Git完整教程

1. Git概述

1.1 为什么要了解Git

试想一下下面的场景，我们在生活中应该经常遇到类似让人马上起高血压的场景。我们做了很多版的方案，未来可能还要修改，可能改着改着发现还是之前的比较好。因此，我们对产生的所有方案都不敢删掉，自己又只能通过这种手动修改标题的方式做最原始的**版本管理**，最后亲手培养出这么一张“美妙绝伦”、“秒治高血压”的画面。

那么，**能不能让软件去帮我做版本管理呢？**不要再让我去体会这痛苦的一切~答案是肯定的。目前主流的版本管理主要有两个：SVN和Git。二者存在一些差异，但这里并不想要聊SVN，所以就不谈了，还是专注到git工具中来。

Git基本思想：从SVN到Git @虫虫

1.2 Git发展史

有一个很重量级的软件叫做 Linux(如果你了解过计算机，你肯定知道这个内核开源项目)，这个软件是有很多人合作来进行更新的。试想一下，如果你和很多人一起写一个故事，任何人都可以看到，并可以往里面写东西，这个故事还要在不同阶段发布不同版本的故事内容，这听起来就困难重重，如果我告诉你，你和这些人都不认识，也基本上没有见过，是不是感觉这个项目真的太难把控，往前推进了？

Linux在最初版发布后，很多人参与功能扩展、后续更新，也遇到了这个问题。当时，绝大多数的 Linux 的维护工作都花在了提交补丁和保存归档这些繁琐事务上（1991 - 2002 年间）。简单说，就是有一个专门负责审核、控制版本号的任来做这些事。

但是历经十年发展，这个项目已经非常非常庞大了，人工去管理也有着巨大的困难。于是，2002年起，他们开始用一个叫做BitKeeper的软件进行版本管理和维护。

可是，2005年，BitKeeper的开发公司不给免费使用了，如果参与者要交钱去做事(试想一下让你付费上班)，那参与的人肯定就跑光了。所以，Linux负责人、缔造者Linus Torvalds就下定决心自己从0开始做一个版本控制的软件，这样就不用担心再遇到这种情况了。

然后，Linus就花了两周时间，用C语言自己写了一个版本控制软件——Git。自此Git诞生，成功挽救了Linux这个重要的项目。而Git诞生后，大家发现它实在太好用了，而且免费、开源，于是逐渐扩散开来，成为如今软件开发中十分重要和关键的版本控制必备工具。

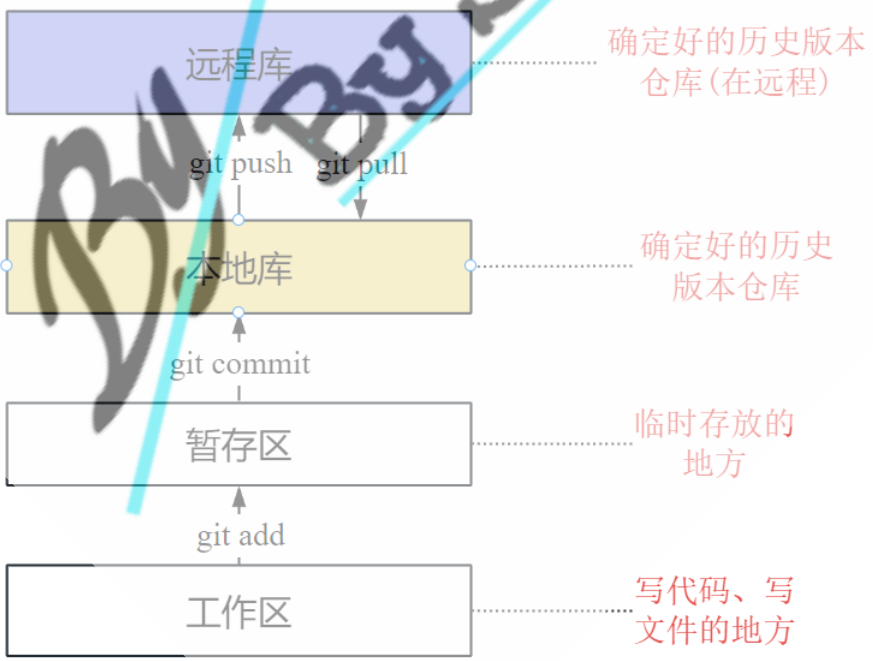
所以说，如果不是BitKeeper的开发公司想赚那点钱，我们现在可能就没有这么香的Git用了。让我们谢谢BitKeeper的开发公司——BitMover。

下面就是Linus，在计算机领域属于神一样的人物：

2. 基础知识

2.1 Git工作机制

下面这张图对于学习、使用Git是非常关键和重要的：



这里只介绍四个区域，对于箭头上面的命令，等后面进行介绍。我们这里以word文件为例：

- **工作区(Workspace)**：理解为你在word文件里面编写时候的区域。
- **暂存区(Index)**：理解为你word文件还没写完，但是到吃饭时间啦，所以先自动保存起来，下午接着写。

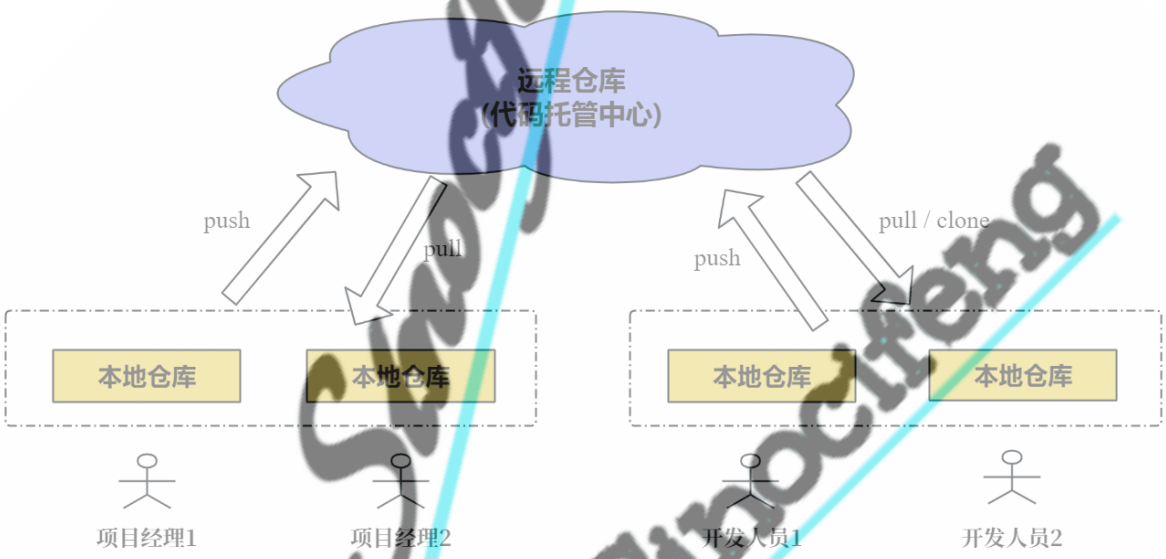
- **本地库(Repository)**：理解为你之前已经写好的版本，所以放到你项目文件夹下面啦。你现在还没写完、没给领导过目的新文件自然就不能纳入本地库里面。
- **远程库(Remote)**：将本地库里面的全存到远端的网盘（云）上面去，这样不仅你，公司其他人也能看到领导已经看过的文件啦。

