# Git版本控制系统

## Git版本控制系统的使用优点

①更方便的存储版本信息

②可以在各个版本之间进行切换

③更方便进行版本对比

④方便协同合作

## 原始的本地版本控制缺点

①本地会存在很多以前到现在一直维护的各个版本文件，附带日期和版本信息（使用Git版本控制系统后，本地管理的内容基本只有一份，不需要很注重命名方式，更新完提交即可，提交后会在版本控制系统里进行保存）

②回退版本会对目前在更新到一半的版本造成麻烦（使用Git版本控制系统后，所有更改过的信息都会记录在系统中，可以直接回退或者切换到最新的版本）

③对比新旧文件过于麻烦，只能打开两个文档逐行逐行比对（使用Git版本控制系统后，可生成对比信息，并且有明显标注）

④不方便于协同合作，难以得知当前其他成员对文档的最新更改

## Git的下载

下载地址：<https://git-scm.com/>

## Git的最小配置

如果在配置前进行操作，git会出现相应的警示提醒；

也是为了多人协作时，代码后面的维护可能需要找到相应的人。

### 1、查询git配置信息

|  |
| --- |
| git config --global --list |

### 2、配置名称和邮箱

|  |
| --- |
| git config --global user.name "..."  git config --global user.email "...@xxx.com" |

## Git的基础操作及指令

### 1、git --version版本查询

|  |
| --- |
| git --version |

### 2、git init建立Git仓库

①已经有文件时

|  |
| --- |
| cd target\_folder  git init |

②没有文件时

|  |
| --- |
| cd target\_folder  git init projectname |

### 3、git config配置信息

git config默认是--local，而local只对当前仓库有效，global则是对登录用户所有的仓库有效，system对系统所有用户都有效

|  |
| --- |
| git config --local //如果配置了则会有限找到local对应信息  git config --global //再然后找global信息  git config --system //最后再找system信息 |

### 4、git commit 提交版本

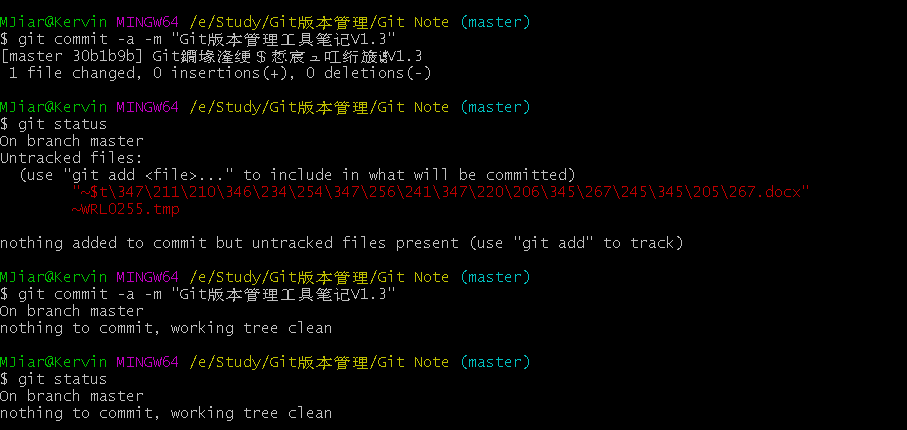
① git add 添加文件到缓存

② git commit 提交版本

|  |
| --- |
| git add file //把修改过得文件提交到临时缓存区，还不是正式版本  git add . //如果文件太多，想提交当前文件夹下的所有，可执行  git commit -m "版本信息" //以填写的“版本信息”作为提交的日志被上传 |

