思考题

thinking 0.1

题目：

> **Thinking 0.1** 思考下列有关 Git 的问题：
>
> - 在前述已初始化的 `~/learnGit` 目录下，创建一个名为 README.txt 的文件。执行命令 `git status > Untracked.txt`（其中的 `>` 为输出重定向，我们将在 0.6.3 中详细介绍）。
> - 在 README.txt 文件中添加任意文件内容，然后使用 `add` 命令，再执行命令 `git status > Stage.txt`。
> - 提交 README.txt，并在提交说明里写入自己的学号。
> - 执行命令 `cat Untracked.txt` 和 `cat Stage.txt`，对比两次运行的结果，体会 README.txt 两次所处位置的不同。
> - 修改 README.txt 文件，再执行命令 `git status > Modified.txt`。
> - 执行命令 `cat Modified.txt`，观察其结果和第一次执行 `add` 命令之前的 status 是否一样，并思考原因。
>
> ■

指令模拟过程：

```
git@23371084:/23371084/learnGit (master)$ vim README.txt
git@23371084:/23371084/learnGit (master)$ git status > Untracked.txt
git@23371084:/23371084/learnGit (master)$ vim README.txt
git@23371084:/23371084/learnGit (master)$ git add README.txt
git@23371084:/23371084/learnGit (master)$ git status > Stage.txt
git@23371084:/23371084/learnGit (master)$ git commit -m "23371-84"
[master 553f38b] 23371-84
 1 file changed, 1 insertion(+), 1 deletion(-)
git@23371084:/23371084/learnGit (master)$ cat Untracked.txt
位于分支 master
尚未暂存以备提交的变更：
  （使用 "git add <文件>..." 更新要提交的内容）
  （使用 "git restore <文件>..." 丢弃工作区的改动）
        修改：      README.txt

未跟踪的文件：
  （使用 "git add <文件>..." 以包含要提交的内容）
        Untracked.txt

修改尚未加入提交（使用 "git add" 和/或 "git commit -a"）
git@23371084:/23371084/learnGit (master)$ cat Stage.txt
位于分支 master
要提交的变更：
  （使用 "git restore --staged <文件>..." 以取消暂存）
        修改：      README.txt

未跟踪的文件：
```

```
未跟踪的文件:
  (使用 "git add <文件>..." 以包含要提交的内容)
        Stage.txt
        Untracked.txt

git@23371084:/23371084/learnGit (master)$ vim README.txt
git@23371084:/23371084/learnGit (master)$ git status > Modified.txt
git@23371084:/23371084/learnGit (master)$ cat Modified.txt
位于分支 master
尚未暂存以备提交的变更:
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git restore <文件>..." 丢弃工作区的改动)
        修改:       README.txt

未跟踪的文件:
  (使用 "git add <文件>..." 以包含要提交的内容)
        Modified.txt
        Stage.txt
        Untracked.txt

修改尚未加入提交(使用 "git add" 和/或 "git commit -a")
git@23371084:/23371084/learnGit (master)$
```

思考：git status 可以查看当前文件的状态（未追踪，已暂存未提交，已提交）

文件	状态变化
`Untracked.txt`	`README.txt` 处于 Untracked（未追踪）状态
`Stage.txt`	`README.txt` 处于 Staged（已暂存） 状态
`Modified.txt`	`README.txt` 处于 Modified（已修改但未提交） 状态

指令分析：

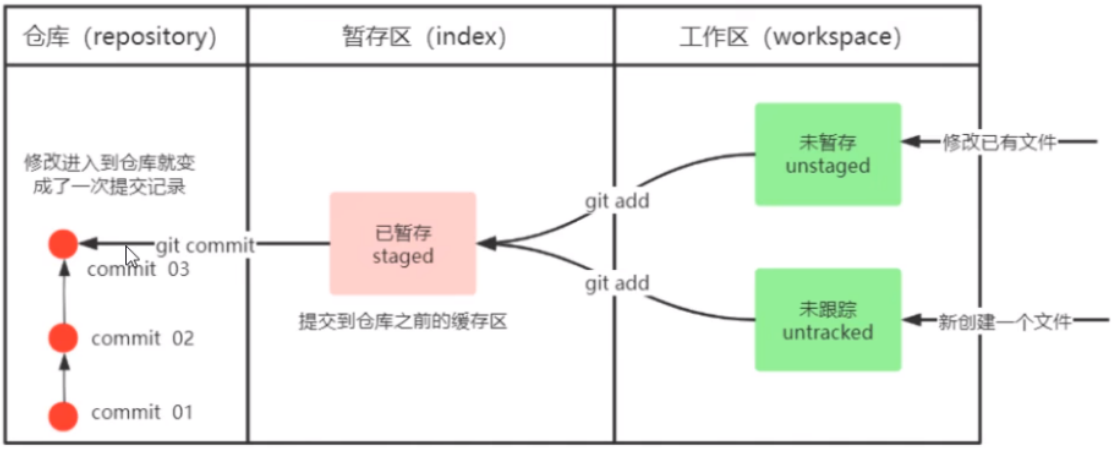
- `git add` 将文件从 Untracked（未追踪）变为 Staged（已暂存）
- `git commit` 将 Staged 文件提交到 Git 历史记录
- 重新修改文件后，它从 Tracked（已追踪）变为 Modified（已修改），需要重新 `git add` 和 `git commit`