当然，如果你的项目没有别人参与，你也不在乎是不是在别的电脑上也能看到，那么远程库就可以不用了。

对于写代码的计算机程序员来说，通常会使用Github、Gitlab等工具作为远程库(远程代码仓库)。这些远程代码仓库这里就不多介绍了。

2.2 工作场景梳理

我们在开始Git使用之前，先来梳理一下工作场景：



工作场景如上图，你(开发人员)把自己在电脑上写好的、打算给老板看的东西发到仓库上面，然后领导(项目经理)从仓库里面拿下来，审核没有问题就给你通过，确认这个是V1.1版。然后你就可以接着写你的V1.2了，但是在你提交V1.2之前，远程仓库里面是只有V1.0和V1.1版本的。如果小王此时加入团队，他就可以去仓库里面拿到最新版(V1.1)去做往下做了。

假如你做到V1.2发现做错了，你就可以去远程仓库把你想要的V1.0或者V1.1下载下来，重新写你的V1.2。

> PS：那么小王和你做的两个V1.2 有可能会有冲突。比如V1.1里面写的"玫瑰"，你提交的V1.2改成了"郁金香"，而小王没有改这块，那么小王提交时候就会产生冲突，这就需要小王去手动处理冲突了，这些这里先做个引子，不详细展开。

3. Git使用(命令)

3.1 Git安装

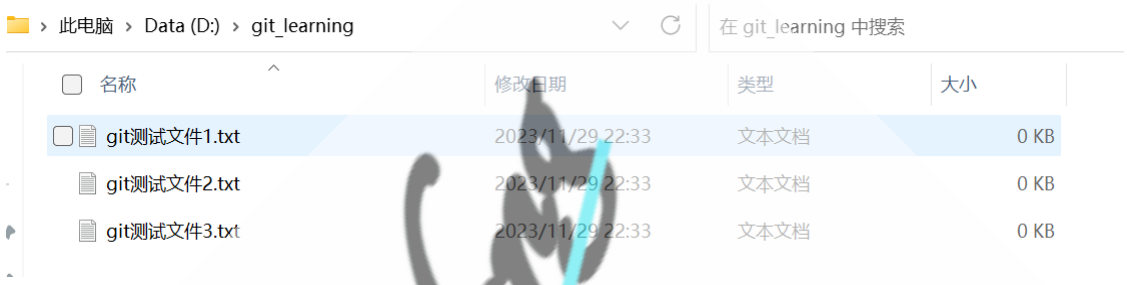
git软件的安装不是本文重点，这里不再阐述了。如果不会的，可以去网上找相关教程。

git的用户签名配置等，这里也不再阐述。需要注意：Git安装好后是必须设置用户签名的，不然是无法进行提交操作的。

3.2 git使用

这里就需要学习一些Git的语法了，这里会以图的形式进行详细阐述。

假如需要对下面的文件夹做版本管理，其中包含了三个文件：



打开工具的方式：在文件夹下点击鼠标右键，选择git bush here

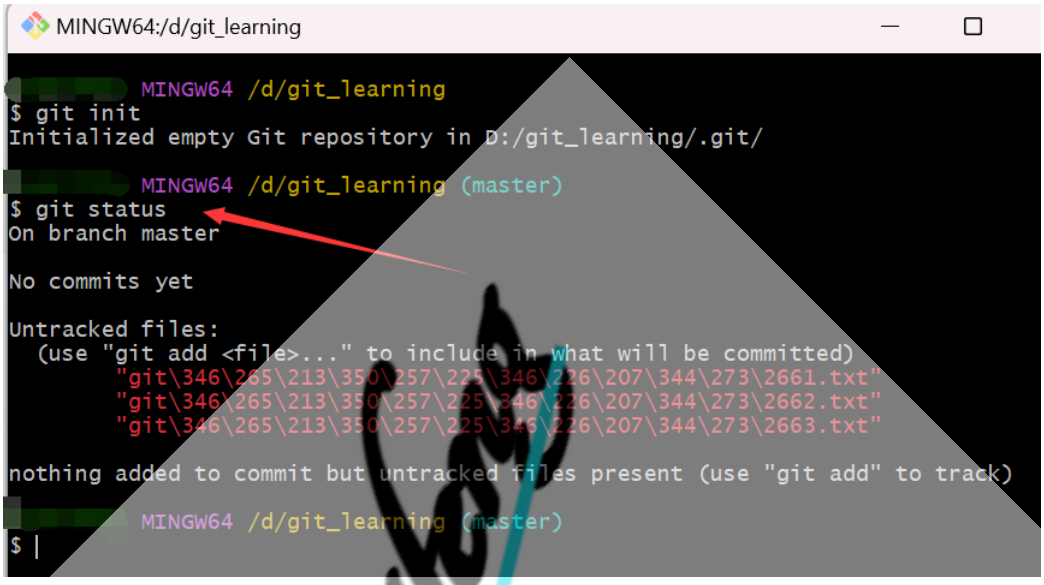
3.2.1 首先，需要进行本地库的初始化

简单理解就是让git去接手管理。

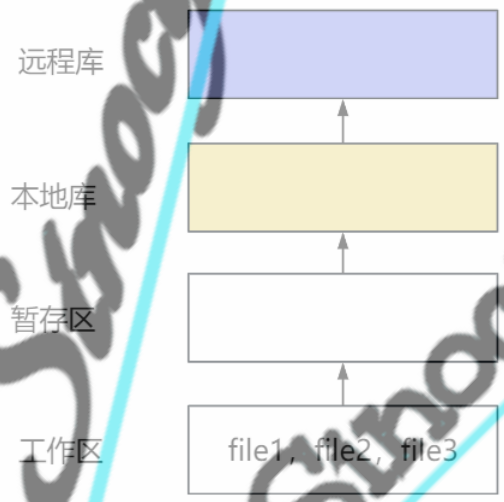
输入`git init`，然后回车即可。可以看到，文件夹下面多了一个名字为 .git 的新文件夹。如果看不到，是因为windows默认对于隐藏的项目(即名称以一个点开头的文件或文件夹)是不显示的。自己百度一下怎么打开就可以了。不打开也没关系，因为我们作为使用者完全不用理会这个文件夹。