与之等价的指令，直接跳过缓存区直接提交

|  |
| --- |
| git commit -a -m "版本信息" |

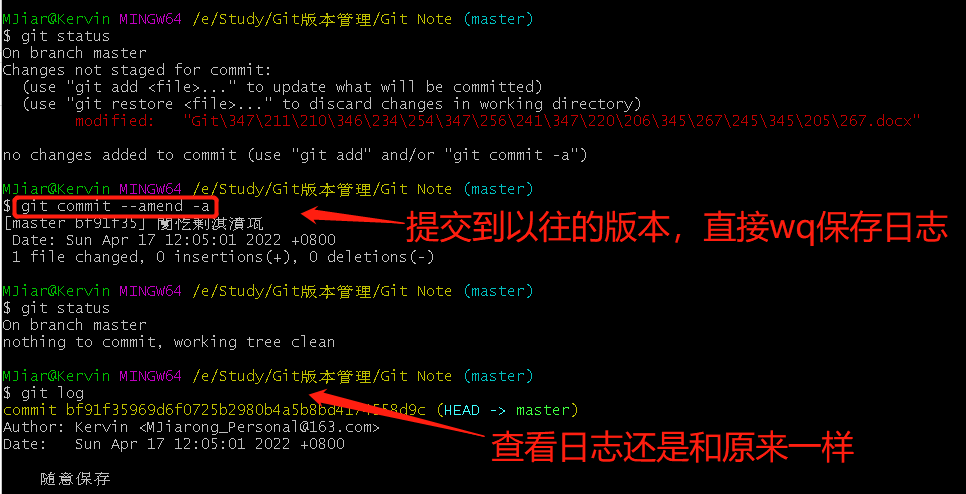


### 5、git commit --amend 版本补录

针对以往的版本提交有遗漏的情况，比如有些代码，或者文件是验证通过的。只是以前忘记提交的。或者用户不想重新建立一个新的版本并提交时，想将其提交到之前已成功提交的版本上。

