 1 file changed, 2 insertions(+), 3 deletions(-)
4
5 $ git status
6 On branch master
7 nothing to commit, working tree clean
8 //表示当前没有需要提交的修改, 而且工作目录是干净的 (working tree clean)
```

小结: 1.要随时掌握工作区的状态, `git status`

2.若 `git status` 命令说明有文件被修改过, 则用 `git diff` 查看修改的内容

3.重新对文件进行提交, `git add`、`git commit -m`

3.版本回退

注: 当我们在对某个文件进行多次修改时, 使用 `git commit` 命令后就会存在多个版本, 类似多个版本快照, 如果后面修改失误可以回退到最近修改的一个版本中重新进行修改

1. 查看文件版本: `git log`

表示显示由近至远的提交日志 `git log --pretty=oneline`: 在一行显示

```

1  $ git log
2  commit 5b1426c7c167ac336d77b44a45b890718d2afc72 (HEAD -> master)
3  Author: heiye-vn <1064239893@qq.com>
4  Date:   Sat Aug 17 10:03:44 2019 +0800
5
6      append GPL
7
8  commit 152646f7d64151a6f61803f4059739084a8915bf
9  Author: heiye-vn <1064239893@qq.com>
10 Date:   Sat Aug 17 09:39:30 2019 +0800
11
12     add distributed
13
14 commit aa375793744933d67e224bdf88afdc83628635b4
15 Author: heiye-vn <1064239893@qq.com>
16 Date:   Sat Aug 17 09:04:15 2019 +0800
17
18     wrote a readme file
19
20 $ git log --pretty=oneline
21 5b1426c7c167ac336d77b44a45b890718d2afc72 (HEAD -> master) append GPL
22 152646f7d64151a6f61803f4059739084a8915bf add distributed
23 aa375793744933d67e224bdf88afdc83628635b4 wrote a readme file
24
25 // 注: 前面的5b1...fc72为commit版本号, 这样不会导致版本冲突 (版本控制系统的特点)

```

2. 回退到上一个版本 (穿梭到过去), 并查看版本 `git reset --hard HEAD^`

注: 首先, Git必须知道当前版本是哪个版本, 在Git中, 用 `HEAD` 表示当前版本, 也就是最新的提交 `1094adb...` (注意我的提交ID和你的肯定不一样), 上一个版本就是 `HEAD^`, 上上一个版本就是 `HEAD^^`, 当然往上100个版本写100个 `^` 比较容易数不过来, 所以写成 `HEAD~100`。

```

1  $ git reset --hard HEAD^
2  HEAD is now at 152646f add distributed
3
4  ZSP@zsp-admin MINGW64 /f/Git-Study/GitCourse-Liao Xuefeng
   version/learnGit (master)
5  $ cat readme.txt
6  Git is a distributed version control system
7  Git is free software
8  ZSP@zsp-admin MINGW64 /f/Git-Study/GitCourse-Liao Xuefeng
   version/learnGit (master)
9  $ git log
10 commit 152646f7d64151a6f61803f4059739084a8915bf (HEAD -> master)
11 Author: heiye-vn <1064239893@qq.com>
12 Date:   Sat Aug 17 09:39:30 2019 +0800
13
14     add distributed
15
16 commit aa375793744933d67e224bdf88afdc83628635b4
17 Author: heiye-vn <1064239893@qq.com>
18 Date:   Sat Aug 17 09:04:15 2019 +0800
19
20     wrote a readme file
21 //此时 append GPL 版本消失不见了
22
23

```

3. 返回最新版本（穿梭到未来），`git reset --hard commit-id`

`commit-id` 不用输完，只需输入几位数（防止出现多个版本，尽量多写几位）就行，`git`会自动查找

```
1 $ git reset --hard 5b1426
2 HEAD is now at 5b1426c append GPL
3
4 ZSP@zsp-admin MINGW64 /f/Git-Study/GitCourse-Liao Xuefeng
  version/learnGit (master)
5 $ cat readme.txt
6 Git is a distributed version control system.
7 Git is free software distributed under the GPL.
8
```

注：`Git`的版本回退速度非常快，因为`Git`在内部有个指向当前版本的 `HEAD` 指针，当你回退版本的时候，`Git`仅仅是把`HEAD`从指向 `append GPL`：

```
1
2 [HEAD]
3
4 |
5 └─> ○ append GPL
6     |
7     ○ add distributed
8     |
9     ○ wrote a readme file
```

改为指向 `add distributed`：

```
1
2 [HEAD]
3
4 |
5 |   ○ append GPL
6 |   |
7 └─> ○ add distributed
8     |
9     ○ wrote a readme file
10
```

4. 版本 (commit) id记录: `git reflog`