3.2.2 下面，先看一下本地库的状态

输入`git status`，然后回车即可。可以看到下面信息说：在master分支，还没有提交过。没有追踪到的文件(对应3个文件)。



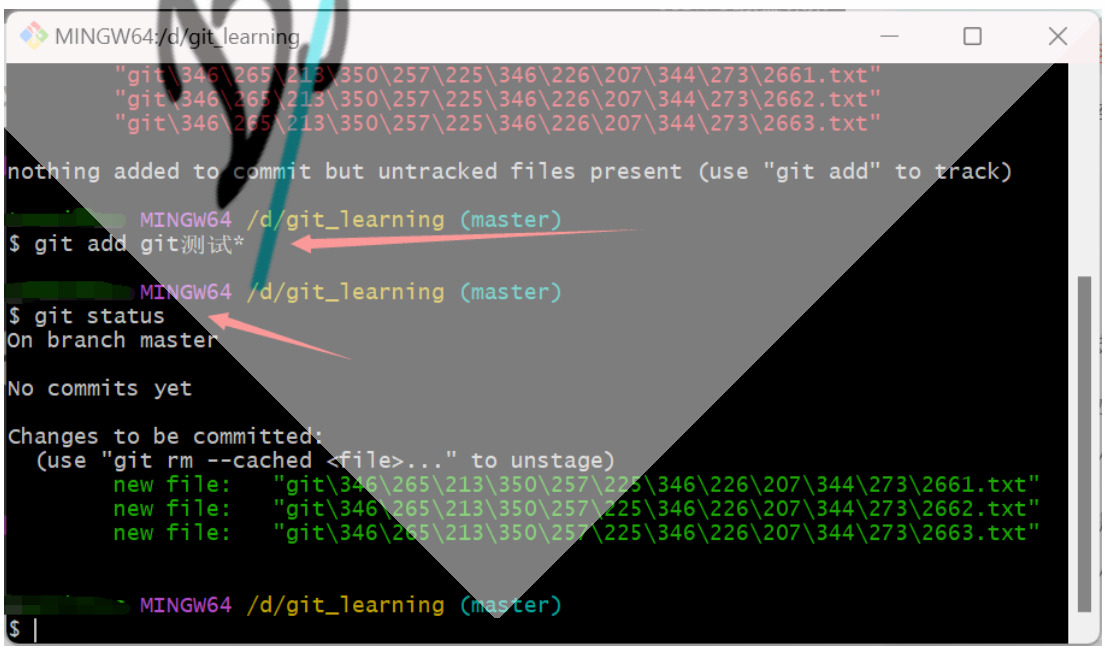
什么意思呢？简单来说如下图：



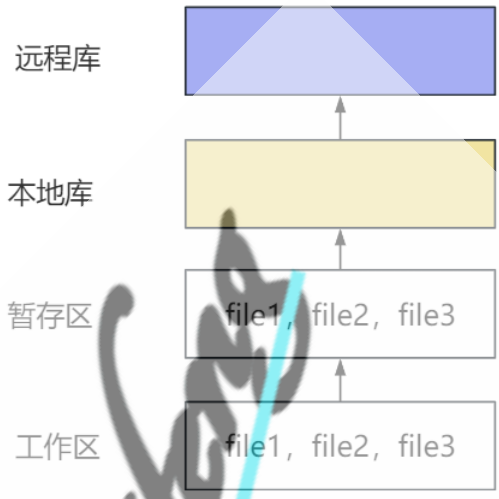
现在这三份文件都还位于工作区。本地仓库里面是空的。可以看到，要把工作区文件给到本地库，还需要先放到暂存区。

3.2.3 接下来，将文件放到暂存区

输入`git add 文件名`，然后回车即可。



添加完成后，输入git status可以看到原本红色的三个文件变成绿色了。此时对应于下图：

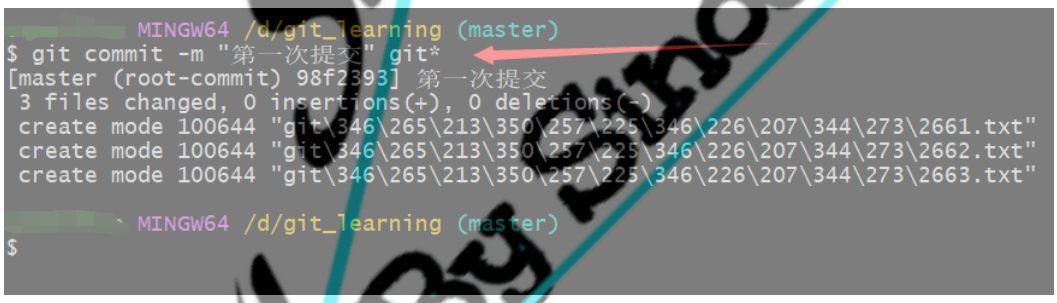


可以看到，文件被放入暂存区了。如果现在你去修改工作区文件，是不会对暂存区文件内容造成影响的。但是，听到暂存区这个名字你就知道，这只是一个暂时存放的，不是长久之计。所以，还需要将文件放入到本地库里面去。

3.2.4 然后，将暂存区文件放入本地库

输入`git commit -m "日志信息" 文件名`，然后回车即可。

"日志信息" 表示这里你可以写点补充信息，比如"第一次提交"之类的，它是归档时候的一个备注信息，方便你自己以后查看这个版本数据时候的一些提示信息。比如可以记录一下此次提交相比本地库中最新版本增加或删除了些什么。