thinking 0.2

题目：

> Thinking 0.2 仔细看看0.10，思考一下箭头中的 add the file 、stage the file 和 commit 分别对应的是 Git 里的哪些命令呢？ ■

git 工作模式示意图：



总结：add the file 和 stage the file 对应 git add 指令，add 本质上就是 staging 的过程，"staging" 是指将文件放入 Git 的暂存区（Staging Area），等待提交；commit the file 对应 git commit 指令，会将暂存区中的文件保存到 Git 本地仓库，形成一个新的提交（commit）

thinking 0.3

题目：

> **Thinking 0.3** 思考下列问题：
>
> 1. 代码文件 **print.c** 被错误删除时，应当使用什么命令将其恢复？
> 2. 代码文件 **print.c** 被错误删除后，执行了 **git rm print.c** 命令，此时应当使用什么命令将其恢复？
> 3. 无关文件 **hello.txt** 已经被添加到暂存区时，如何在不删除此文件的前提下将其移出暂存区？

①如果 `print.c` 还存在于 Git 的版本管理中（即 `git rm` 未执行），可以使用以下命令恢复：

```
git checkout -- print.c
```

`git checkout -- print.c` 可以从最新的提交版本中恢复 `print.c`

但这个方法**仅适用于** `print.c` 已被 Git 追踪过，并且仍然存在于 Git 历史记录中

②如果已经执行了 `git rm print.c`，但还**未提交**（即 `git commit` 还没有执行），可以使用：

```
git checkout -- print.c
```

③如果 `git commit` 也已经执行了，则需要先回退版本，再进行恢复：

```
git reset HEAD~  # 撤销最近的一次提交
git checkout -- print.c  # 恢复文件
```

或者如果不想撤销整个提交，而只是恢复 `print.c`：

```
git checkout HEAD^ -- print.c
```

`git reset HEAD~`：撤销最近的一次提交，回到 `git commit` 之前的状态

`git checkout HEAD^ -- print.c`：从前一次提交恢复 `print.c`，但不会影响其他文件

④如果 `hello.txt` 只是被 `git add` 了，还没有 `commit`，可以用：

```
git reset HEAD hello.txt
```

`git reset HEAD hello.txt`：将 `hello.txt` **从暂存区移除**，但不会删除本地文件，之后 `hello.txt` 仍然存在于工作目录中，不会影响其他提交

示例：

```
git@23371084:/23371084/learnGit (master)$ rm README.txt
git@23371084:/23371084/learnGit (master)$ ls
git@23371084:/23371084/learnGit (master)$ git checkout -- README.txt
git@23371084:/23371084/learnGit (master)$ ls
README.txt
git@23371084:/23371084/learnGit (master)$
```

thinking 0.4

题目：

> **Thinking 0.4** 思考下列有关 Git 的问题：
>
> - 找到在 `/home/22xxxxxx/learnGit` 下刚刚创建的 `README.txt` 文件，若不存在则新建该文件。
> - 在文件里加入 `Testing 1`，`git add`，`git commit`，提交说明记为 `1`。
> - 模仿上述做法，把 1 分别改为 2 和 3，再提交两次。
> - 使用 `git log` 命令查看提交日志，看是否已经有三次提交，记下提交说明为 3 的哈希值[a]。
> - 进行版本回退。执行命令 `git reset --hard HEAD^` 后，再执行 `git log`，观察其变化。
> - 找到提交说明为 1 的哈希值，执行命令 `git reset --hard <hash>` 后，再执行 `git log`，观察其变化。
> - 现在已经回到了旧版本，为了再次回到新版本，执行 `git reset --hard <hash>`，再执行 `git log`，观察其变化。
>
> [a]使用 `git log` 命令时，在 `commit` 标识符后的一长串数字和字母组成的字符串