```
1 $ git reflog
2 5b1426c (HEAD -> master) HEAD@{0}: reset: moving to 5b1426
3 152646f HEAD@{1}: reset: moving to 152646f
4 152646f HEAD@{2}: reset: moving to HEAD^
5 5b1426c (HEAD -> master) HEAD@{3}: reset: moving to HEAD
6 5b1426c (HEAD -> master) HEAD@{4}: commit: append GPL
7 152646f HEAD@{5}: commit: add distributed
8 aa37579 HEAD@{6}: commit (initial): wrote a readme file
```

小结：通过 `git reset --hard 版本号` 能够进行任意版本的回退，`git log` 查看提交历史，`git reflog` 能够记录提交的每个commit版本id，这样就能够真正做到随心所欲的回退到想要的版本

3.工作区和暂存区

Git 和其他版本控制系统如 SVN 的不同之处就是存在暂存区这一概念

1. 工作区 (Working Directory)

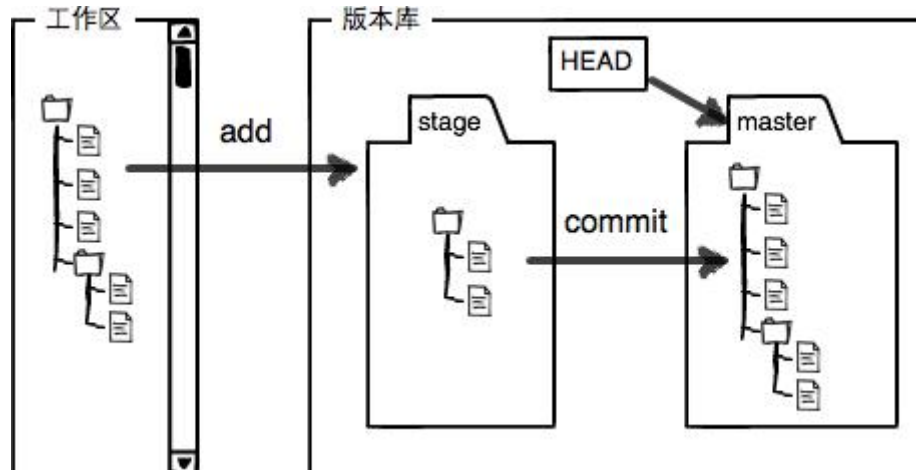
注：即在电脑中能看到的目录，如：learnGit就是一个工作区

> F-Disk (F:) > Git-Study > GitCourse-Liao Xuefeng version					
名称	修改日期	类型	大小		
images	2019/8/17 11:55	文件夹			
learnGit	2019/8/17 11:18	文件夹			
Notes.md	2019/8/17 11:54	Typora	11 KB		

2. 版本库 (Repository)

注：工作区中的 .git 目录不算在工作区中，而是Git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 (index) 的暂存区，还有Git自动创建的第一个分支master，以及指向master的一个指针 HEAD

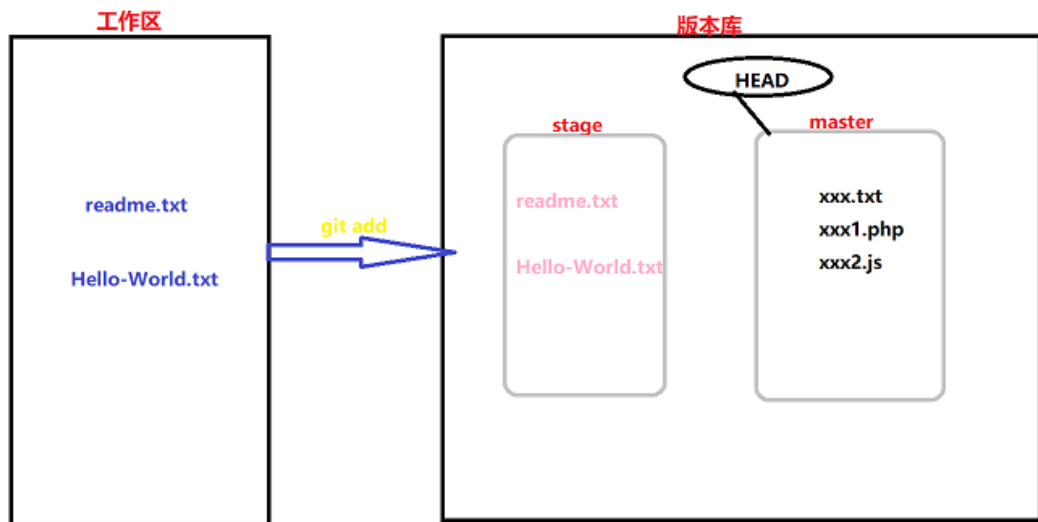


前面提到的文件提交中 `git add` 实际上就是把文件修改添加到暂存区，`git commit` 实际上就是把暂存区的所有内容提交到当前分支，即所有需要提交的文件开始都需要放到暂存区，然后一次性提交暂存区的所有修改到master分支上

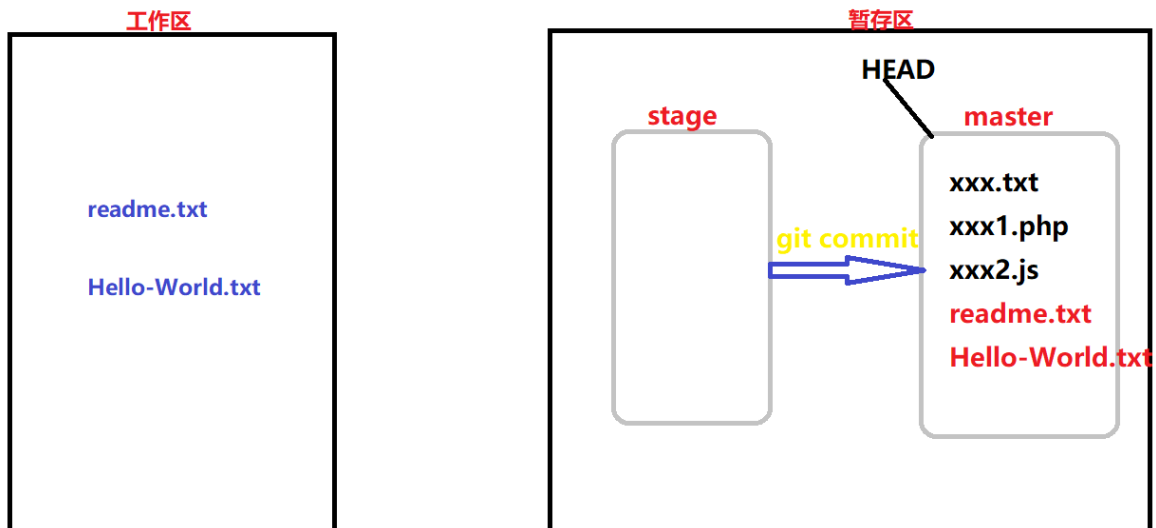
举栗子：修改readme.txt文件并新建一个Hello-World.txt文本，并添加到暂存区