执行指令：

|  |
| --- |
| git commit --amend |



### 6、git log查看当前项目的信息

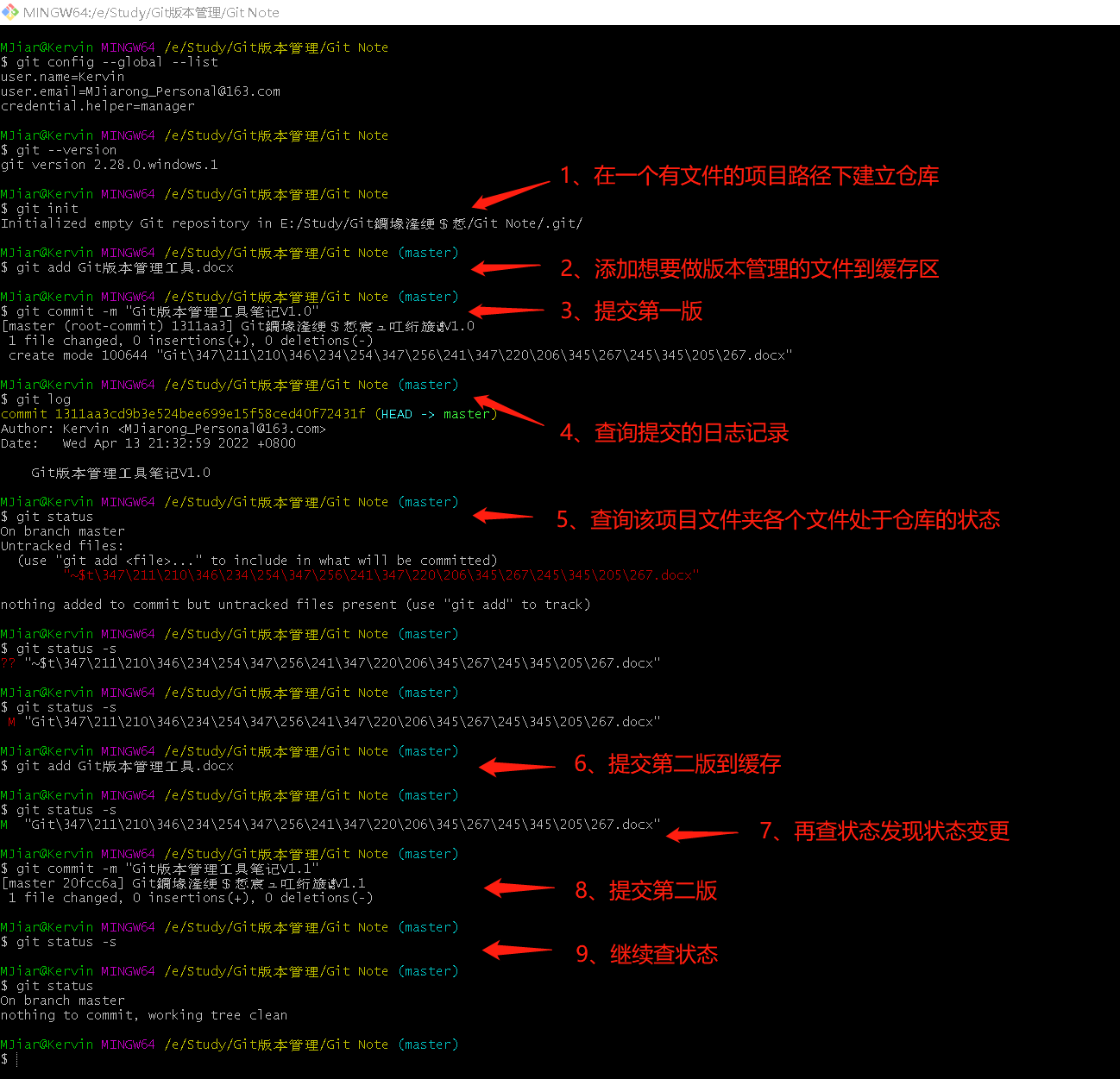
|  |
| --- |
| git log |

### 7、git status版本状态查看

当提交完一个项目到git，都会进行版本状态的查看，看看文件修改成什么样，在版本控制系统中，处于一个什么状态（比如：目前处于暂存区间？只是刚好处于修改状态？或者已经完全被保存到版本控制系统）