三次修改以及提交：

```
git@23371084:/23371084/learnGit (master)$ git log
commit 7bcbf7186f64864f93360bf439b3f5dc3d3eca9a (HEAD -> master)
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:54 2025 +0800

    3

commit 036d04d15fe64d8a52c866b579b41de71a5a790f
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:29 2025 +0800

    2

commit ec898608b292909416a05bf90f46bfe0f4727a64
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:06 2025 +0800
```

git log 可以显示历次的提交信息，而使用 git log –all –abbrev-commit –graph –decorate 则可以精简显示信息如下：

```
git@23371084:/23371084/learnGit (master)$ git log --all --decorate --graph
-abbrev-commit
* commit 7bcbf71 (HEAD -> master)
| Author: 李昊宸 <23371084@buaa.edu.cn>
| Date:   Thu Mar 13 08:05:54 2025 +0800
|
|     3
|
* commit 036d04d
| Author: 李昊宸 <23371084@buaa.edu.cn>
| Date:   Thu Mar 13 08:05:29 2025 +0800
|
|     2
|
* commit ec89860
  Author: 李昊宸 <23371084@buaa.edu.cn>
  Date:   Thu Mar 13 08:05:06 2025 +0800

      1
```

使用 git reset –hard HEAD^ 可以回到当前提交的上一次提交状态，即从 3 修改回到 2 修改状态：

```
git@23371084:/23371084/learnGit (master)$ git reset --hard HEAD^
HEAD 现在位于 036d04d 2
git@23371084:/23371084/learnGit (master)$ git log
commit 036d04d15fe64d8a52c866b579b41de71a5a790f (HEAD -> master)
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:29 2025 +0800

    2

commit ec898608b292909416a05bf90f46bfe0f4727a64
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:06 2025 +0800

    1
```

使用 git reset –gard 可以回退到任何一个提交状态，也可以在回退之后再次回到后面的提交状态，即可以从 3 回到 1，也可以再从 1 回到 3，这里的 值只需要前 7 位即可：

```
git@23371084:/23371084/learnGit (master)$ git reset --hard ec898608b29290941
6a05bf90f46bfe0f4727a64
HEAD 现在位于 ec89860 1
git@23371084:/23371084/learnGit (master)$ git log
commit ec898608b292909416a05bf90f46bfe0f4727a64 (HEAD -> master)
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:06 2025 +0800

    1
git@23371084:/23371084/learnGit (master)$ git reset --hard 7bcbf7186f64864f9
3360bf439b3f5dc3d3eca9a
HEAD 现在位于 7bcbf71 3
git@23371084:/23371084/learnGit (master)$ git log
commit 7bcbf7186f64864f93360bf439b3f5dc3d3eca9a (HEAD -> master)
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:54 2025 +0800

    3

commit 036d04d15fe64d8a52c866b579b41de71a5a790f
Author: 李昊宸 <23371084@buaa.edu.cn>
Date:   Thu Mar 13 08:05:29 2025 +0800

    2
```

总结：本例介绍了 git reset –hard 的用法，可以用以下两种形式

git reset –hard HEAD^ 回退到上一个提交状态

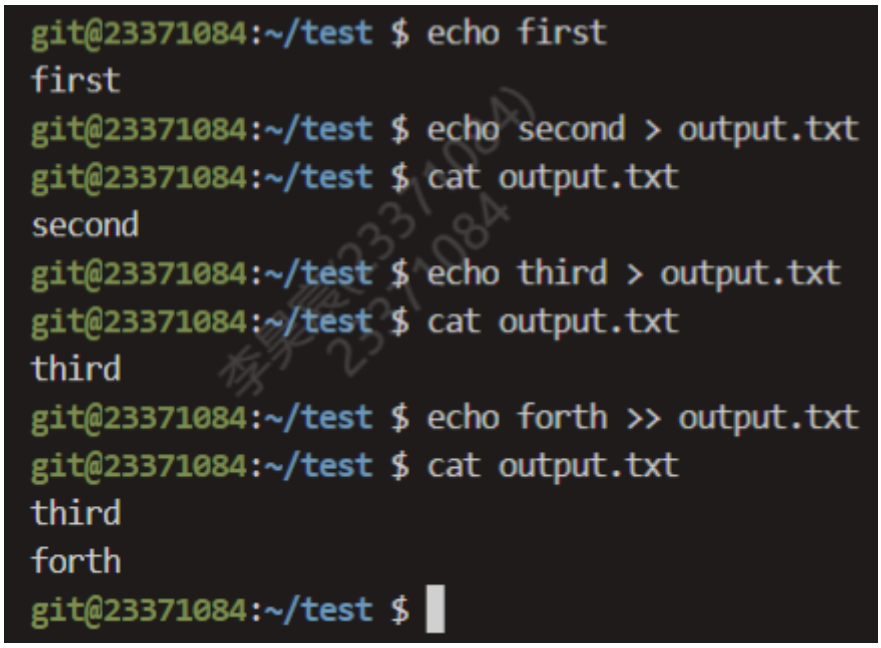
git reset –hard 回退到任意一个哈希值为 hash 的提交状态

thinking 0.5

题目：

> Thinking 0.5 执行如下命令，并查看结果
>
> - `echo first`
> - `echo second > output.txt`
> - `echo third > output.txt`
> - `echo forth >> output.txt`