```
1  $ git add readme.txt Hello-world.txt
2
3  ZSP@zsp-admin MINGW64 /f/Git-Study/GitCourse-Liao Xuefeng
  version/learnGit (master)
4  $ git status
5  On branch master
6  Changes to be committed:
7    (use "git reset HEAD <file>..." to unstage)
8
9      new file:   Hello-world.txt
10     modified:   readme.txt
11
```

此时暂存区的状态图示：



提交之后且未对工作区的文件再做修改，此时暂存区的状态：



总结：暂存区是Git中很重要的概念，了解的暂存区的工作原理，就知道了 git 中很多操作的具体作用

4.管理修改

Git 相较于其他版本控制系统不同的是：Git跟踪并管理的是修改而非文件，修改即为对文件进行增删改等操作

举个例子：对readme.txt进行修改 → 添加 `git add`，查看状态 `git status` → 再对readme.txt进行修改 → 提交 `git commit -m "xxx"`，查看状态 `git status`，出现问题（第二次修改没有被提交）

小结：提交修改有两种方式：

1. 第一次修改 → `git add` → `git commit` → 第二次修改 → `git add` → `git commit`
2. 第一次修改 → `git add` → 第二次修改 → `git add` → `git commit`

故：每次在修改文件之后一定要进行添加，否则不会被提交到暂存区

5.撤销修改

我们在对文件做修改时，有时需要撤销修改 `git checkout -- file`，恢复到最近提交的版本，注：-- 参数很重要，没有的话就变成了切换到另一个分支的命令。总之，撤销修改就是让文件回到最近一次 `git add` 或 `git commit` 时的状态，**撤销修改的条件是在工作区或者暂存区或者版本库，无法撤销推送到远程仓库 (GitHub) 的修改**

`git checkout -- readme.txt` 表示把 `readme.txt` 在工作区的修改全部撤销，存在两种情况：

1. `readme.txt` 修改后还未添加到暂存区，此时撤销就回到和版本库一模一样的状态

```
1 $ cat readme.txt
2 Git is a distributed version control system.
3 Git is free software distributed under the GPL.
4 Git has a mutable index called stage.
5 Git tracks changes.
6 Git tracks change of files.
7 My stupid boss still prefers SVN.
8 // 需要撤销第7行的修改
9 $ git checkout -- readme.txt
10
11 $ cat readme.txt
12 Git is a distributed version control system.
13 Git is free software distributed under the GPL.
14 Git has a mutable index called stage.
15 Git tracks changes.
16 Git tracks change of files.
```

2. `readme.txt` 已经添加 `git add` 到暂存区后，又做了修改，测试撤销修改就回到添加到暂存区后的状态

`git reset` 命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用 `HEAD` 时，表示最新的版本

```
1 $ cat readme.txt
2 Git is a distributed version control system.
3 Git is free software distributed under the GPL.
4 Git has a mutable index called stage.
5 Git tracks changes.
6 Git tracks change of files.
7 My stupid boss still prefers SVN.
8
9 $ git add readme.txt
10
11 $ git reset HEAD readme.txt
12 Unstaged changes after reset:
13 M      readme.txt
14
15 $ git checkout -- readme.txt
16
17 $ cat readme.txt
18 Git is a distributed version control system.
19 Git is free software distributed under the GPL.
20 Git has a mutable index called stage.
21 Git tracks changes
22 .
23 Git tracks change of files.
24
```

小结：撤销修改三种情况：

1. 当错误修改了**工作区**某个文件的内容，想要撤销修改时，使用 `git checkout -- file` 命令。
2. 当错误修改了**工作区**某个文件的内容，并且还提交到了**暂存区**，撤销修改：先使用 `git reset HEAD <file>` 命令，就回到了第一种情况，按照第一种情况操作。
3. 当某个文件的错误修改被添加且提交到了版本库时，撤销本次修改：参考版本回退（前提是没有推送到远程仓库）

6.删除文件

在 Git 中，删除文件也是一种修改

1. 新文件添加并提交到了版本库进行删除（确定要删除），

```
1  $ rm test.txt // 只是删除了工作区的文件，此时工作区和版本库就不一致
2
3  $ git rm test.txt // 使用 git命令删除
4  rm 'test.txt'
5
6  $ git commit -m "remove test.txt" // 再进行commit提交，就彻底将工作区和版本库的文件删除
7  [master 334cbda] remove test.txt
8  1 file changed, 1 deletion(-)
9  delete mode 100644 test.txt
10
```

2. 新文件添加并提交到了版本库进行删除（文件删错了，从版本库中恢复）

```
1  $ git checkout -- test.txt
2
3  $ ls
4  Hello-world.txt  readme.txt  test.txt
```

`git checkout` 是利用版本库里的版本替换工作区的版本，无论对工作区的文件进行修改还是删除都能够还原

三、远程仓库

1.概念及创建SSH key

1. 定义：Git 是分布式版本控制系统，同一个Git 仓库，可以分布到不同的机器上，即一台主机上一个原始版本库，其他主机可以“克隆”这个原始版本库，而且每个主机上的版本库都是一样的，我们可以自己搭建一台运行Git的服务器，或者使用GitHub提供远程仓库托管服务
2. 创建 SSH Key（密钥对）
 1. 在用户目录下查看是否有.ssh目录，再进入查看是否存在id_rsa 和 id_rsa.pub 两个文件，如果存在则忽略这一步，没有的话打开Shell（Windows下打开 Gti Bash），创建SSH Key

```
1  $ ssh-keygen -t rsa -C "邮箱地址"
```

2. 打开GitHub的setting设置界面，进入SSH and GPG keys选项，点击 Add SSH Key，设置任意title，把id_rsa.pub里面的文本粘贴到key文本框里，最后点击添加key

3. 其他远程仓库知识

1. 创建SSH Key的意义：因为GitHub需要识别出你推送的提交确实是本人推送的，并且Git支持SSH协议，所以GitHub只要知道你的公钥，就可以确认只有本人才能够推送
2. 创建多个SSH Key：GitHub允许添加多个key，考虑到了一个人可能会在不同的电脑上进行工作，只要把每台电脑的key添加到GitHub上，就能够向GitHub上推送
3. **公有仓库 (Public)** 和**私有仓库 (Private)**：在GitHub上创建远程仓库分为公有仓库和私有仓库，公有仓库即所有人都能够看到提交的内容（但只有自己才能修改），最好别把敏感信息加进去；私有仓库即别人查看不到提交的内容，创建私有仓库的方法：一是支付一定的费用来创建，二是自己搭建一个git服务器，只为自己提供服务（公司内部开发必备）

2.添加远程仓库

1. 在GitHub个人界面点击 + 里面的New repository来创建一个新的仓库

2. 本地仓库关联远程仓库