|  |
| --- |
| git status  git status -s  git status --short |

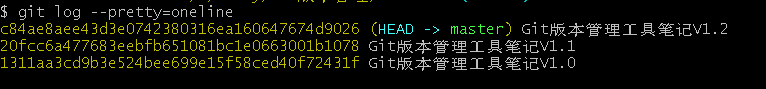
示例：



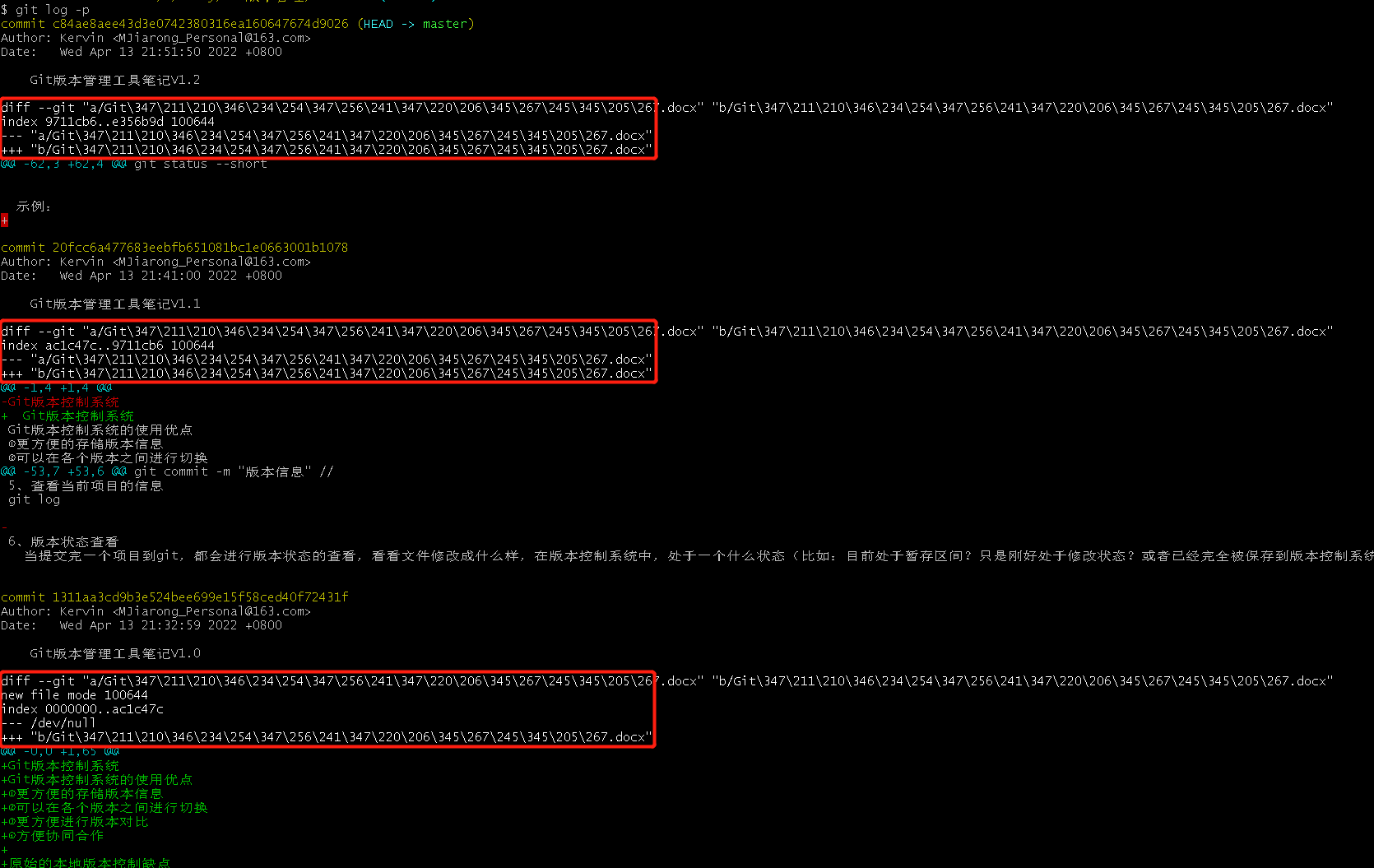
### 8、git log查看历史版本

|  |
| --- |
| git log //最简单的使用  git log –pretty=oneline //只显示一行  git log -p //显示版本之间的差别  git log -5 //显示最近5个版本  git log -5 -p //显示最近5个版本之间的差别  git log –stat //查看大体的统计信息 |

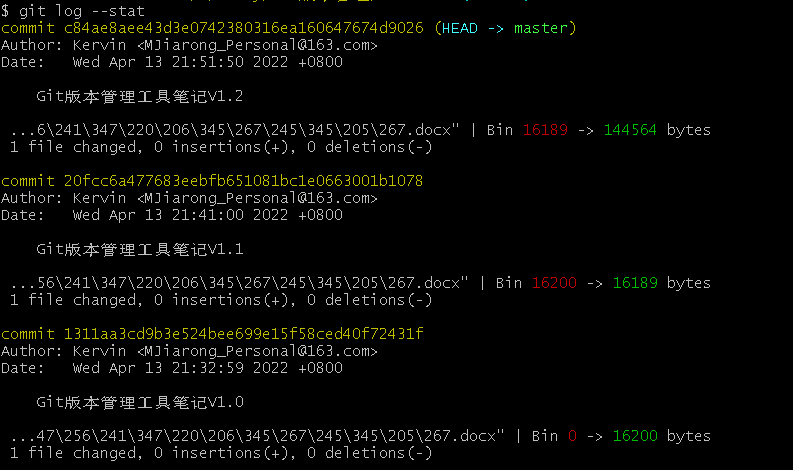
示例 git log –pretty=oneline：**（简短显示了各个版本不同日志）**