执行结果：



总结：'>' 会用新的标准输出覆盖被输出文件的原有内容，'>>' 会把新的标准输出拼接到被输出文件的原有内容之后

thinking 0.6

> Thinking 0.6 使用你知道的方法（包括重定向）创建下图内容的文件（文件命名为 `test`），将创建该文件的命令序列保存在 `command` 文件中，并将 `test` 文件作为批处理文件运行，将运行结果输出至 `result` 文件中。给出 `command` 文件和 `result` 文件的内容，并对最后的结果进行解释说明（可以从 `test` 文件的内容入手）。具体实现的过程中思考下列问题：`echo echo Shell Start` 与 `echo` `` `echo Shell Start` `` 效果是否有区别；`echo echo $c>file1` 与 `echo` `` `echo $c>file1` `` 效果是否有区别。

command 内容：command 为具有可执行权限的 sh 文件

```
#! /bin/bash

output=$1

echo 'echo "Shell Start..."' > $output
echo 'echo "set a = 1"' >> $output
echo 'a=1' >> $output
echo 'echo "set b = 2"' >> $output
echo 'b=2' >> $output
echo 'echo "set c = a+b"' >> $output
echo 'c=$[$a+$b]' >> $output
echo 'echo "c = $c"' >> $output
echo 'echo "save c to ./file1"' >> $output
echo 'echo "$c>file1"' >> $output
echo 'echo "save b to ./file2"' >> $output
echo 'echo "$b>file2"' >> $output
echo 'echo "save a to ./file3"' >> $output
echo 'echo "$a>file3"' >> $output
echo 'echo "sace file1 file2 file3 to file4"' >> $output
echo 'cat file1>file4' >> $output
echo 'cat file2>>file4' >> $output
echo 'cat file3>>file4' >> $output
echo 'echo "save file4 to ./result"' >> $output
echo 'cat file4>>result' >> $output
```

test内容：test 为具有可执行权限的 sh 文件

```
echo "Shell Start..."
echo "set a = 1"
a=1
echo "set b = 2"
b=2
echo "set c = a+b"
c=$[$a+$b]
echo "c = $c"
echo "save c to ./file1"
echo "$c>file1"
echo "save b to ./file2"
echo "$b>file2"
echo "save a to ./file3"
echo "$a>file3"
echo "sace file1 file2 file3 to file4"
cat file1>file4
cat file2>>file4
cat file3>>file4
echo "save file4 to ./result"
cat file4>>result
```

总结：关键在于 echo 输出内容加单引号，不加引号，加双引号的区别，不加引号或加双引号的时候，echo 输出内容的所有 $ 变量会被解析，有值的赋值后输出，空变量输出空串；加单引号的时候，不进行任何变量解析， 会把字面量原样输出

①两次执行命令的区别：

```
git@23371084:~/test $ echo echo Shell Start
echo Shell Start
git@23371084:~/test $ echo `echo Shell Start`
Shell Start
git@23371084:~/test $
```