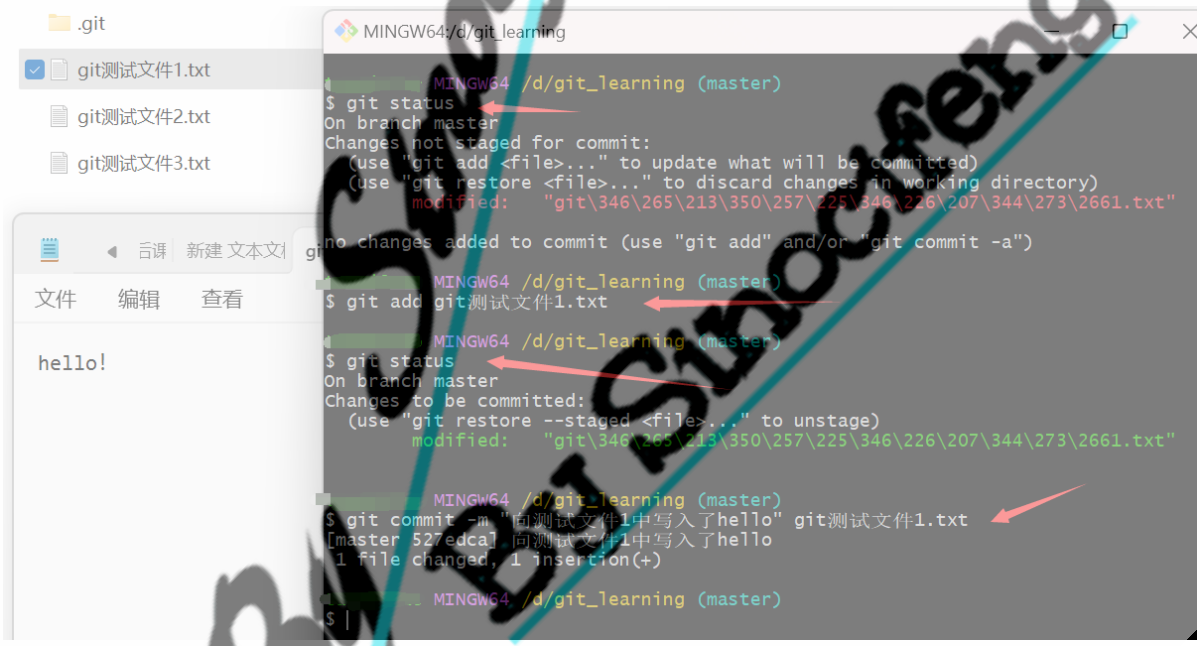
可以看到，提交成果了，此时如果输入git status可以看到暂存区为空了。现在没有需要提交的文件(绿色)，也没有和仓库不一致的文件(红色)。

此时对应于下图：

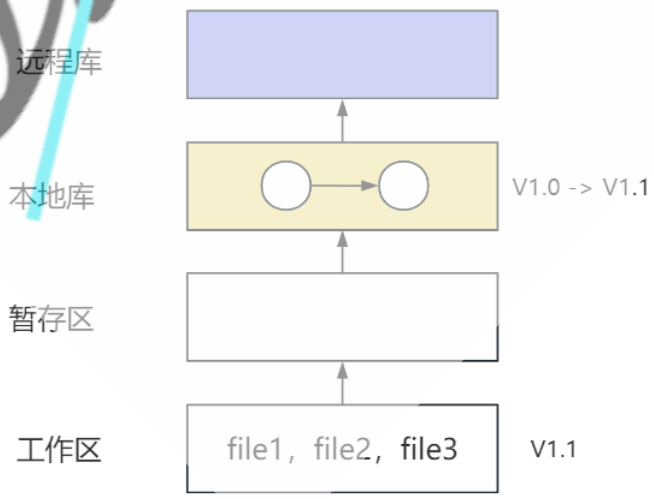
我们就已经实现了对1.0版本文件的提交。这样后续我们无论对工作区文件做什么操作，后续都可以从本地库找回提交时候V1.0版的三个文件啦。

插入部分

我们打开测试文件1，对其做一些修改。然后重复上面步骤，再进行一次提交工作。



现在对应于下图：



我们工作区现在是修改过的V1.1版文件了，但是在本地库里面，由于我们提交过两次，所以本地库里面有V1.0和V1.1两个版本的文件。这样，如果我们随时可以回退工作到V1.0去。

我们肯定不会只提交几次，那么怎么去查看历史提交版本内容呢?

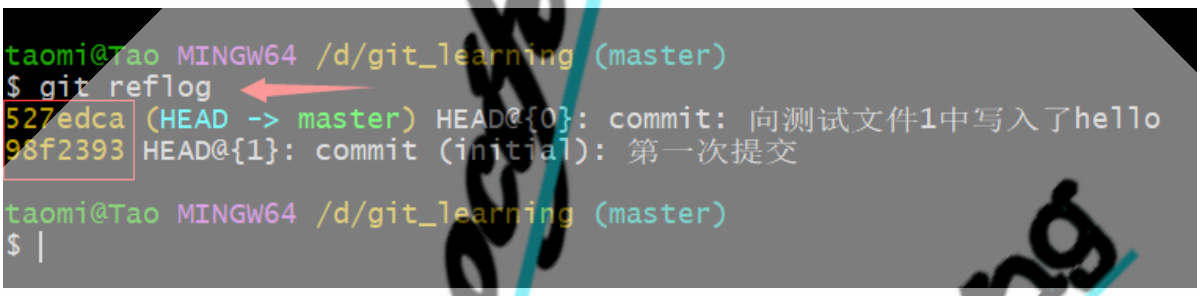
3.2.5 查看历史版本

这里有两个命令需要了解:

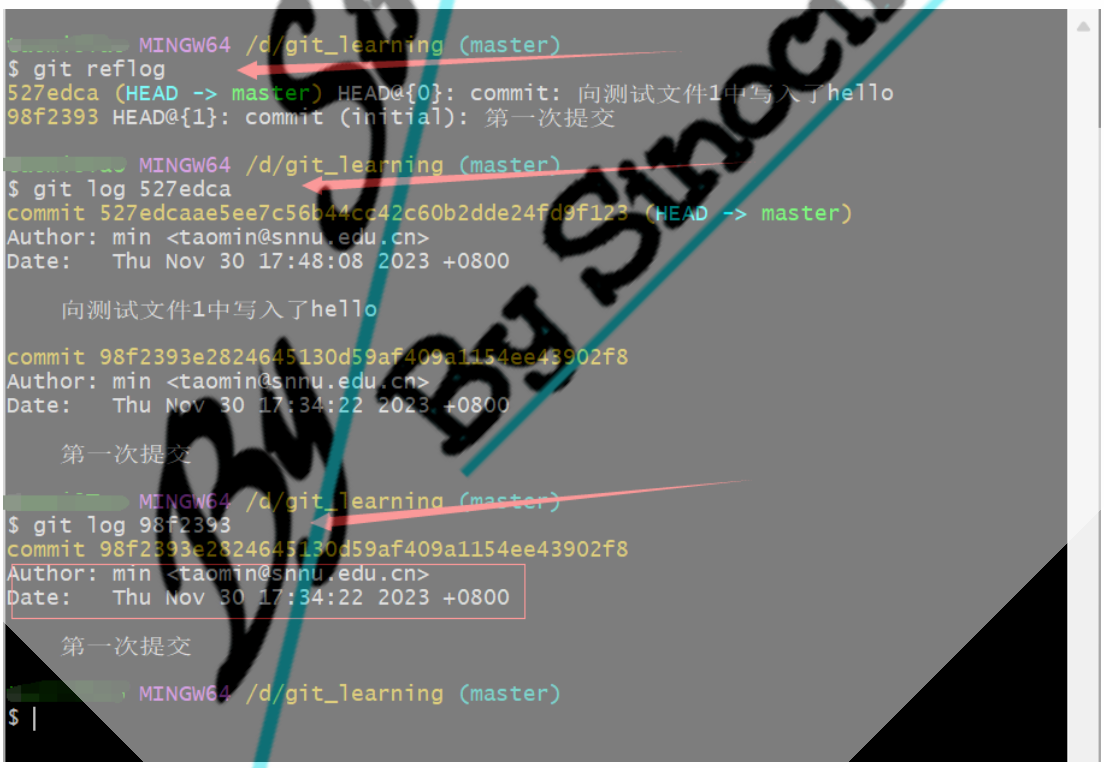
`git reflog` 查看版本信息

`git log` 查看版本详细信息

我们先来看一下git reflog命令，可以看到提示我们提交了两次，最上面的是最新一次提交。红色框是git自动给的版本号，我们可以根据版本号去查看对应版本的详细信息。



用git log和版本号，就可以看详细提交信息了。主要是谁提交的这个版本，什么时候提交的，以及提交时候写的一些日志信息。



3.2.6 版本回退

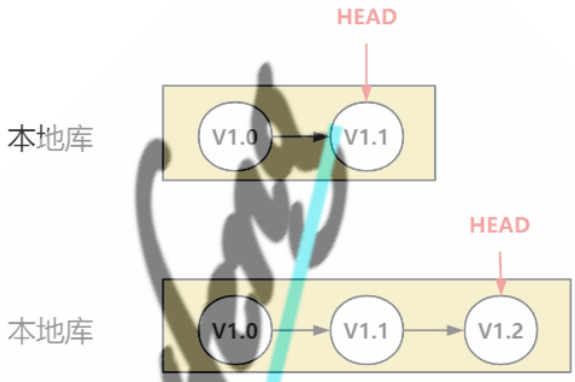
进行回退是需要使用前面3.2.5中版本号的。回退有以下回退方式:

git reset

- 软回退: `git reset --soft 版本号`，回退到某个版本，只回退了commit的信息。即：重置HEAD，保留暂存区和工作区，让仓库恢复到执行git commit之前的状态
- mixed回退: `git reset --mixed 版本号`，当我们写`git reset 版本号`时的默认回退方式。即：重置HEAD和暂存区，保留工作区

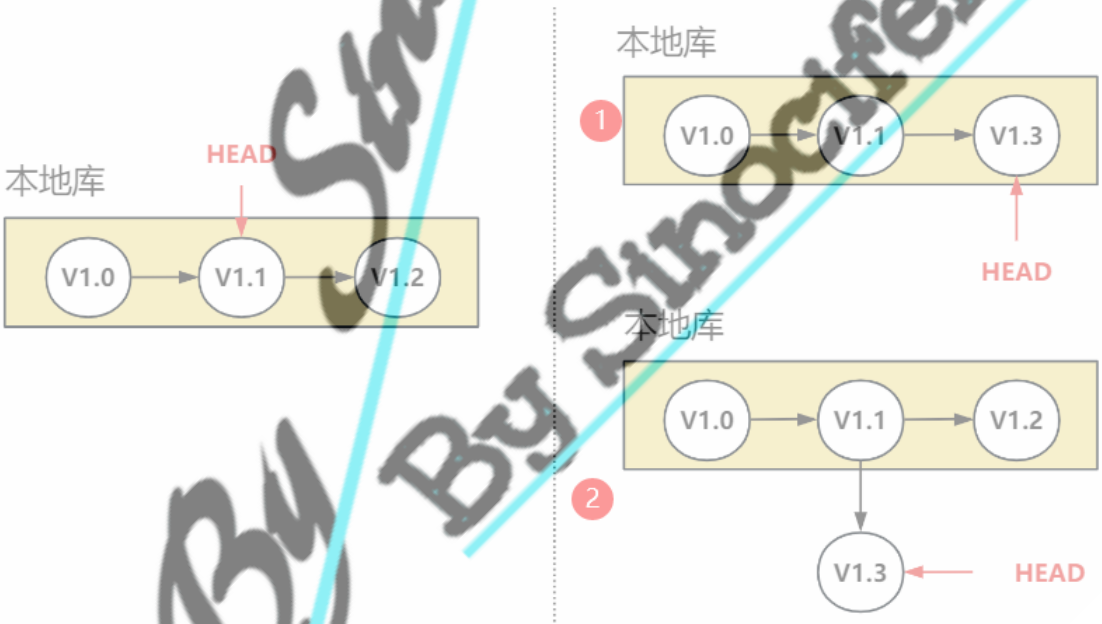
- 硬回退：`git reset --hard 版本号`，彻底回退到某个版本，本地的源码也会变为上一个版本的内容，撤销的commit中所包含的更改被冲掉。即：重置HEAD,暂存区和工作区