示例 git log -p：**（统计了各个版本不同的内容）**



示例 git log --stat：**（统计了各个版本不同的内容、字节大小等）**



### 9、.gitignore 忽略文件

文件有着全局global匹配特性中的编写规则：

#（跟）注释，

\*（匹配任意字符）如：\*.o（忽略所有编译输出的对象（可执行）文件）

[abc]，可以是括号内任何一个字符，如：LOG[IDE].txt

[0-9]，匹配0-9的字符，如：LOG[0-6].txt

? 匹配任意一个字符

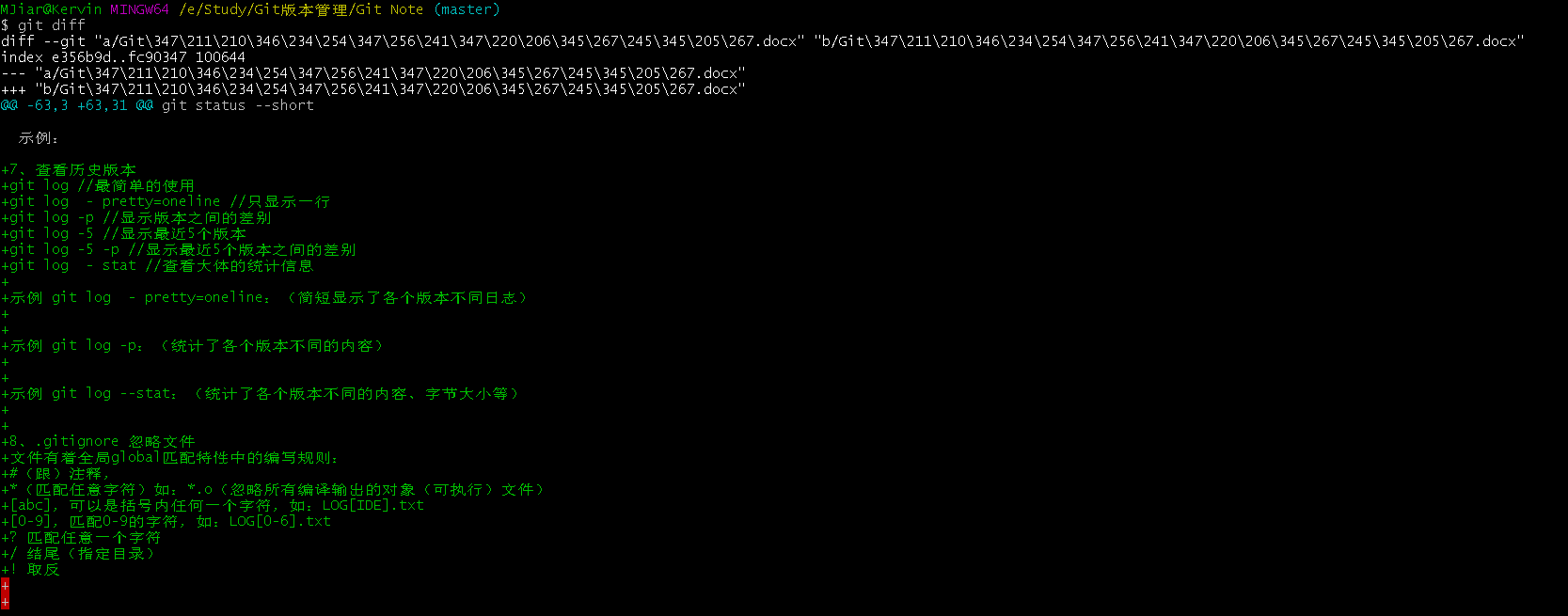
/ 结尾（指定目录）

! 取反

### 10、git diff 版本比对（输出版本差异信息）

**比对当前文件和最近一次提交的更改地方**

|  |
| --- |
| git diff |



比对指定版本（旧-新）

|  |
| --- |
| git diff 以往版本的哈希值 现在版本的分支名（或哈希值） |



也可以比对（新-旧）版本



对于提交到缓存区间的版本，想要查看差异信息，可以添加 --staged 参数

|  |
| --- |
| git diff --staged |



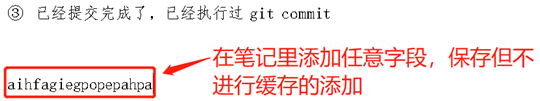
### 11、git restore 恢复文件内容（返回变更）

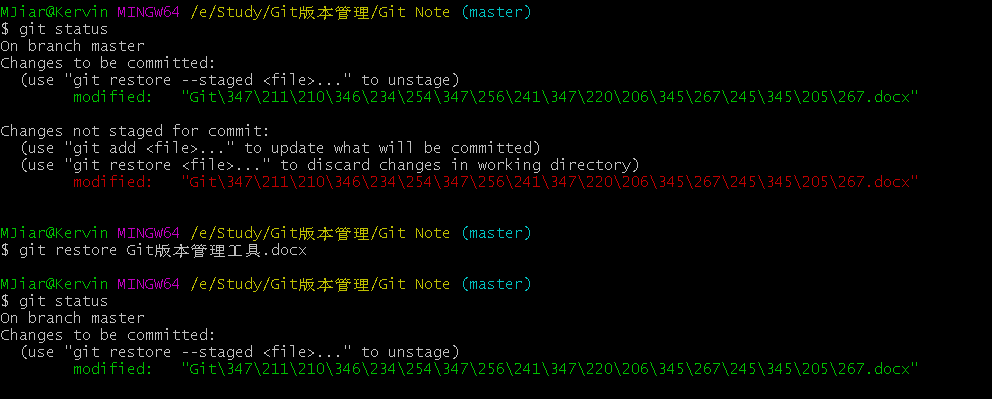
一般恢复文件会遇到这两种情况：

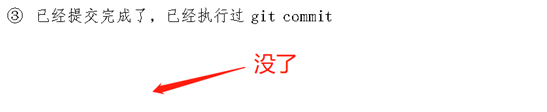
1. 还没添加到缓存区间，还没有执行 git add

执行指令：

|  |
| --- |
| git restore 文件名 |





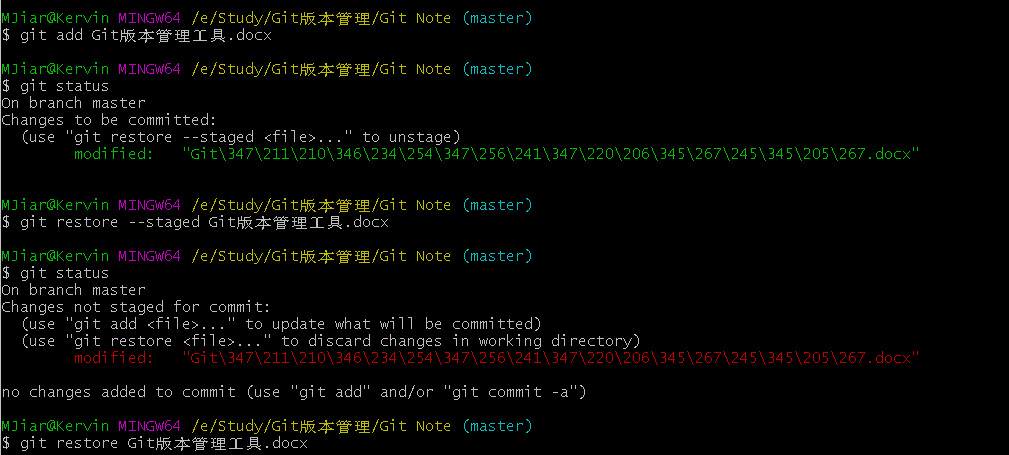


1. 对已经修改了的文件，已经添加到缓存区间，已经执行过 git add

执行指令：