第一条命令：直接输出 echo Shell Start 这个字符串的内容到标准输出

第二条命令：把 echo Start Shell 这条命令的标准输出捕获，并且输出到标准输出，即为 Shell Start

②两次执行命令的区别：

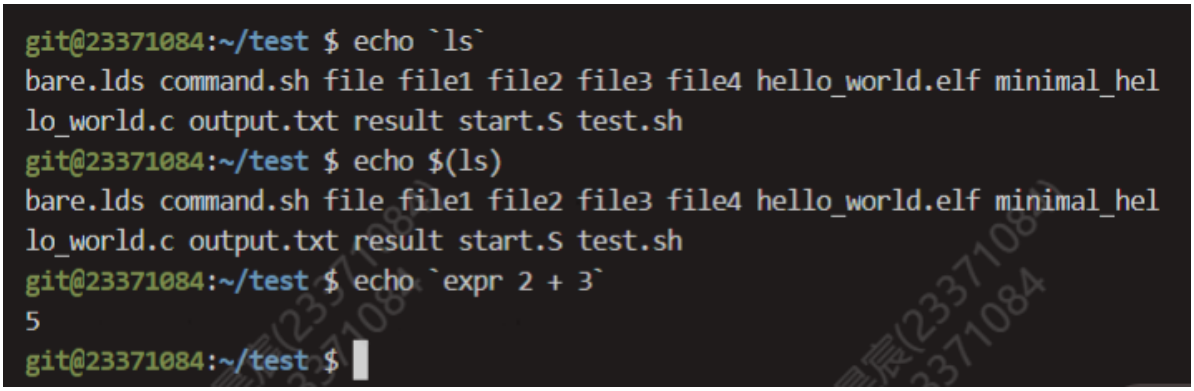
第一条命令：把 echo $c 这个字符串重定向输出到 file1 文件中，其中 c 的值会被代入，因为 c 本身是空，所以 file1 文件中只有 echo 这个字符串

第二条命令：同理，用双引号引起来和不用双引号都会把字符串中的变量代入相应的引用值

第三条命令：用单引号引起来的内容不会进行任何变量值引用，而是把字符串的内容原样输出，所以 file1 的文件就是 echo $c 这个字符串

第四条命令：echo 捕获并输出 echo $c > file1 这条命令的标准输出，而这条命令没有标准输出，所以输出为空

③其他例子：



总结：echo 捕获的是某条命令的**标准输出**，无特殊情况时，会把其捕获到的标准输出输出到 stdout 中，可以用 echo 来捕获函数的返回值（这里的返回值指 stdout 而不是 exit code）

作业难点

T1

（1）完成一个简单的 C 语言判断回文数即可

（2）使用 Makefile 编译并链接形成可执行文件

（3）简单的 Shell 语法，提取文件的某些特定行并且重定向输出到目标文件中，后者可以用 sed p 参数结合管道符来实现

思考：

区别传参和标准输入，本例要求的是传参，即来自参数 $

```
input=$1
outut=$2
```

标准输入是来自 stdin：

```
read input
read outuput
```

（4）文件的复制，使用 cp 指令即可

思考：区别 mv 和 cp，前者是剪切，后者是复制

T2

完成简单的 shell 脚本，考察对 if 语句，判断语句，循环语句的掌握，代码如下：

```
#! /bin/bash

for((i=1;i<=100;i++))
do
    if [ $i -ge '71' ]
    then
        rm "file${i}" -r
    fi
    if [ $i -ge '41' ] && [ $i -lt '71' ]
    then
        mv "file${i}" "newfile${i}"
    fi
done
```

总结：test 指令可以用 [] 来简单代替，也可以用更加安全的 [[]] 来代替

T3

提取文件中包含特定内容的某些行，只输出行号到目标文件中，可以使用 grep，管道，结合 awk 命令实现，因为 C 文件中行号和正文中以：分开，因此我们可以规定 awk 中的输入分隔符为：，取第一列输出即可，代码如下：

```
grep -n "int" $input | awk -v FS=':' '{print $1}' > $output
```

T4

（1）修改文件中的某个字符串为新内容，并把修改后内容写回原文件，可以用 sed 命令结合其内置参数 s 配合 g 实现，代码如下：

```
sed -i "s/$old/$new/g" $input
```

（2）完成 Makefile 文件的递归调用

总结：递归调用 Makefile 文件的格式如下

我们可以在一个 makefile 文件中调用其他目录下的 makefile 文件

命令	作用
`$(MAKE) -C subdir`	进入 `subdir` 目录，执行该目录的 `Makefile`
`$(MAKE) -C subdir clean`	在 `subdir` 目录执行 `clean` 目标

示例：

```
all:
  $(MAKE) -C src
```

递归调用也是一条命令（command），最好写在 all 的内部

实验感想

本次上机，exam 部分并未出现太大问题，主要问题出在 extr 部分，有以下需要改进的地方：

1. 对于 shell 的扩展指令理解不够全面，比如替换文件后缀，如何遍历一个目录下的所有文件等常用操作，不能第一时间想到对应的指令，导致思路不连贯；同时对于 sed 替换，grep 截取等语法记忆不牢，需要经常翻阅 help 文档，耽误大量调试时间
2. 知识面不够宽，对于 sort 函数，软链接等知识没有提前接触，现场学习耗费太多时间

参考文献

【黑马程序员Git全套教程，完整的git项目管理工具教程，一套精通git】 https://www.bilibili.com/video/BV1MU4y1Y7h5/?share_source=copy_web&vd_source=069a61898eba498b92493563cbd57459
————其中有关 git 的理解和部分示意图来自此教程