HEAD是一个特殊的指针，不了解的人可以理解成一个标记，是用来标示位置的。就像下图一样，我们每次提交之后，HEAD都会指向最新一次提交的版本记录。



可以看到，假如我们通过 git reset 回退V1.1版本，如果是硬回退，那么我们就等于彻底丢掉工作区的V1.2版本数据了。其他两种回退方式的话，我们还是可以通过一些方法找回V1.2版本数据的。所以，除非确认V1.2以后真的再也不用了，否则谨慎选择硬回退。

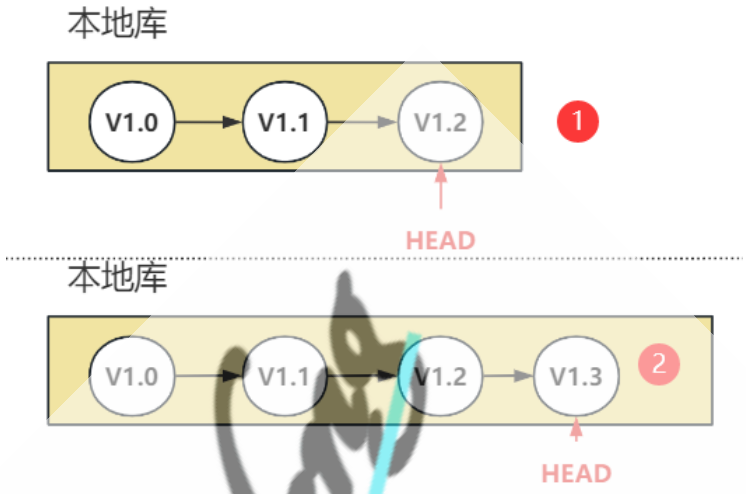
那么，回退完成后，如果我们再提交一次，会是下图中第一种结果还是第二种呢？



肯定是第二种，因为如果是第一种的话，只要提交后面的V1.2就会被干掉，那我们避开硬回退就没有意义了。但是第二种方式就带来一个问题：我们本来是一条链的结构现在开始分叉了。那么我们要怎么去在这个"树"结构上面找到任意一个节点（版本数据）呢？这就设计到关于git分支的学习了。因为分支的内容还有很多，在此只提一下先挖个矿，填坑的工作放到后面。

🌸git revert

git revert是用来"反做"某一个版本，以达到撤销该版本的修改的目的的。比如说，我们在仓库里面现在存在三个版本，如下图种第一部分一样。现在如果我们发现V1.1里面有问题，但是又不想影响到V1.2.那就可以用git revert来反做V1.1，产生一个V1.3。这个V1.3会保留V1.2里面的内容，但是会撤销V1.1的内容。



所以，`git revert`一般是要和`git commit`一起使用的。例如实现上面图中反做过程，可以如下：

```
git revert -n "V1.1"                                    # git revert -n "版本号"
(这里可能会出现冲突，那么需要手动修改冲突的文件。而且要git add 文件名)
git commit -m "revert V1.1 and add text.txt"            # git commit -m "提示信息"
```

实际上，我们常说的Git 切换版本， 底层其实就是移动 HEAD 指针。

4. 分支

在项目开发中，我们正常发布了1.0，2.0和3.0版本，并正在推进4.0的开发工作。但是此时突然发现，3.0版本中存在严重问题，需要尽快进行修复。如果我们将修复版本3.1加入主分支(Master分支，即默认的分支)，会导致4.0就算开发完成了也必须等3.1先合并(需要注意：我们这里3.1和4.0是为了便于理解起的版本号，不能认为4.0版本必须在3.1版本之后)，这会对主分支上4.0开发造成干扰。所以，应该在3.0版本处打一个新的分支，在3.1，3.2版本修复bug。当4.0开发完成后，就可以选择将3.2中已经修复的bug进行merge合并到主分支上去。

4.1 什么是分支

通过上面的例子可以看到，在版本控制中需要考虑多个任务同时推进的产经，所以Git采用的方式就是“分支”，通过为每个任务单独建立分支，这样参与具体项目者就可以把自己的工作从开发主线上分离开来，开发自己分支的时候， 不会影响主线分支的运行。

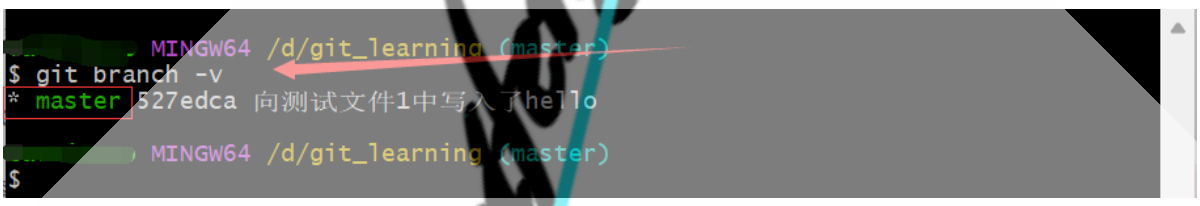
分支的好处就在于，可以同时并行地推进多个功能的开发工作，从而大大提升工作效率。在开发过程中， 如果某一个分支开发失败， 不会对其他分支有任何影响。失败的分支删除重新开始即可。

4.2 分支操作

```
git branch 分支名                                    # 创建分支
git branch -v                                        # 查看分支
git checkout 分支名                                  # 切换分支
git merge 分支名                                     # 把指定的分支合并到当前分
支上
```

4.2.1 分支查看

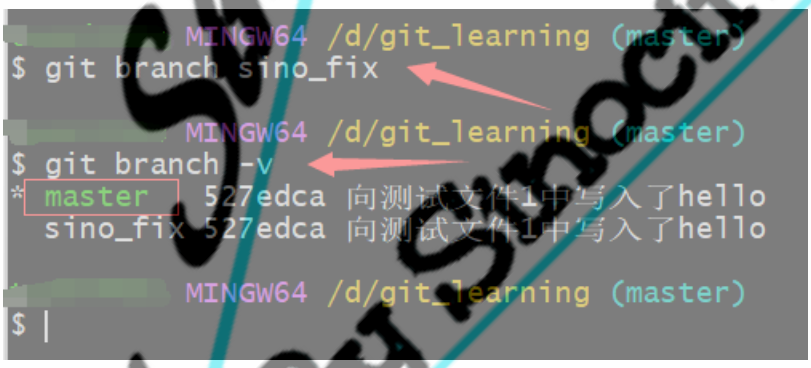
我们先来看一下目前有些什么分支，输入`git branch -v`然后回车即可：