```
1 $ git remote add origin git@github.com:heiye-vn/learnGit.git
2 或者
3 $ git remote add origin https://github.com/heiye-vn/233.git
```

两种命令都能够关联远程仓库，只是方式不同：一种是HTTPS协议方式，另一种是SSH协议方式，其中origin参数表示为一个远程仓库

第三种关联方式是通过GitHub Desktop 桌面APP进行操作（了解）

3. 把本地仓库的内容推送到远程仓库

```
1 $ git push -u origin master
2 Enumerating objects: 23, done.
3 Counting objects: 100% (23/23), done.
4 Delta compression using up to 4 threads
5 Compressing objects: 100% (17/17), done.
6 Writing objects: 100% (23/23), 1.93 KiB | 58.00 KiB/s, done.
7 Total 23 (delta 4), reused 0 (delta 0)
8 remote: Resolving deltas: 100% (4/4), done.
9 To github.com:heiye-vn/learnGit.git
10 * [new branch]      master -> master
11 Branch 'master' set up to track remote branch 'master' from 'origin'.
```

注：把本地仓库内容推送到远程仓库命令：`git push`，实际上是把当前分支 `master` 推送到远程，因为是第一次推送 `master` 分支，故需要添加 `-u` 参数。Git不但会把本地的`master`推送到远程新的`master`分支，还会把二者关联起来，这样在后面进行推送或者拉取时就能够简化命令。**以后只要本地做了提交**，就能够通过命令 `git push origin master` 把本地`master`的最新修改推送到GitHub

4. SSH警告

当第一次使用 Git 的 `clone` 或者 `push` 命令连接GitHub时，会出现一个警告

```
1 The authenticity of host 'github.com (xx.xx.xx.xx)' can't be
  established.
2 RSA key fingerprint is xx.xx.xx.xx.xx.
3 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服务器，输入yes回车即可

小结：1.关联远程仓库：`git remote add origin git@server-name:path/repository-name.git` 或者 `git remote add origin https://server-name/path/repository-name.git`

2.关联后使用 `git push -u origin master` 第一次推送master分支的所有内容

3.以后，每次在本地提交后，只要有必要就可以使用 `git push origin master` 推送最新修改

3.从远程仓库克隆

一个项目在进行多人开发时，每个人可以从远程仓库里克隆一个原始版本，添加内容后再推送到远程仓库，这时就可以使用 `git clone` 命令进行版本克隆

```
1 $ git clone git@github.com:heiye-vn/gitskills.git
2 Cloning into 'gitskills'...
3 remote: Enumerating objects: 3, done.
4 remote: Counting objects: 100% (3/3), done.
5 remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
6 Receiving objects: 100% (3/3), done.
7
```

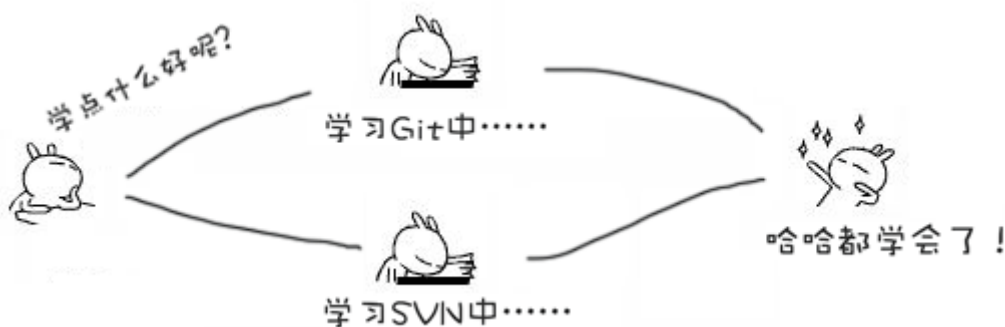
小结：1.要克隆一个仓库，首先必须知道仓库的地址，然后使用 `git clone` 命令克隆

2.Git支持多种协议，包括https，ssh等，但通过ssh支持的原生 git 协议速度最快

四、分支管理

1.概念

定义：什么是分支？举个栗子，平时在看一些科幻类的电影或者动漫中总会听到过平行宇宙（平行空间）这个名词，一个人在不同的空间里做着不同的事情，相互之间不会干预，分支就是如此，不过可能在某个时间点，两者会相遇，这样就造成了分支冲突

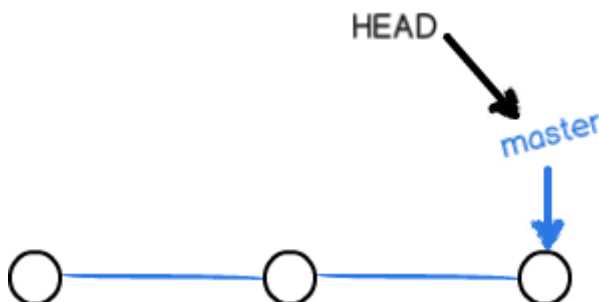


意义：比如在进行多人开发项目中，你的工作内容可能不会在短时间内完成，就不能提交到主分支上（影响别人工作），但是又必须要做内容提交，此时就可以创建一个属于自己的分支，可以进行任何操作，等工作内容完成之后再合并到主分支上，这样就给自己和别人带来很大的方便

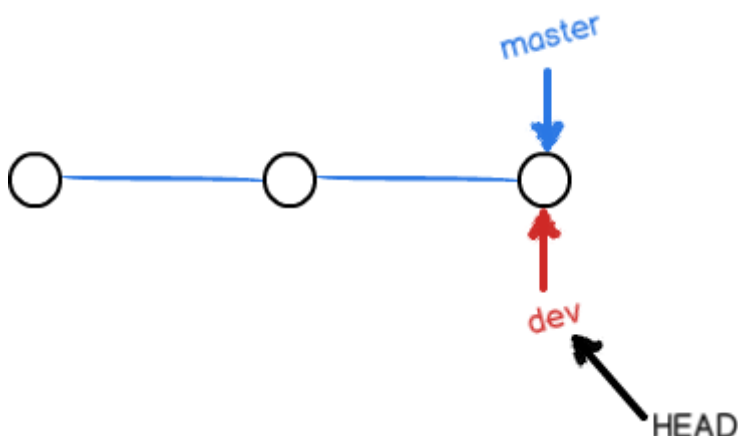
2.创建与合并分支

在**版本回退**小节里，我们知道，每次提交修改，git 都会把不同的版本串成一条时间线，这条时间线就是一个分支，这个分支在 Git 中叫做**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** 指向的是当前分支。