|  |
| --- |
| git add 文件名  git restore --staged 文件名  git restore 文件名 |







### 12、git rm 删除文件

1. 对于已经完成提交了的项目文件，不需要版本控制系统继续管理，可以针对某个文件进行移除，则会把文件从本地以及版本控制系统进行**完全丢弃**，执行指令：

|  |
| --- |
| git rm 文件名 |

1. 对于已经完成提交了的项目文件，不需要版本控制系统继续管理，但有需要保留在本地时，执行指令：

|  |
| --- |
| git rm –cached 文件名 |

1. 批量删除，可用\*通配符实现，执行指令：

|  |
| --- |
| git rm \*.o |

**但请记得，删除操作也是需要执行commit指令进行提交！！！**

### 13、git mv 移动文件

执行指令：

|  |
| --- |
| git mv 文件名 目标路径/ |

△首先移动文件时，如果需要移动到其他文件夹，请确保目标路径存在这么个文件夹



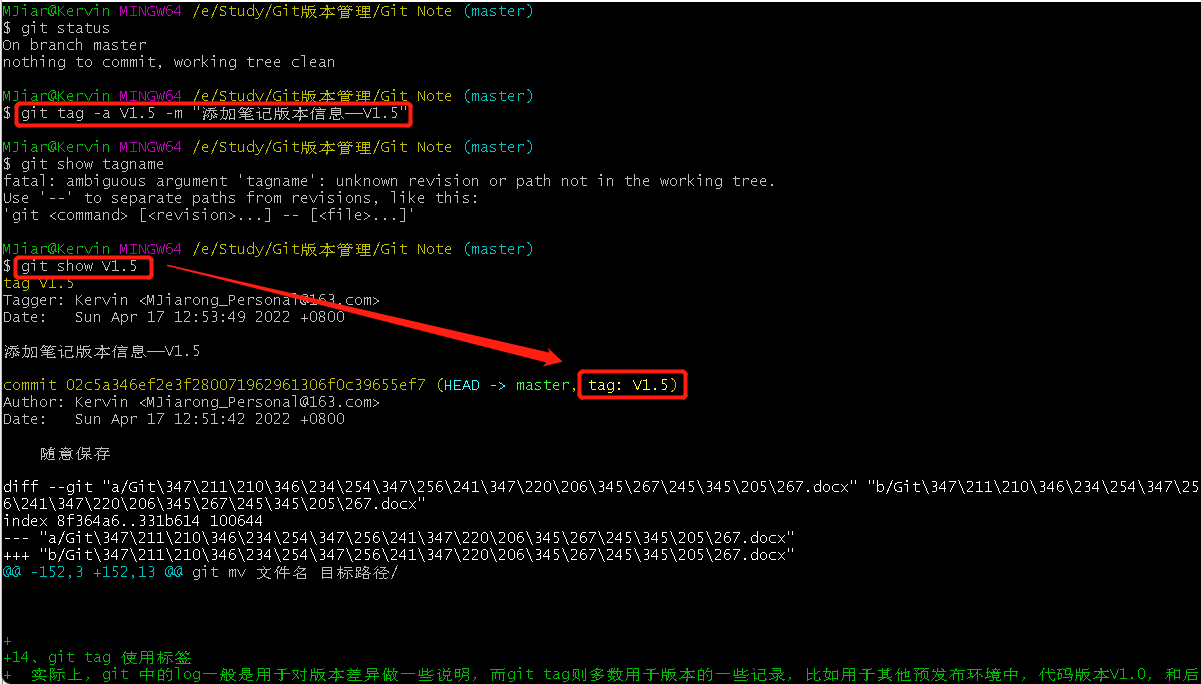




### 14、git tag 使用标签

实际上，**git 中的log一般是用于对版本差异做一些说明**，而**git tag则多数用于版本的一些版本编码**，比如用于其他预发布环境中，代码版本V1.0，和后来完善后用于发布的环境代码V2.0。

|  |
| --- |
| **git tag -a tagname -m “提交信息”**  git tag tagname 轻量标签  **git tag -a tagname 某版本哈希值 -m “提交信息” //忘记给某些版本打标签**  git show tagname  **git tag** //获取所有标签  git tag -l ‘v1.8\*’ //获取拥有某一部分特征的标签  **git tag -d 标签名称 //删除标签** |