可以看到，现在只有一个master分支。这是因为在之前我们刻意避开了分支操作，所以在这一过程中并没有创建新分支，当然只有一个默认的master分支了。

4.2.2 分支创建

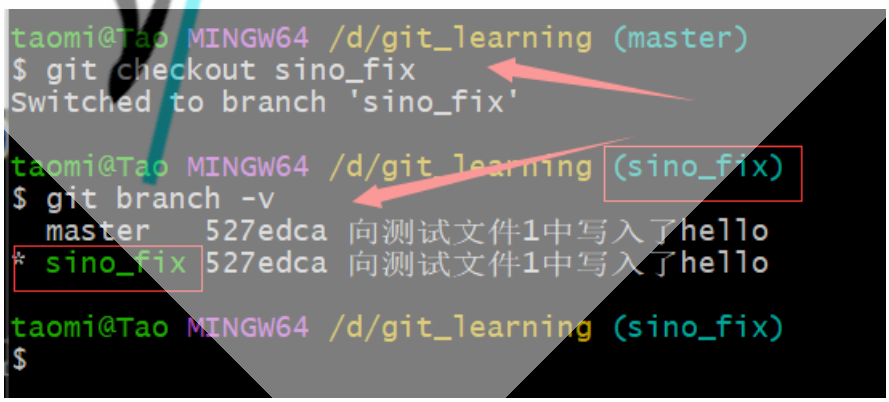
现在我们创建一个名为sino_fix的分支。输入命令`git branch sino_fix`回车即可



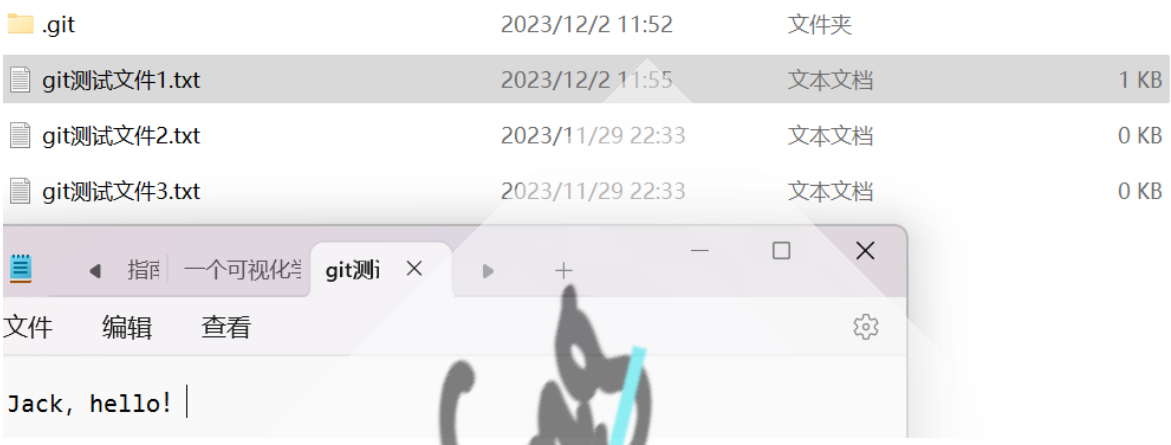
可以看到，我们成功创建了一个新的名为sino_fix的新分支，但现在我们还工作在master分支上。如果我们现在提交还是在master分支上进行提交，所以我们需要了解如何进行分支切换。

4.2.3 分支切换

输入命令`git checkout sino_fix`回车即可



可以看到，我们已经成功实现分支切换，从master分支切换到了sino_fix分支。如果现在我们进行提交，就是提交到sino_fix分支上了，不会对master分支造成影响。现在我们对工作区文件进行修改。本来文件中只有一个"Hello！"，现在我们添加了新数据进去，我们尝试把它进行提交。




```
MINGW64 /d/git_learning (sino_fix)
$ git status
On branch sino_fix
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   "git\346\265\213\350\257\225\346\226\207\344\273\2661.txt"

no changes added to commit (use "git add" and/or "git commit -a")

MINGW64 /d/git_learning (sino_fix)
$ git add git测试文件1.txt

MINGW64 /d/git_learning (sino_fix)
$ git status
On branch sino_fix
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   "git\346\265\213\350\257\225\346\226\207\344\273\2661.txt"


MINGW64 /d/git_learning (sino_fix)
$ git commit -m "sino_fix分支的第一次提交" git测试文件1.txt
[sino_fix 0eb000b] sino_fix分支的第一次提交
 1 file changed, 1 insertion(+), 1 deletion(-)

MINGW64 /d/git_learning (sino_fix)
$
```

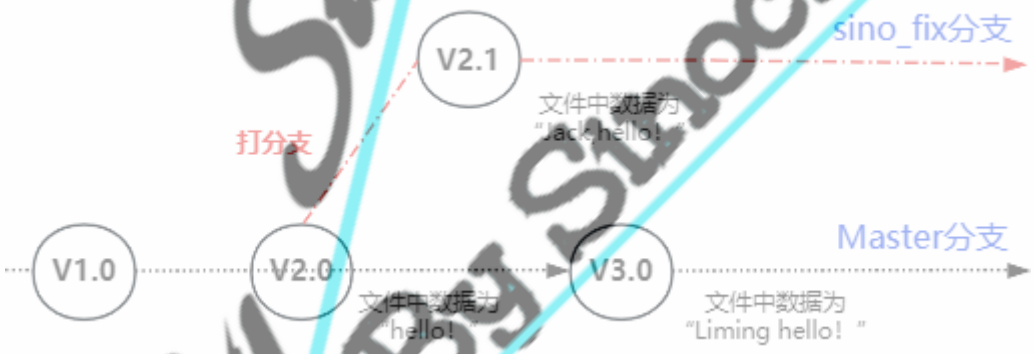

完成提交，现在就是提交到了sino_fix分支中去。现在我们的本地库中情况如下图，且我们目前工作在sino_fix分支上。

如果我们2.1已经修完V2.0中所有bug，则可以选择进行分支合并。为了后续演示方便，我们切回master分支，对文件中数据进行修改，从"hello!"改为"Liming hello!"。