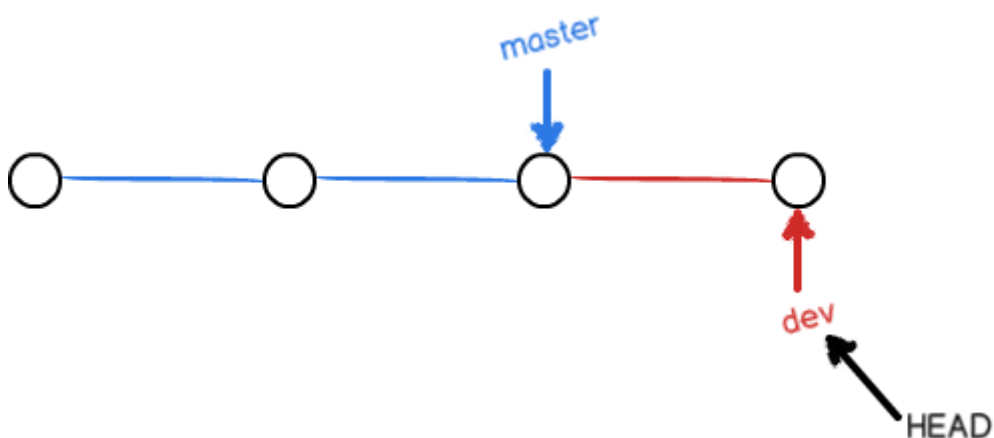
一开始，master分支是一条线，Git 用master指向最新的提交，再用 HEAD 指向 master，以及当前分支的提交点：



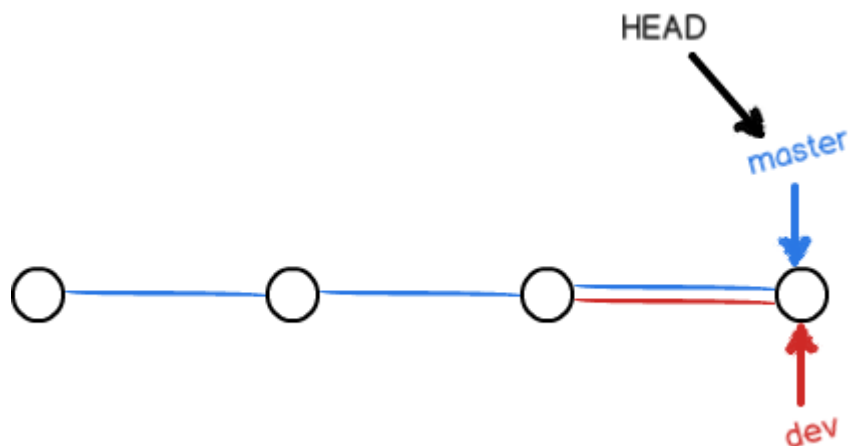
每次提交，master分支就会向前移动一步，这样，随着不断提交，master分支线也逐渐变长，下面创建一个新的分支 **dev**，指向 **master** 相同的提交，再把 **HEAD** 指向 **dev**，就表示当前分支在 **dev** 上：



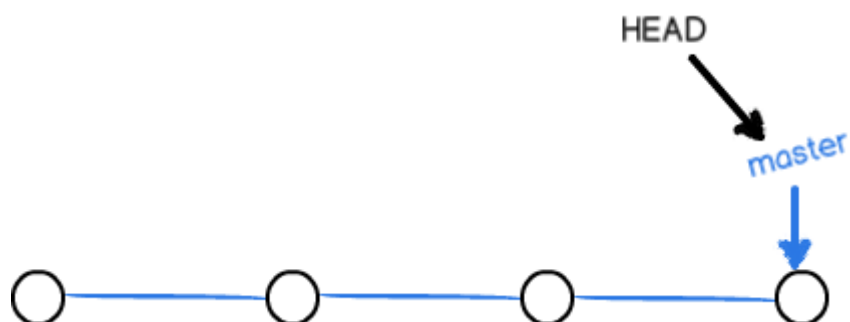
由此可见，Git创建一个分支很快，因为除了增加一个dev指针，改下HEAD的指向，工作区的文件没有发生任何，这样，对工作区的修改和提交就是针对 **dev** 分支了，如：新提交一次后，dev指针向前移动，而master指针不变：



接下来，如果我们在dev分支上的工作完成之后，就可以把 **dev** 合并到 **master** 上，怎么合并？直接把 **master** 指向 **dev** 的当前提交，从而完成合并：



最后，合并完之后可以根据需求删除 `dev` 分支，删除 `dev` 分支就是把 `dev` 指针给删掉，删掉后就保留了一条 `master` 主分支：



举栗子演示：

1. 创建分支并切换

```
git checkout -b dev
```

```
1 $ git checkout -b dev           // -b 参数表示创建并切换
2 Switched to a new branch 'dev'
3 // 相当于以下命令
4 $ git branch dev
5 $ git checkout dev
```

2. 使用 `git branch` 查看当前分支

```
1 $ git branch
2 * dev
3   master
4 // *表示当前分支
```