补录标签：

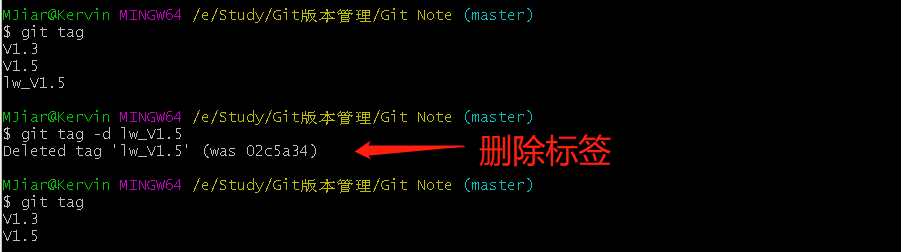




轻量标签：



删除标签：



### 14、git checkout 检出

如果当前这个版本在使用上有任何问题，需要进行版本回退时，回退到上一次上线的版本时。

1. 可以先通过git log 查看上一次上线的版本哈希值是多少
2. 使用git checkout 目标版本哈希值，可以把头指针指向目标版本
3. 使用 git log无法查看头指针以后的版本信息，需要使用git reflog

|  |
| --- |
| git checkout 版本号  git reflog  git checkout 标签名 |







## Git版本管理的经验洽谈

### 1、HEAD detached from XXX（HEAD 游离）

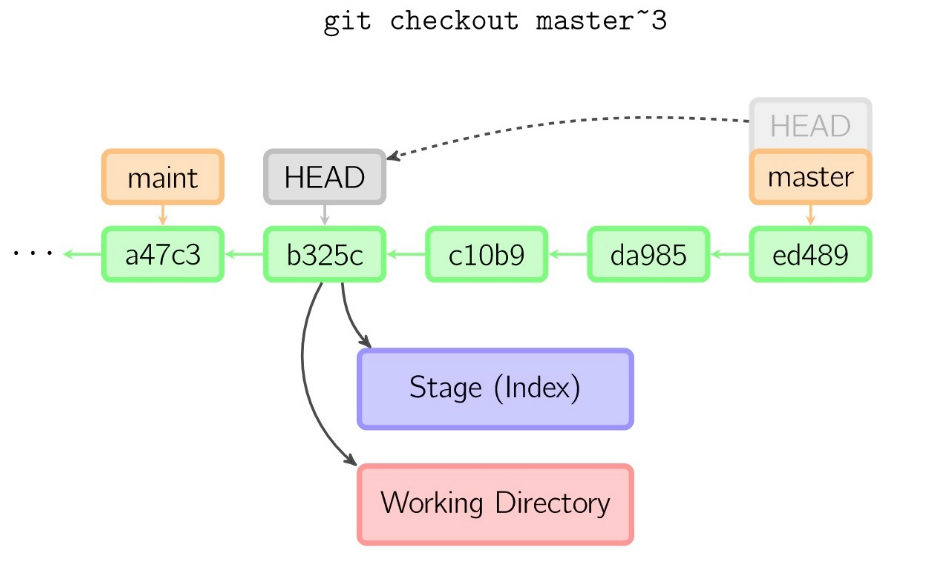
先说说为什么会出现这种情况，Head指针可以指向任意一个分支，一般可以通过

|  |
| --- |
| git checkout 分支名 |

的方式，进行切换分支，但如果用户选择直接检出到任一提交ID