> 或许你会疑惑，我们不是已经从"hello!"改为"Jack,hello!"了吗，为什么现在切换回master分支还是从"hello!"进行修改呢？这是因为"hello!"改为"Jack,hello!"只是在sino_fix分支上，当切换回master分支，我们master分支中最新数据仍然是"hello!"而不是"Jack,hello!"。

```
MINGW64 /d/git_learning (sino_fix)
$ git checkout master
Switched to branch 'master'

MINGW64 /d/git_learning (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   "git\346\265\213\350\257\225\346\226\207\344\273\2661.txt"

no changes added to commit (use "git add" and/or "git commit -a")

MINGW64 /d/git_learning (master)
$ git add git测试文件1.txt

MINGW64 /d/git_learning (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   "git\346\265\213\350\257\225\346\226\207\344\273\2661.txt"


MINGW64 /d/git_learning (master)
$ git commit -m "修改master分支中数据，进行提交" git测试文件1.txt
[master 5c324be] 修改master分支中数据，进行提交
 1 file changed, 1 insertion(+), 1 deletion(-)

MINGW64 /d/git_learning (master)
$
```

现在我们的本地库中情况如下图，且我们目前工作在master分支上。



4.2.4 分支合并

切换到要合并到的分支(一般来说是master分支)，然后执行`git merge 分支名称`即可进行分支合并。上面的工作中，我们已经处于master分支，所以不需要进行分支切换，直接执行合并即可。

```
taomi@Tao MINGW64 /d/git_learning (master)
$ git merge sino_fix
Auto-merging git测试文件1.txt
CONFLICT (content): Merge conflict in git测试文件1.txt
Automatic merge failed; fix conflicts and then commit the result.

taomi@Tao MINGW64 /d/git_learning (master|MERGING)
$
```

可以看到提示合并时候出现冲突了，这其实是我们在4.2.3后面引入的。我们2.1中的数据和3.0中数据是冲突的。合并时候，git就不知道哪个是对的，此时我们的文件变成了

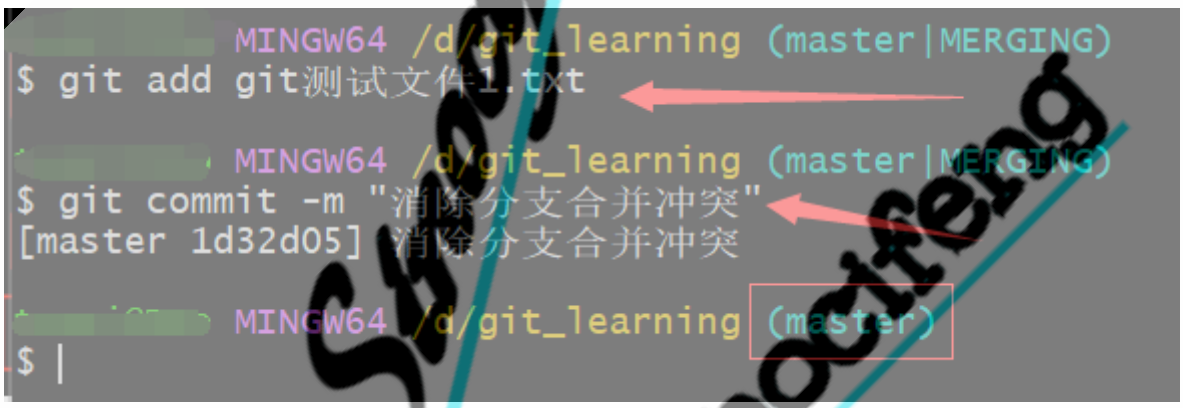
```
<<<<<<< HEAD
Liming hello!
=======
Jack，hello!
>>>>>>> sino_fix
```

**特殊符号：　<<<<<<< HEAD　当前分支的代码　=======合并过来的代码 >>>>>>> sino_fix

我们修改为：

```
Liming hello!
Jack，hello!
```

进行一次提交，先添加到暂存区，再进行commit。但是**需要注意**的是： 此时使用git commit 命令时不能带文件名！！！



现在我们的本地库中情况如下图，且我们目前工作在master分支上。

看到这里，你对于Git的基本知识就掌握了。但是分支本身十分复杂，我们如何定位到我们想要的分支上某个版本的提交数据，这个就比较复杂了，这里推荐一个可视化学习网站：https://learngitbranching.js.org/?locale=zh_CN 。正是因为分支这一设计，使得团队协作的同时进行版本管理变得简单起来，这也是git'这一工具的魅力所在。

5. Git团队协作

前面我们所讲的其实一直都在工作区、暂存区和本地库之间，并未涉及到远程库。实际上团队协作是需要一个所有人都可以访问的、内容唯一的远程库的，这也就是本节所要介绍的了，这个相对简单，放两张图就清楚了，不再进行展开。

5.1 团队内部协作

5.2 跨团队协作

6. 远程库介绍

作为个人用户，可以使用Github、Gitlab、Gitee等作为自己的远程库。配置过程网上有着许多教程，这里主要讲Git，就不多说了。

简单给出远程仓库常用操作命令：

命令名称	作用
git remote -v	查看当前所有远程地址别名
git remote add 别名 远程地址	起别名
git push 别名 分支	推送本地分支上的内容到远程仓库
git clone 远程地址	将远程仓库的内容克隆到本地
git pull 远程库地址别名 远程分支名	#将远程仓库对于分支最新内容拉下来后与当前本地分支直接合并 #

7. 后记

计算机知识的学习十分注重动手，建议学习过程中可以跟着案例进行练习加强理解。在写本文的过程中参考了许多前人资料，特在此表示感谢！

参考资料：

1. Git教程（超详细，一文秒懂）:https://blog.csdn.net/weixin_47824895/article/details/130169142
2. Git恢复之前版本的两种方法reset、revert（图文详解）: https://blog.csdn.net/yxlshk/article/details/79944535
3. Git的四个工作区域和主工作流程(workspace index repository remote）理解： https://blog.csdn.net/wl18271672781/article/details/126588155
4. Learn Git Branching：https://learngitbranching.js.org/?locale=zh_CN
5. git reset详解：https://blog.csdn.net/qq_39852676/article/details/129094985
6. git版本回退（git reset、git revert、git stash）：https://blog.csdn.net/qq_48617322/article/details/128631370