3. 可以在 `dev` 分支上进行正常提交

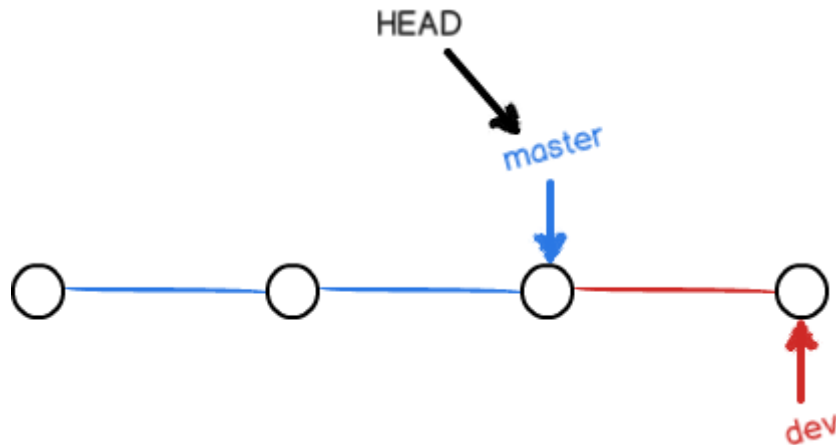
```
1 // 在README.md上添加内容并查看
2 $ echo Creating a new branch is quick >> README.md
3 $ cat README.md
4 # gitSkillsCreating a new branch is quick
5
6 // 然后添加并提交README.md文件
7 $ git add README.md
8 warning: LF will be replaced by CRLF in README.md.
9 The file will have its original line endings in your working directory
10 $ git commit -m "branch test"
```

```

11 [dev c2472e7] branch test
12 1 file changed, 1 insertion(+), 1 deletion(-)
13
14 // 再切换到master分支上并查看README.md内容
15 $ git checkout master
16 Switched to branch 'master'
17 Your branch is up to date with 'origin/master'.
18 $ cat README.md
19 # gitSkills

```

4. 可以发现切换到master分支后并没有README.md文件的修改内容，这是因为开始提交在了 dev 分支上，此时 master 分支的提交点并没有变：



5. 合并分支 `git merge dev`

```

1 // 合并分支并查看文件内容
2 $ git merge dev
3 Updating e369667..c2472e7
4 Fast-forward // 表示当前提交模式为“快进模式”，并不是每次合并都是
    在该模式下
5 README.md | 2 +-
6 1 file changed, 1 insertion(+), 1 deletion(-)
7 $ cat README.md
8 # gitSkillsCreating a new branch is quick
9

```

6. 删除分支 `git branch -d dev`

```

1 $ git branch -d dev
2 Deleted branch dev (was c2472e7).
3 $ git branch
4 * master

```

小结： 查看分支 `git branch` → 创建分支 `git branch name` → 切换分支 `git checkout name` → 创建+切换分支 `git checkout -b name` → 合并某分支到当前分支 `git merge name` → 删除分支 `git branch -d name`

Git 建议多使用分支来进行工作，合并之后再删除分支，效果和直接在 master 主分支上工作一样，但创建分支工作过程更安全稳定

3.解决冲突（重）

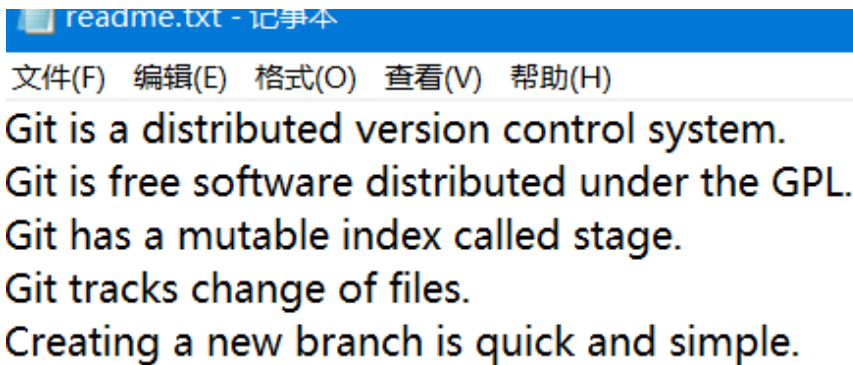
在工作中**分支冲突**的问题也是可能会出现，如何解决分支冲突对于我们来说是有必要知道的

实例：

1. 创建一个branch1分支并切换到该分支

```
1 | $ git checkout -b branch1
```

2. 在readme.txt文件中新增一行内容



readme.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

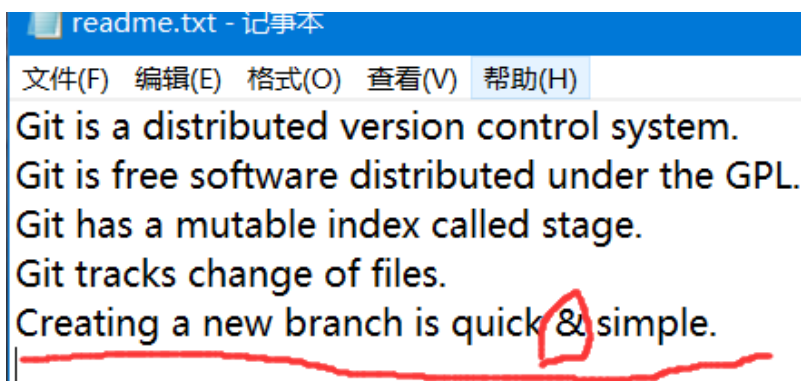
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks change of files.
Creating a new branch is quick and simple.

3. 在branch1分支上添加并提交修改

```
1 | $ git add readme.txt
2 | $ git commit -m "AND simple"
3 | [branch1 9e582c6] AND simple
4 | 1 file changed, 2 insertions(+), 1 deletion(-)
```

4. 切换到主分支master上，再修改readme.txt文本内容

```
1 | $ git checkout master
2 | Switched to branch 'master'
```



readme.txt - 记事本

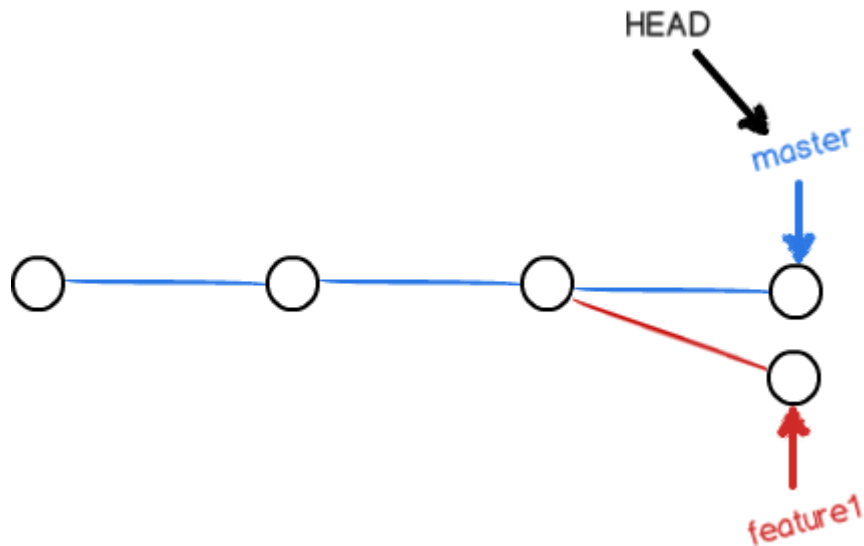
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks change of files.
Creating a new branch is quick & simple.