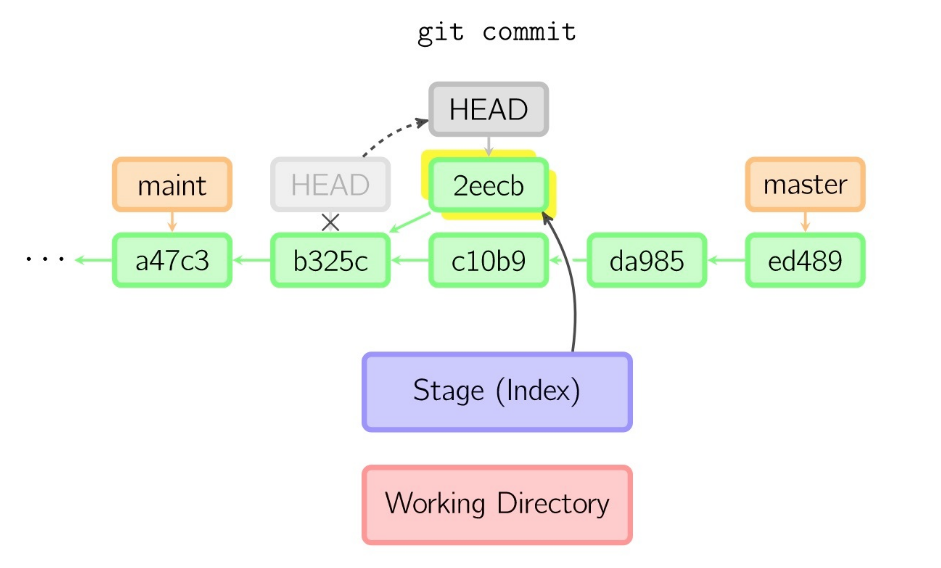
|  |
| --- |
| git checkout commitID |

快速地切换到某一次提交，此时Head指针就会处于游离状态。



游离状态优点：能很方便得切换到某次提交

缺点：在这个游离状态下的提交是会新开一个匿名分支，提交保存无法可见，一旦切换到别的分支，这个游离状态以后的提交则无法追溯了



解决方法：可以通过新建一个分支，保存游离状态后的提交，最后合并，并删除原来的临时分支

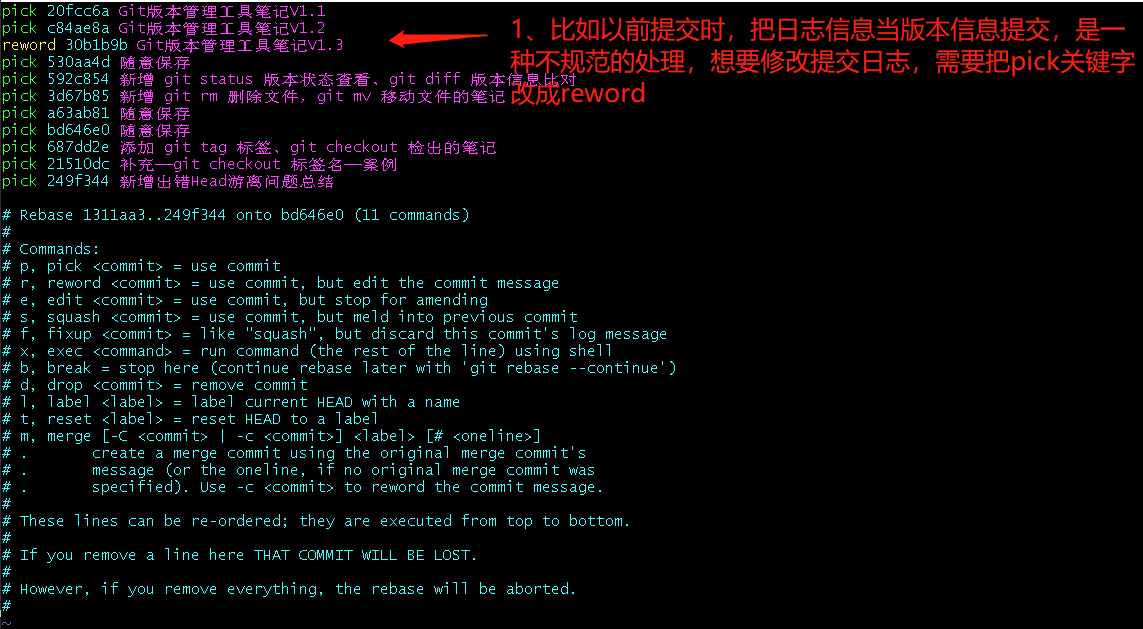


### 2、修改以前的版本日志

如果在以前上传日志时随意上传，或者后来想修改上传时的日志信息，整理好所有的日志记录，使用下面的指令：

|  |
| --- |
| git commit –amend //修改最近提交的版本日志  git rebase -i HEAD~3 //修改最近3次的历史日志 |

1. 把需要修改的版本前的关键字 pick 修改成 reword



1. Esc+:wq退出保存
2. 重新进入后，利用vi编辑器对关键字后的日志内容进行修改，再次退出保存