5. 在master分支上添加并提交修改

```
1 | $ git add readme.txt
2 | $ git commit -m "& simple"
3 | [master 2cf795e] & simple
4 | 1 file changed, 2 insertions(+), 1 deletion(-)
```

此时分支图如下：



6. 当合并branch1时，会出现**分支冲突**的报错

```

1  $ git merge branch1
2  Auto-merging readme.txt
3  CONFLICT (content): Merge conflict in readme.txt
4  // 冲突(内容):readme.txt中的合并冲突
5  Automatic merge failed; fix conflicts and then commit the result.
6  // 自动合并失败;修复冲突, 然后提交结果。
7
8  $ git status          // 状态显示也报错
9  On branch master
10 You have unmerged paths.
11   (fix conflicts and run "git commit")
12   (use "git merge --abort" to abort the merge)
13
14 Unmerged paths:
15   (use "git add <file>..." to mark resolution)
16
17       both modified:   readme.txt
18
19 no changes added to commit (use "git add" and/or "git commit -a")
20

```

此时readme.txt内容为:

```

1  $ cat readme.txt
2  Git is a distributed version control system.
3  Git is free software distributed under the GPL.
4  Git has a mutable index called stage.
5  Git tracks change of files.
6  <<<<<<< HEAD
7  Creating a new branch is quick & simple.
8  =====
9  Creating a new branch is quick AND simple.
10 >>>>>>> branch1

```

Git用 <<<<<<<, =====, >>>>>>> 标记出不同分支的内容

7. 手动修改readme.txt的内容，并在master分支上添加提交


```
1 $ git checkout -b dev
2 Switched to a new branch 'dev'
```

2. 修改readme.txt文件，并添加提交

```
1 $ git add readme.txt
2 $ git commit -m "add merge"
3 [dev 72446e9] add merge
4 1 file changed, 1 insertion(+)
```

3. 切换至主分支master，并合并分支，使用 no-ff参数

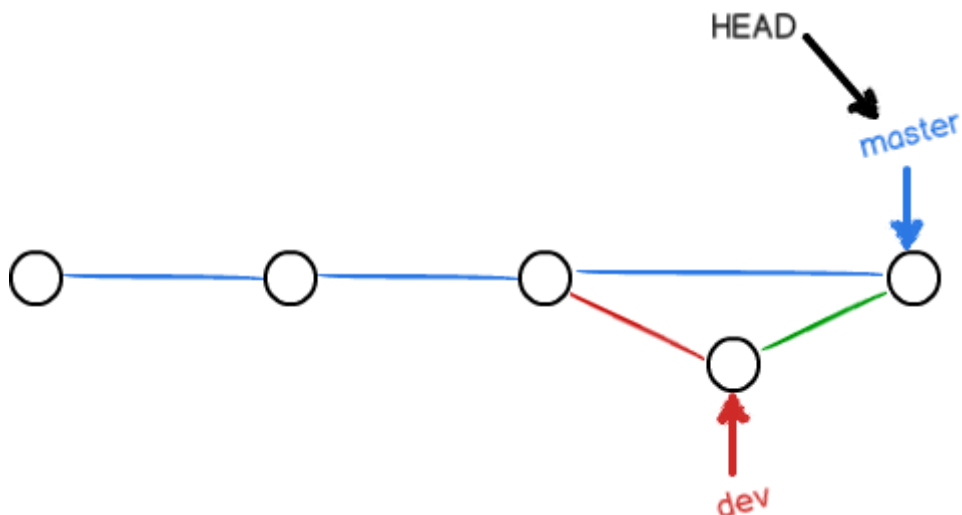
```
1 $ git checkout master
2 Switched to branch 'master'
3
4 $ git merge --no-ff -m "merge with no-ff" dev
5 Merge made by the 'recursive' strategy.
6  readme.txt | 1 +
7  1 file changed, 1 insertion(+)
```

因为使用了 `no-ff` 参数，故合并时生成了一个新的commit，所以加上 `-m` 参数，并添加提交描述

4. 查看分支历史

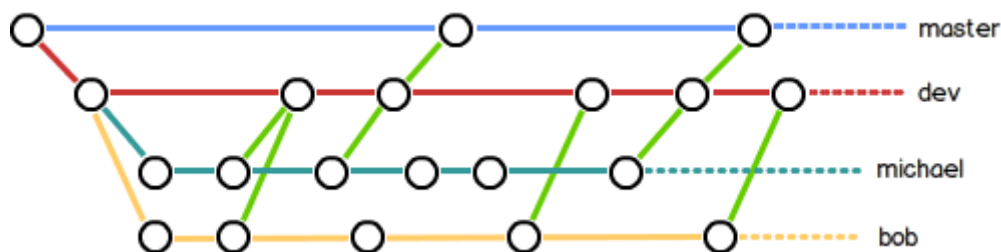
```
1 $ git log --graph --pretty=oneline --abbrev-commit
2 * 72546fc (HEAD -> master) merge with no-ff
3 | \
4 | * 72446e9 (dev) add merge
5 | /
6 * f1444b5 conflict fixed
7 | \
8 | * 9e582c6 AND simple
9 * | 2cf795e & simple
10 | /
11 * 19cdd8b branch-test
```

可以看到不使用 `Fast forward` 模式下的分支merge图：



5. 分支管理策略

1. `master` 主分支应该是非常稳定的，也就是仅用来发布新版本，不会在上面进行内容频繁提交（工作）
2. 工作时都是在 `dev` 上，`dev` 分支不稳定，完成工作需要提交某个版本时直接合并到 `master` 分支上，在 `master` 分支上发布这个版本
3. 开发成员都在 `dev` 分支上工作，每个人都有自己的分支，随时向 `dev` 分支合并就行了



小结：Git 分支十分强大，在团队开发中应该充分应用；合并分支时，加上 `--no-ff` 参数就可以使用普通模式合并，合并后的历史有分支，能显示出以前做过合并，而 `Fast forward` 模式下合并就无法看出

5. Bug分支

6. Feature分支

7. 多人协作

8. Rebase

五、标签管理

1. 概念

我们在发布一个版本时，通常要现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，以后在取某个标签的版本时，就是把那个打标签时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，（和分支类似，但是分支可以移动，tag不能移动），所以创建和删除标签都是快速完成的

引入标签的意义：前面说了，标签是指向某个commit的指针（二者绑定），由于每个commit的 id 很长且繁琐，使用标签来代替commit的 id 就可以快速找到需要的某个版本

2. 创建标签

1. 切换到需要打上标签的分支上
2. 使用 `git tag name` 打上一个新的标签，并查看所有标签

```
1 $ git tag v1.0
2 // 查看所有标签
3 $ git tag
4 v1.0
```

注：标签默认是打在最新提交的commit 上的，但有时候需要在以前的提交上打标签，就要通过commit的 id 来操作

3. 对历史提交的commit打上标签

```
1 $ git log --pretty=oneline --abbrev-commit
2 72546fc (HEAD -> master, tag: v1.0) merge with no-ff
3 72446e9 (dev) add merge
4 f1444b5 conflict fixed
5 2cf795e & simple
6 9e582c6 AND simple
7 19cdd8b branch-test
8
9 // 给 & simple commit打上标签
10 $ git tag v0.9 2cf795e
11
12 // 查看所有标签
13 $ git tag
14 v0.9
15 v1.0
```

注：标签列出顺序不是按照时间列出的，而是按照字母数字排列出的

4. 可以创建带说明的标签，`-a` 指定标签名，`-m` 添加说明文字，使用 `git show name` 查看说明信息

```
1 $ git tag -a v0.2 -m "version 0.2 released" 19cdd8b
2 // 查看标签信息
3 $ git show v0.2
4 tag v0.2
5 Tagger: heiye-vn <1064239893@qq.com>
6 Date: Tue Aug 20 12:00:08 2019 +0800
7
8 version 0.2 released
9
10 commit 19cdd8b762f9ffc329e66693d97a811eaeb02d34 (tag: v0.2)
11 Author: heiye-vn <1064239893@qq.com>
12 Date: Mon Aug 19 11:55:59 2019 +0800
13
14 branch-test
15
16 diff --git a/readme.txt b/readme.txt
```

注：因为标签和commit是相关联的，如果这个commit在多个分支上都有，那么同一个标签也会在多个分支上

小结：1.使用 `git tag name` 创建一个新的标签，默认为HEAD，也可以指定某一个commit（通过id查找）

2.使用 `git tag -a name -m "xxxxx" commit_id` 可以指定标签信息

3.使用 `git tag` 查看所有标签

4.使用 `git show name` 查看标签信息

3.操作标签

1. 可以删除错误的标签

```
1 $ git tag -d v0.1
2 Deleted tag 'v0.1' (was cc4ffb0)
```

2. 推送标签到远程仓库

因为创建的标签只存储在本地，不会自动推送到远程，所以错误的标签可以在本地安全删除，并且还可以推送到远程仓库

```
1 // 推送某一个标签
2 $ git push origin v1.0
3 Total 0 (delta 0), reused 0 (delta 0)
4 To github.com:michaelliao/learngit.git
5 * [new tag]          v1.0 -> v1.0
6 // 一次性推送所有标签
7 $ git push origin --tags
8 Total 0 (delta 0), reused 0 (delta 0)
9 To github.com:michaelliao/learngit.git
10 * [new tag]          v0.9 -> v0.9
```

3. 删除远程仓库中的标签

```
1 // 首先从本地删除
2 $ git tag -d v0.9
3 Deleted tag 'v0.9' (was f52c633)
4
5 // 再从远程删除
6 $ git push origin :refs/tags/v0.9
7 To github.com:michaelliao/learngit.git
8 - [deleted]          v0.9
9 // 可以在GitHub中验证是否删除
```

小结: 1.使用 `git push origin name` 可以推送一个本地标签到远程仓库

2.使用 `git push origin --tags` 可以推送所有未推送的本地标签到远程仓库

3.使用 `git tag -d name` 可以删除一个本地标签

4.使用 `git push origin :refs/tags/name` 可以删除远程仓库的标签（前提是先在本地仓库删除标签）

六、自定义Git

1.忽略特殊文件

2.配置别名

3.搭建Git服务器

七、Git Bash 常用操作文件命令

1.Git Bash 下操作文件及文件夹命令

- `cd`：切换到哪个目录下，如 `cd d:/xxx` 切换到 D 盘下的 xxx 目录，当我们用 `cd` 进入文件夹时，可以使用通配符*，`cd f*`，如果 E 盘下只有一个 f 开头的文件夹，就会进入到这个文件夹
- `cd ..`：回退到上一个目录，注：`cd` 和两个点之间有一个空格
- `pwd`：显示当前目录路径
- `ls (ll、dir)`：列出当前目录中的所有文件，只不过 `ll` 列出的内容更加详细
- `rm`：删除一个文件，如 `rm index.js` 就会把 `index.js` 文件删除
- `mkdir`：新建一个文件夹（目录），如 `mkdir src` 就会新建一个 `src` 目录
- `rm -r`：删除一个文件夹，如 `rm -r src` 就会把 `src` 文件删除
- `mv`：移动文件，`mv index.js src`，`index.js` 是要移动的文件，`src` 是目标文件夹，这样写必须要保证文件和目标文件在同一目录下
- `reset (clear)`：把 `git bash` 命令窗口中内容清空
- 写文件（window 中）：`echo xxx > readme.txt`（有内容的话会覆盖）、`echo xxx >> readme.txt`（追加到最后一行）
- `cat xxx.txt`：查看 `xxx.txt` 文件内容